Summingbird

Integrating Batch and Online Computation

Boykin, Oscar, et al. "Summingbird: A framework for integrating batch and online mapreduce computations." *Proceedings of the VLDB Endowment* 7.13 (2014): 1441-1451.

Contents

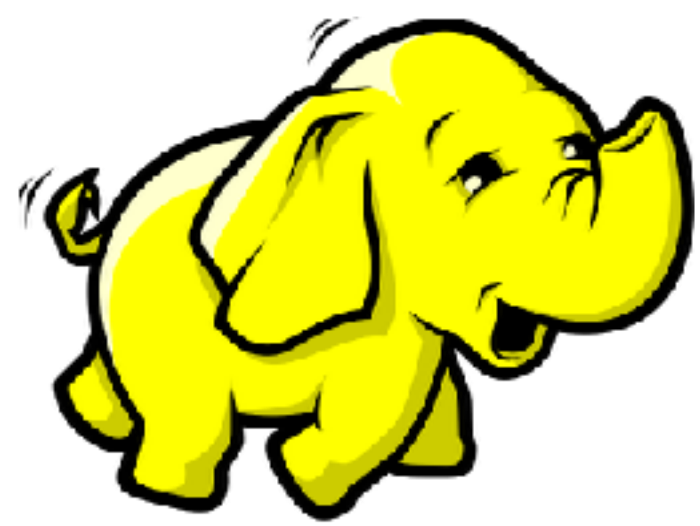
- Background
- Computation Model
- Algebraic Structures
- Hybrid Processing

Background

- Hadoop and batch analytics
- Online analytics
- Need for batch/online hybrids

Hadoop and batch analysis

- Mature production system
- Use high-level data flow language (Pig and Scalding)
- Exact answer are not necessary



Online analysis

Traditionally build code to "hook" the libraries

- Maintain codes for different framework
- Features in Hadoop may be too slow for online processing
- Data structures in disk cannot meet the latency requirements

STORM

Need for online/batch hybrids

- Online analytics could provide low latency with looser guarantees.
- Batch analytics could provide more accurate answers.
- Summingbird provides abstraction for analytical queries and handling for abnormal behaviour.

Computation Model

Basic Concepts:

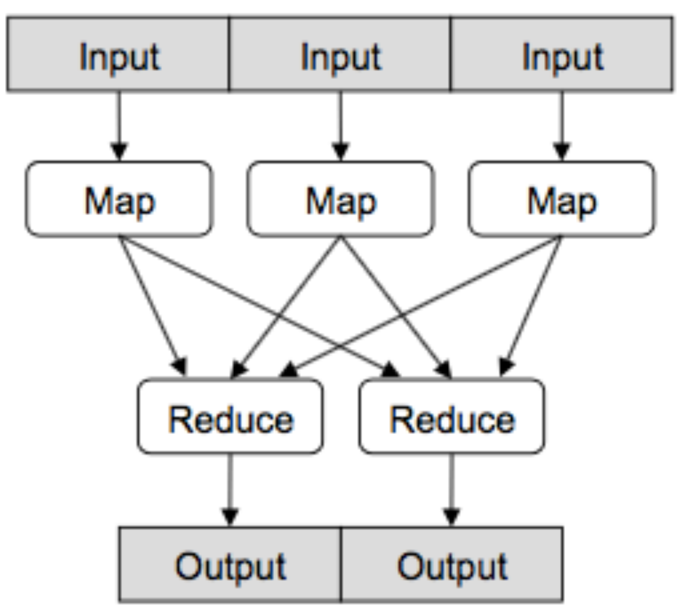
- Producer
- Source
- Platform
- Store
- Sink

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

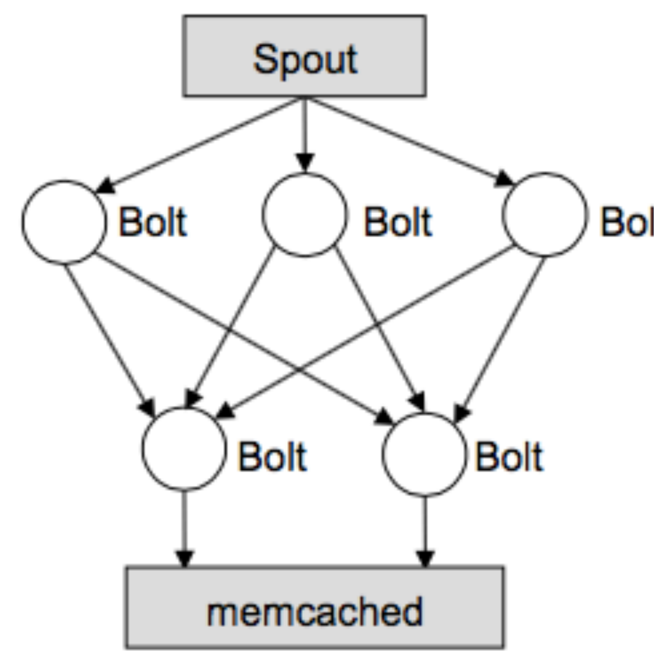
Sample code for word count

Computation Model

```
// Running in Hadoop (via Scalding/Cascading)
Scalding.run {
  wordCount[Scalding](
    Scalding.source[Tweet]("source_data"),
    Scalding.store[String, Long]("count_out")
  )
}
```




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



Computation Model

"Map" Computation:

