

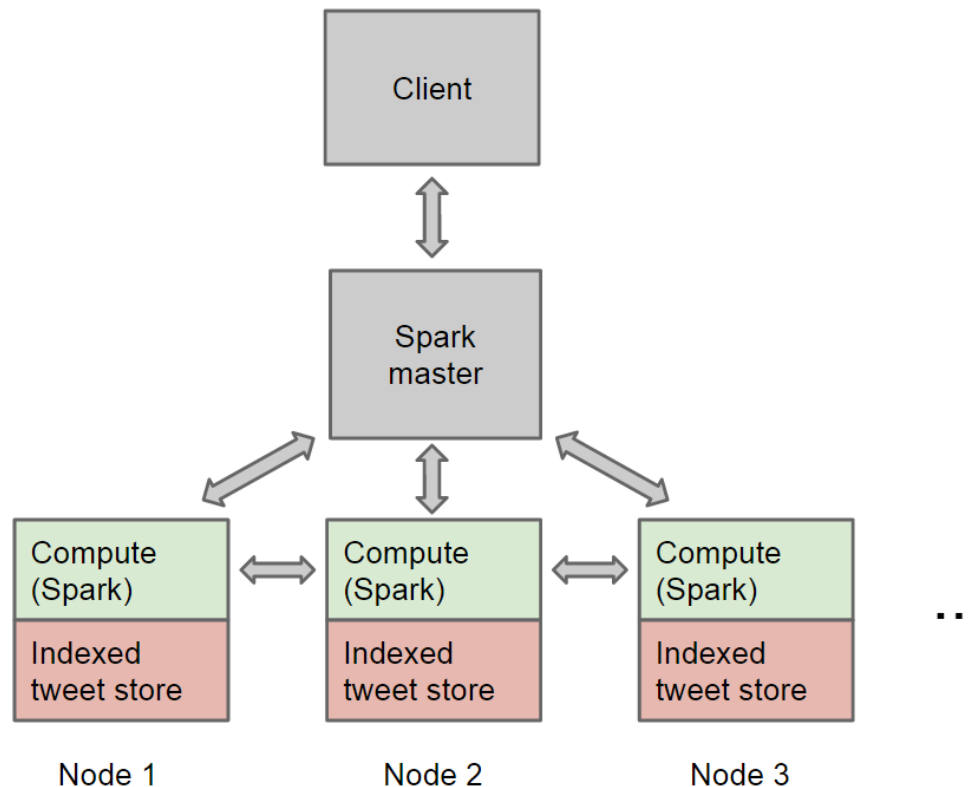
Delta Processing for Social Analytics

James Thomas, Xinhao Yuan

Myeongjae Jeon, Jorgen Thelin,
Lidong Zhou

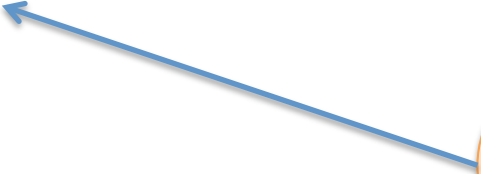
DSoAP System Architecture (from before)

- 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

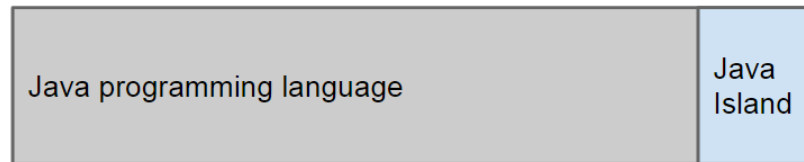
Motivation

- Honing in on target tweet set with plain text search

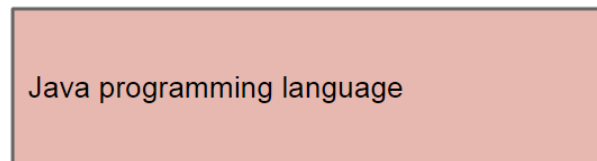
Search 1: "Java"



Search 2: "Java
AND (NOT coffee)
AND (NOT brew)"



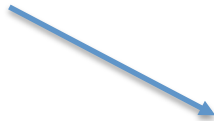
Search 3: "Java
AND (Oracle OR
JVM OR language
....)"



Problem

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

Oh no...

