Summingbird: A Framework for Integrating Batch and Online MapReduce Computations

Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin

VLDB 2014
Thursday, September 4, 2014
Scene:
Internet company in Silicon Valley (circa 2010)

Standard data science task:
What have people been clicking on?
Simple!

Write some Pig…

```scala
raw = load '/logs/' using LogLoader();

a = filter raw by action == 'click';
b = group a by target;
c = foreach b generate COUNT(a), group;

store c into 'counts/';
```

Or some Scalding (more recently)…

```scala
val input = TypedTsv[(String, String)]("/logs")
val raw = TypedPipe.from(input)

raw.groupBy { case (target, action) => target } .size
  .write(TypedTsv("counts"))
```
Standard data science task:
What have people been clicking on?

Now try:
What have people been clicking on right now?

*grumble*  *ugh*  *hrmmm*  

Two major pain points (circa 2010):
1. Lack of a standardized online processing framework
2. Having to write everything twice
State of the industry (circa 2013):
Good handle on batch processing at scale
Increasing convergence on online processing frameworks

Storm

Diagram:
- Spout
- Bolt
- Bolt
- Bolt
- memcached
Two major pain points:

1. Lack of a standardized online processing framework
   ✓ Widespread adoption of Storm at Twitter

2. Having to write everything twice
   The point of this work…
Summingbird

A domain-specific language (in Scala) designed to integrate batch and online MapReduce computations

Idea #1: Algebraic structures provide the basis for seamless integration of batch and online processing

Idea #2: For many tasks, close enough is good enough

Probabilistic data structures as monoids
Primary goal is developer productivity
Optimizations can come later...

Scope: “the easy problems”
counting etc. (min, max, mean, moments…)
set membership
histograms
Batch and Online MapReduce

“map”

flatMap[T, U](fn: T => List[U]): List[U]

map[T, U](fn: T => U): List[U]

filter[T](fn: T => Boolean): List[T]

“reduce”

sumByKey
Semigroup = ( \( M, \oplus \) )
\[ \oplus : M \times M \rightarrow M, \text{s.t., } \forall m_1, m_2, m_3 \ni M \]
\[ (m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3) \]

Monoid = Semigroup + identity
\[ \varepsilon \text{ s.t., } \varepsilon \oplus m = m \oplus \varepsilon = m, \forall m \ni M \]

Commutative Monoid = Monoid + commutativity
\[ \forall m_1, m_2 \ni M, m_1 \oplus m_2 = m_2 \oplus m_1 \]

Simplest example: integers with + (addition)
Idea #1: Algebraic structures provide the basis for seamless integration of batch and online processing.

Summingbird values must be at least semigroups (most are commutative monoids in practice).

Power of associativity =

You can put the parentheses anywhere!

(a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f)
((((((a ⊕ b) ⊕ c) ⊕ d) ⊕ e) ⊕ f))

Batch = Hadoop
Online = Storm
Mini-batches

Results are exactly the same!
def wordCount[P <: Platform[P]](
  source: Producer[P, String],
  store: P#Store[String, Long]) =
source.flatMap {
  sentence =>
    toWords(sentence).map(_ -> 1L)
}.sumByKey(store)

Scalding.run {
  wordCount[Scalding](
    Scalding.source[Tweet]("source_data"),
    Scalding.store[String, Long]("count_out")
  )
}

Storm.run {
  wordCount[Storm](
    new TweetSpout(),
    new MemcacheStore[String, Long]
  )
}

Summingbird Word Count
where data comes from
where data goes
“map”
“reduce”

Run on Scalding (Cascading/Hadoop)
read from HDFS
write to HDFS

Run on Storm
read from message queue
write to KV store
“Boring” monoids

addition, multiplication, max, min
moments (mean, variance, etc.)
sets
tuples of monoids
hashmaps with monoid values

More interesting monoids?
Idea #2: For many tasks, close enough is good enough!

“Interesting” monoids
- Bloom filters (set membership)
- HyperLogLog counters (cardinality estimation)
- Count-min sketches (event counts)

Common features
1. Variations on hashing
2. Bounded error
<table>
<thead>
<tr>
<th></th>
<th>Exact</th>
<th>Approximate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set membership</td>
<td>set</td>
<td>Bloom filter</td>
</tr>
<tr>
<td>Set cardinality</td>
<td>set</td>
<td>hyperloglog counter</td>
</tr>
<tr>
<td>Frequency count</td>
<td>hashmap</td>
<td>count-min sketches</td>
</tr>
</tbody>
</table>
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, Map[String, Long]]) =
  source.flatMap { query =>
    (query.getHour, Map(query.getQuery -> 1L))
  }.sumByKey(store)

**Task: count queries by hour**

**Exact with hashmaps**

```scala
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, Map[String, Long]]) =
  source.flatMap { query =>
    (query.getHour, Map(query.getQuery -> 1L))
  }.sumByKey(store)
```

**Approximate with CMS**

```scala
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, SketchMap[String, Long]])
  (implicit countMonoid: SketchMapMonoid[String, Long]) =
  source.flatMap { query =>
    (query.getHour,
     countMonoid.create((query.getQuery, 1L)))
  }.sumByKey(store)
```
(Left) Joins

Task: count expanded URLs

def urlCount[P <: Platform[P]]
(tweets: Producer[P, Tweet],
urlExpander: P#Service[String, String],
store: P#Store[String, Long]) =
source.flatMap { tweet =>
  extractUrls(tweet.getText)
}.map { url => (url, 1L) }
  .leftJoin(urlExpander)
  .map {
    case (shortUrl, (count, optResolvedUrl)) =>
      (optResolvedUrl.getOrElse("unknown"), count)
  }
  .sumByKey(store)
Hybrid Online/Batch Processing

Example: count historical clicks and clicks in real time
Deployment Status

Multiple generation of systems for “real-time counting”
(lots of experience on use cases)

Began late 2012
First production usage early 2013
Open-sourced Sept 2013

Currently:
A few dozen jobs, account for ~half of online analytics
Powers dashboards, signals for products
Related work

Lots of work on dataflow languages:
Pig, Scaling, DryadLINQ, Spark, etc.

Lots of work on online MapReduce:
HOP, DEDUCE, MapUpdate, etc.

Lots of work on incremental batch processing:
CBP, Incoop, Hourglass, etc.

Lots of work on stream processing:
Aurora, S4, Samza, BlockMon, Spark Streaming, MillWheel, Photon, etc.

Lots of work on pub-sub:
Kafka, RabbitMQ, SQS, etc.

Some work on category theory and big data:
monad comprehensions, monoids for ML, CRDT
Future Work

More target execution frameworks, e.g., Spark

Optimizations:
Standard “bag of tricks”
Automatic tuning of mini-batches for Storm
Integrating batch and online MapReduce

Idea #1: Algebraic structures provide the basis for seamless integration of batch and online processing.

Idea #2: For many tasks, close enough is good enough. Probabilistic data structures as monoids.

Questions?