6.824 2018 Lecture 13: Naiad Case Study

Naiad: A Timely Dataflow System
Murray, McSherry, Isaacs, Isard, Barham, Abadi
SOSP 2013

Today: streaming and incremental computation
  Case study: Naiad
  Why are we reading Naiad?
    elegant design
    impressive performance
    open-source and still being developed

recap: Spark improved performance for iterative computations
  i.e., ones where the same data is used again and again
  but what if the input data changes?
    may happen, e.g., because crawler updates pages (new links)
    or because an application appends new record to a log file
  Spark (as in last lecture's paper) will actually have to start over!
    run all PageRank iterations again, even if just one link changed
    (but: Spark Streaming -- similar to Naiad, but atop Spark)

Incremental processing
  input vertices
    could be a file (as in Spark/MR)
    or a stream of events, e.g. web requests, tweets, sensor readings...
    inject (key, value, epoch) records into graph
  fixed data-flow
    records flow into vertices, which may emit zero or more records in response
  stateful vertices (long-lived)
    a bit like cached RDDs
    but mutable: state changes in response to new records arriving at the vertex
    ex: GroupBy with sum() stores one record for every group
      new record updates current aggregation value

Iteration with data-flow cycles
  contrast: Spark has no cycles (DAG), iterates by adding new RDDs
    no RDD can depend on its own child
  good for algorithms with loops
    ex: PageRank sends rank updates back to start of computation
  "root" streaming context
    special, as introduces inputs with epochs (specified by program)
  loop contexts
    can nest arbitrarily
    cannot partially overlap
  [Figure 3 example w/ nested loop]

Problem: ordering required!
  [example: delayed PR rank update, next iteration reads old rank]
  intuition: avoid time-travelling updates
    vertex sees update from past iteration when it already moved on
    vertex emits update from a future iteration it's not at yet
  key ingredient: hierarchical timestamps
    timestamp: (epoch, [c1, ..., ck]) where ci is the ith-level loop context
  ingress, egress, feedback vertices modify timestamps
    Ingress: append new loop counter at end (start into loop)
    Egress: drop loop counter from end (break out of loop)
    Feedback: increment loop counter at end (iterate)
  [Figure 3 timestamp propagation: lecture question]
    epoch 1:
    (a, 2): goes around the loop three times (value 4 -> 8 -> 16)
      A: (1, []), I: (1, [0]), ... F: (1, [1]), ... F: (1, [2]), ... E: (1, [])
    (b, 6): goes around the loop only once (value 6 -> 12)
      A: (1, []), I: (1, [0]), ... F: (1, [1]), ... E: (1, [])
    epoch 2:
    (a, 5): dropped at A due to DISTINCT.
  timestamps form a partial order
    (1, [5, 7]) < (1, [5, 8]) < (1, [6, 2]) < (2, [1, 1])
    but also: (1, [5, 7]) < (1, [5]) -- lexicographic ordering on loop counters
    and, for two top-level loops: (1, 1A [5]) doesn't compare to (1, 1B [2])
  allows vertices to order updates
    and to decide when it's okay to release them
    so downstream vertex gets a consistent outputs in order

Programmer API
  good news: end users mostly don't worry about the timestamps

    timely dataflow is low-level infrastructure
    allow special-purpose implementations by experts for specific uses
  [Figure 2: runtime, timely dataflow, differential dataflow, other libs]
  high-level APIs provided by libraries built on the timely dataflow abstraction
    e.g.: LINQ (SQL-like), BSP (Pregel-ish), Datalog
    similar trend towards special-purpose libraries in Spark ecosystem
      SparkSQL, GraphX, MLLib, Streaming, ..

Low-level vertex API
  need to explicitly deal with timestamps
  notifications
    "you'll no longer receive records with this timestamp or any earlier one"
    vertex may e.g., decide to compute final value and release
  two callbacks invoked by the runtime on vertex:
    OnRecv(edge, msg, timestamp) -- here are some messages
    OnNotify(timestamp)   -- no more messages <= timestamp coming
  two API calls available to vertex code:
    SendBy(edge, msg, timestamp) -- send a message at current or future time
    NotifyAt(timestamp)   -- call me back when at timestamp
  allows different strategies
    incrementally release records for a time, finish up with notification
    buffer all records for a time, then release on notification
  [code example: Distinct Count vertex]

Progress tracking
  protocol to figure out when to deliver notifications to vertices
  intuition
    ok to deliver notification once it's *impossible* for predecessors to
    generate records with earlier timestamp
  single-threaded example
    "pointstamps": just a timestamp + location (edge or vertex)
    [lattice diagram of vertices/edges on x-axis, times on y-axis]
    arrows indicate could-result-in
      follow partial order on timestamps
      ex: (1, [2]) at B in example C-R-I (1, [3]) but also (1, [])
      => not ok to finish (1, []) until everything has left the loop
    remove active pointstamp once no events for it are left
      i.e., occurrency count (OC) = 0
    frontier: pointstamp without incoming arrows
      keeps moving down the time axis, but speed differs for locations
  distributed version
    strawman 1: send all events to a single coordinator
      slow, as need to wait for this coordinator
      this is the "global frontier" -- coordinator has all information
    strawman 2: process events locally, then inform all other workers
      broadcast!
      workers now maintain "local frontier", which approximates global one
      workers can only be *behind*: progress update may still be in the network
        local frontier can never advance past global frontier
        hence safe, as will only deliver notifications late
      problem: sends enormous amounts of progress chatter!
        ~10 GB for WCC on 60 computers ("None", Figure 6(c))
    solution: aggregate events locally before broadcasting
      each event changes OC by +1 or -1 => can combine
      means we may wait a little longer for an update (always safe)
        only sends updates when set of active pointstamps changes
      global aggregator merges updates from different workers
        like global coordinator in strawman, but far fewer messages

Fault tolerance
  perhaps the weakest part of the paper
  option 1: write globally synchronous, coordinated checkpoints
    recovery loads last checkpoint (incl. timestamps, progress info)
    then starts processing from there, possibly repeating inputs
    induces pause times while making checkpoints (cf. tail latency, Fig 7c)
  option 2: log all messages to disk before sending
    no need to checkpoint
    recovery can resume from any point
    but high common-case overhead (i.e., pay price even when there's no failure)
  Q: why not use a recomputation strategy like Spark's?
  A: difficult to do with fine-grained updates. Up to what point do we recompute?
  cleverer scheme developed after this paper
    some vertices checkpoint, some log messages
    can still recover the joint graph
    see Falkirk Wheel paper -- https://arxiv.org/abs/1503.08877

Performance results

  optimized system from ground up, as illustrated by microbenchmarks (Fig 6)
  impressive PageRank performance (<1s per iteration on twitter_rv graph)
    very good in 2013, still pretty good now!
  Naiad matches or beats specialized systems in several different domains
    iterative batch: PR, SCC, WCC &c vs. DB, DryadLINQ -- >10x improvement
    graph processing: PR on twitter vs. PowerGraph -- ~10x improvement
    iterative ML: logistic regression vs. Vowpal Wabbit -- ~40% improvement

References:
  Differential Dataflow: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf
  Rust re-implementations:
    * https://github.com/frankmcsherry/timely-dataflow
    * https://github.com/frankmcsherry/differential-dataflow