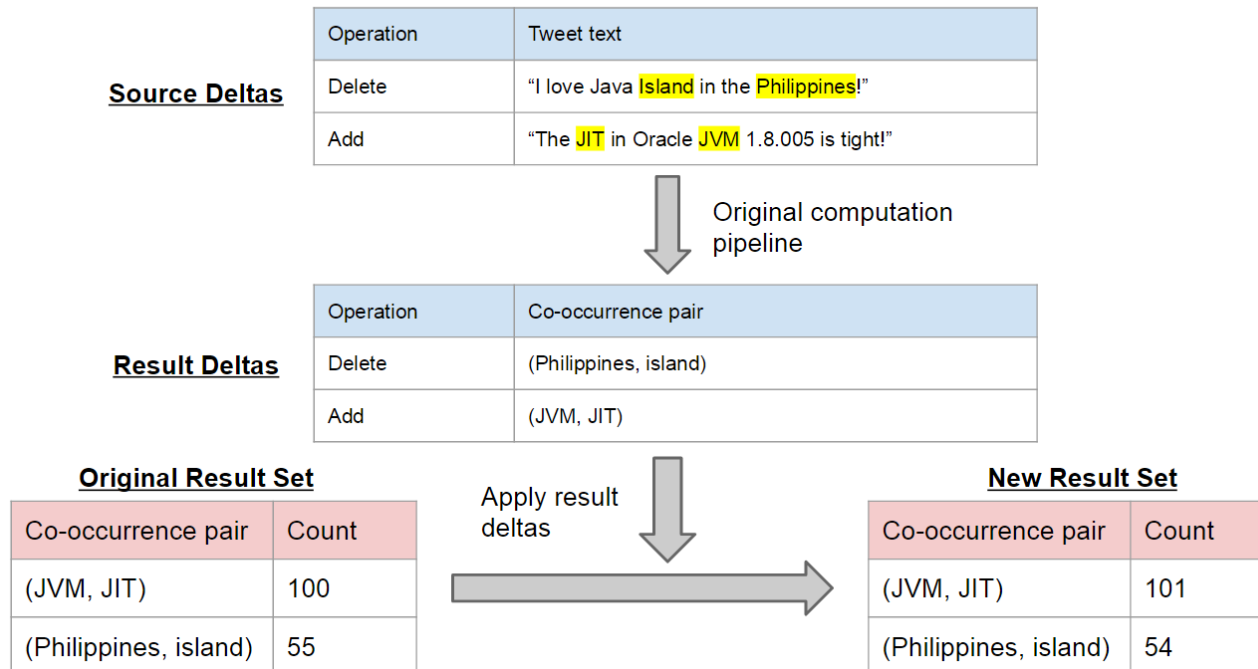


Co-occurrence pair	# of appearances
(JVM, JIT)	100
(Philippines, island)	55
(Scala, functional)	30