- flatmap[T,U](fn: T => List[U]): List[U]
- map[T,U](fn: T => U): List[U]
- filter[T](fn: T => Boolean): List[T]

Computation Model

Algebraic structure:

- semigroup
- monoid
- commutative semigroup (or monoid)

Computation Model

Reasons for “commutativity”:

- Partitioned input requires commutativity
- Commutativity enables optimisation

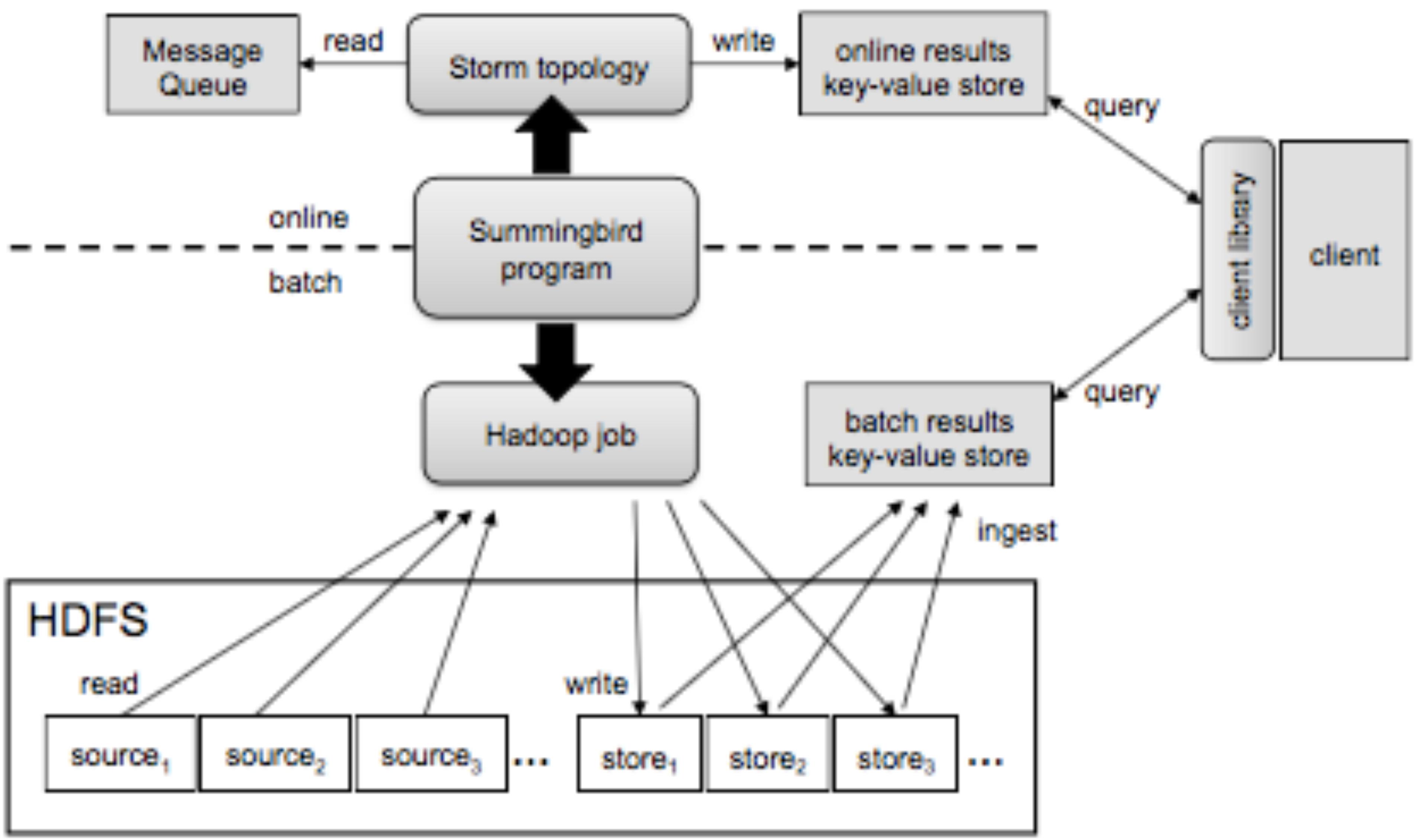
Algebraic Structure

- Addition, multiplication
- Set union
- max, min operator
- Element-wise application of commutative operator

Algebraic Structure

- Minhash
- Bloom Filters
- Hyperloglog counters
- Count-Min Sketches

Hybrid Processing



Hybrid Processing

- TimeExtractor, Batcher
- Hadoop job is triggered to compute aggregates on the next incremental source batch => (K, (batchId, V))
- Storm topology for online processing => ((K, batchId), V)
- Client would first get from batch results, then get online results to “fill in” the gap

Production Experience

- Typical Summingbird job process 1-20 MB/s
- Simple primitives (selection, joins) are sufficient
- Research opportunities for query optimisation

Limitation: generic folds

```
(1 2 3 4 5)         Here is my list.
1   + (2 3 4 5)     I take the  first-item-of-the-list  and  the-rest-of-the-list.
1+2 <-(_ 3 4 5)     I add  the-first-item-of-the-rest-of-the-list  to the initial first item.
3   + (3 4 5)       This leaves me with a new  rest-of-the-list.
6   + (4 5)         I do the same thing again.
10  + (5)           And again.
15  + ()            And again. Now I have no list left.
15                  Therefore, the sum is 15. (((1 +2) +3) +4) +5 = 15.
```