- 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

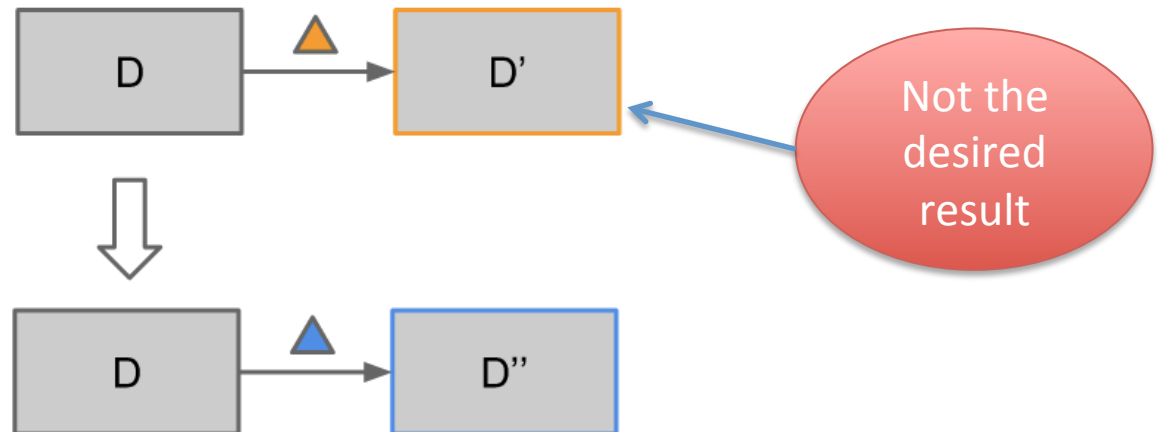


Design Goals / Limits of Prior Work

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

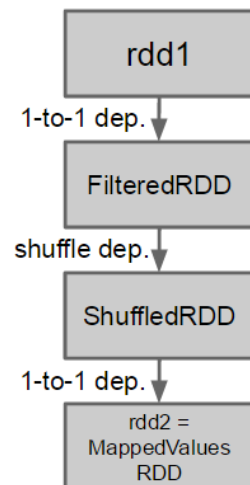


Spark Review

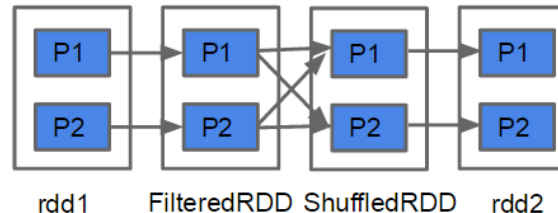
- Partitioned datasets (RDDs)
- One-to-one and shuffle dependencies (narrow and wide)
- DAG of dependencies
- Programmer-specified caching of datasets in memory

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

Dependency DAG



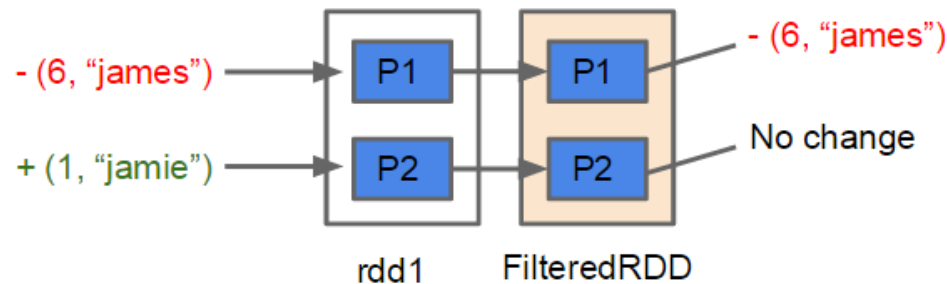
Physical Execution



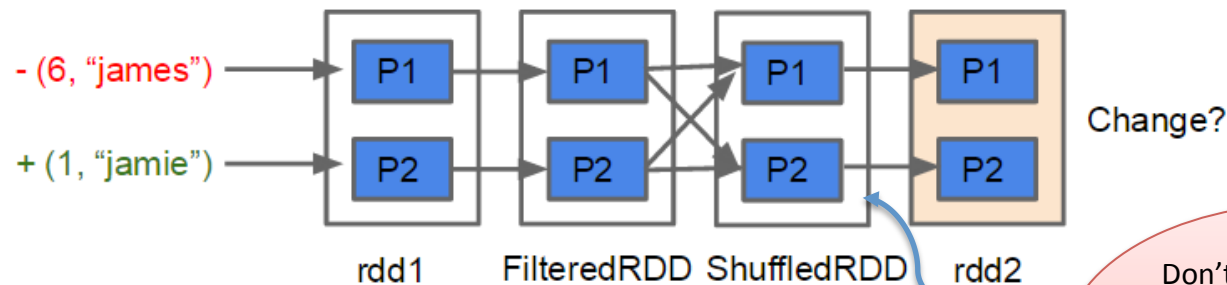
Feeding Deltas through Computation

```
rdd2 =  
rdd1.filter(t=>t._1>5)  
      .groupByKey()  
      .mapValues(v=>v.min)
```

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

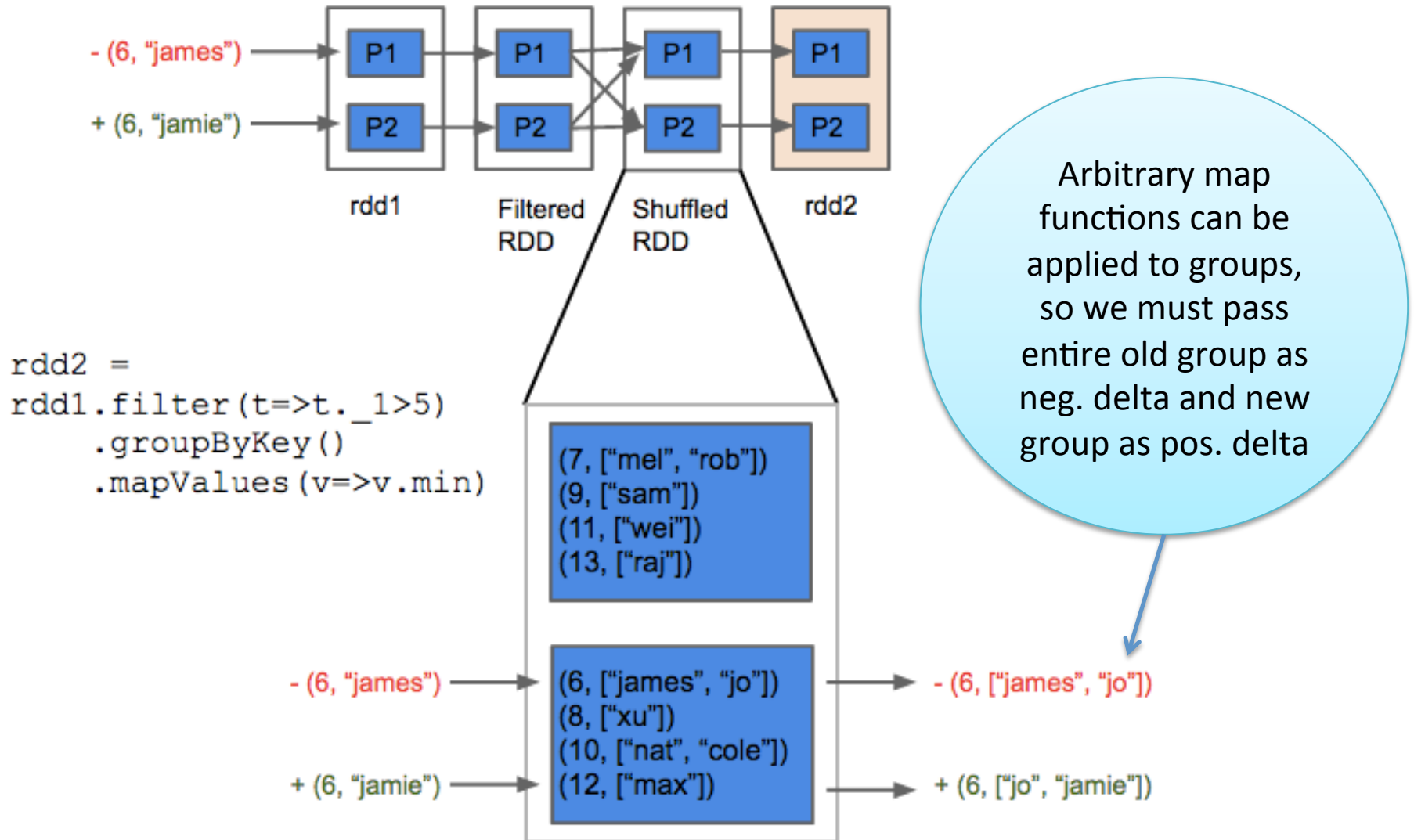


Case 2: shuffle transforms (many-to-one) as well -- must save intermediate results



Don't know other keys
in group 6, so can't
pass deltas through
this stage without
saving ShuffledRDD

Feeding Deltas through Shuffle



Target API (Design Goals 3 & 4)

- `d = new DeltaComputation(rdd)`
 - `rdd` can be any map-filter-groupby computation
- `newD = d.applyDeltas(pos, neg)`
 - `pos`, `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
- `result = newD.getResult()`
 - `result` is an RDD on which any Spark transformations (incrementalizable or not) can be run

API Example

```
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)
...
```

Mixing in non-incremental computation

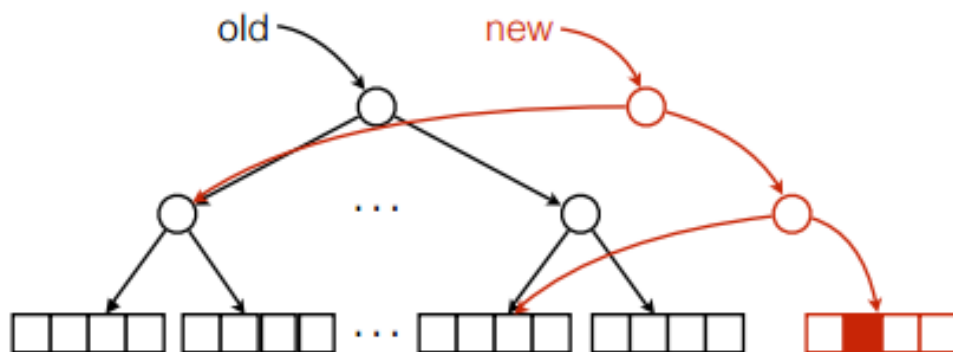
Handle to old version allows us to apply deltas to it

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)



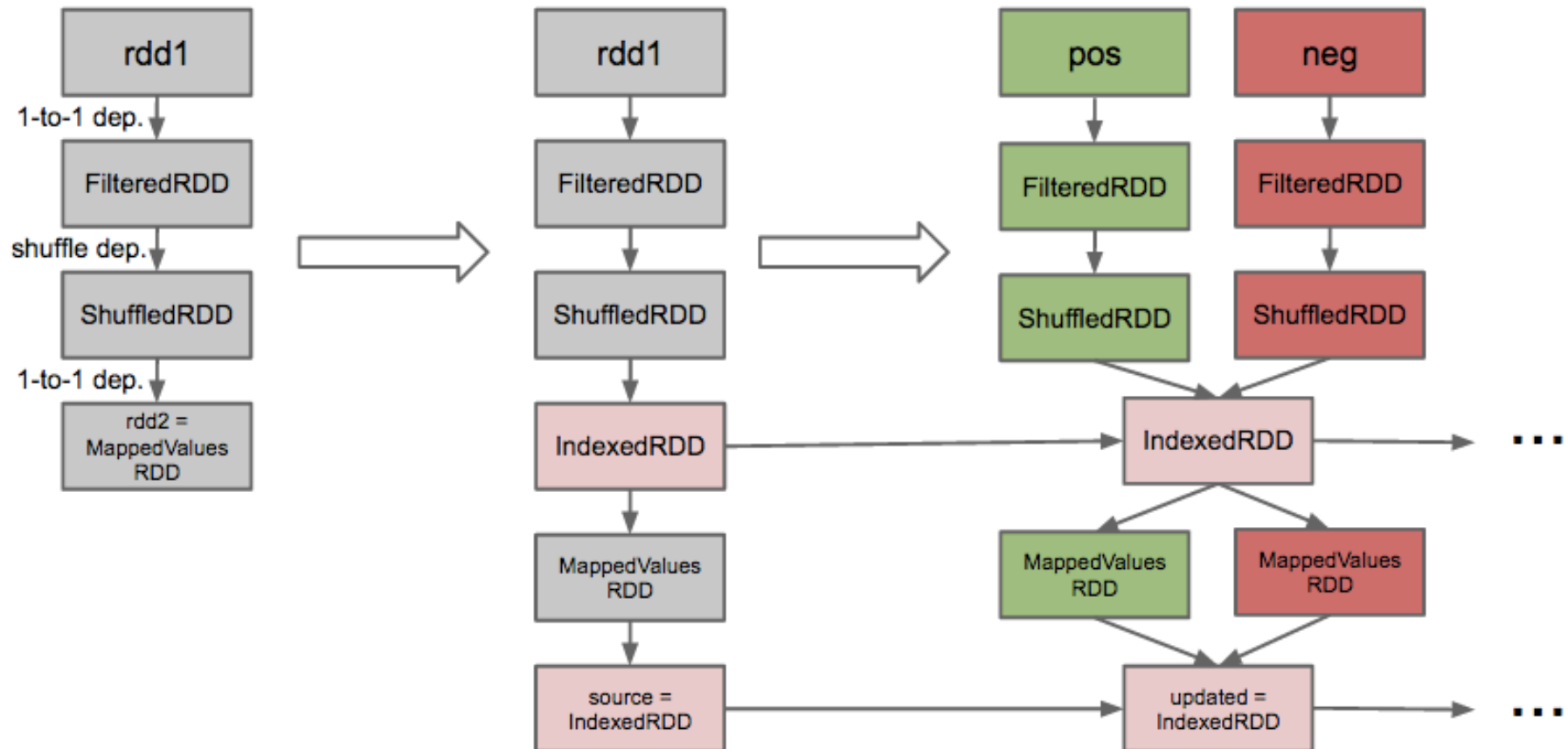
(fig. by Ankur Dave, AMP Lab)

Computation DAG Rewriter (Design Goals 1 & 2)

```
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



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



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