Batch Processing

Natacha Crooks - CS 6453
Data (usually) doesn’t fit on a single machine

CoinGraph (900GB)

LiveJournal (1.1GB)

Orkut (1.4GB)

Twitter (between 5 and 20GB)

Netflix Recommendation (2.5BGB)

Sources: Musketer (Eurosys’15), Spark (NSDI’17), Weaver (VLDB’17), Scalability, but at what COST (HotOS’16)
Where it all began*: MapReduce (2004)

- Introduced by Google
- Stated goal: allow users to leverage power of parallelism/distribution while hiding all its complexity (failures, load-balancing, cluster management, ...)
- Very simple programming model:
  - \( \text{map} \ (k1,v1) \rightarrow \text{list}(k2,v2) \)
  - \( \text{reduce} \ (k2,\text{list}(v2)) \rightarrow \text{list}(v2) \)
- Simple fault-tolerance model
  - Simply reexecute...

* Stonebraker et al./database folks would disagree
PageRank in MapReduce (Hadoop)

Input:
adjacency matrix

HDFS

Map Phase
Write to local storage

Shuffle Phase

Reduce Phase

Iterate

Write to HDFS

PR(a) = 1 - l/N + l * sum(PR(y)/out(y))

(a, PR(a)/out(a))

(a, [c])
(b, [a])
(c, [a, b])

(c, PR(a) / out (a)), (a, [c])
(a, PR(b) / out (b)), (b, [a])
(a, PR(c) / out (c)), (c, [a, b])
(b, PR(c) / out (c))

(a, PR(a)/out(a))
Issues with MapReduce

- Difficult to express certain classes of computation:
  - Iterative computation (ex: PageRank)
  - Recursive computation (ex: Fibonacci sequence)
  - “Reduce” functions with multiple outputs

- Read and write to disk at every stage
  - Leads to inefficiency
  - No opportunity to reuse data
Arrive Dataflow! Dryad (2007)

- Developed (concurrently?) by Microsoft. Similar objective to MapReduce

- Introduce a more flexible dataflow graph. A job is a DAG where:
  - Nodes representing arbitrary sequential code
  - Edges representing communication graph (shared memory, files, TCP)

- Benefits
  - Acyclic -> easy fault tolerance
  - Nodes can have multiple inputs/outputs
  - Easier to implement SQL operations than in the map/reduce framework
Arrive Dataflow! Dryad (2007)

- Language to generate graphs from composition of simpler graphs

- Local job manager locally selects free nodes (job may have constraints) to run vertices
  - Both MapReduce and Dryad use greedy placement algorithms: simplicity first!

- Support for dynamic refinement of the graph
  - Optimize graph according to network topology
Arrive Recursion/Iteration! CIEL (2011)

- Dryad DAG is: 1) acyclic  2) static => limits expressiveness

- CIEL enables support for iterative/recursive computations by
  ○ Supporting data-dependent control-flow decisions
  ○ Spawning new edges (tasks) at runtime
  ○ Memoization of tasks via unique naming of objects

Lazily evaluate task: Start from the result future and attempt to execute tasks if dependencies are both concrete references. If future references, recursively attempt to evaluate tasks charged with generating these objects.
Arrive In-Memory Data Processing! Spark (2012)

- Claim: lack abstraction for leveraging distributed memory
  - No mechanism to process large amounts of in-memory data in parallel
  - Necessary for sub-second interactive queries as well as in-memory analytics

- Need abstraction to re-use in-memory data for iterative computation

- Must support generic programming language

- Propose new abstraction: Resilient distributed datasets
  - Efficient data reuse
  - Efficient fault tolerance
  - Easy programming
The magic ingredient: RDDs

- RDD: interface based on coarse-grained transformations (map, project, reduce, groupBy) that apply the same operation to many data items.

- Lineage: RDDs can be reconstructed “efficiently” by tracking sequence of operations and reexecuting them (few operations, but applied on large data).

- RDDs can be
  - actions (computed immediately) / transformations (lazy applied)
  - Persistent / In-memory with or without custom sharding
PageRank - Take 2 : Spark

// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()

var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
    // Build an RDD of (targetURL, float) pairs
    // with the contributions sent by each page
    val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
        .mapValues(sum => a/N + (1-a)*sum)
}

Figure 3: Lineage graph for datasets in PageRank.
Spark Architecture

- RDD implementation:
  - Set of partitions (atomic pieces of the dataset)
  - Set of dependencies (function for computing dataset based on parents)
    - Dependencies can be narrow (each partition of the parent RDD is used by at most one partition of the child RDD)
    - Dependencies can be wide (multiple partitions may be used)
  - Metadata about partitioning + data placement
Spark Architecture

- When user executes action on RDD, scheduler examines RDD’s lineage to build a DAG of stages to execute
  - Stage consists of as many pipelined transformations with narrow dependencies as possible.
  - Stage boundary defined by shuffle (for wide dependencies)
  - Task to where RDD resides in memory (or preferred location)
Evaluation - Iterative Workloads

Benefits of keeping data in-memory
(K-Means is more compute intensive)

Would have been nice to include comparison to Hadoop when memory is scarce
Are RDD really the magic ingredient?

- The ability to “name” transformations (entire datasets) rather than individual objects is pretty cool.

- But is it the “key” to Spark’s performance?
  - What if you just ran CIEL in memory?
  - Also has memoization techniques for data re-use

- I don’t fully understand what they bring for fault-tolerance
  - Doesn’t the CIEL re-execution model from the output node do exactly the same?
  - In CIEL also you only reexecute “part” of the output that has been lost (as that’s the granularity of objects.)
Where RDDs fall short

- Act as a caching mechanism where intermediate state can be saved, and where can pipeline data from one transformation to the next efficiently.

- What about reusing computation and enabling support for fine-grain access?
  - Ex: what if the page rank doesn’t change in one round. In Spark, still have to compute on the whole data (or filter it). Top-K doesn’t require recomputing everything when new data arrives.

- RDDs by nature do not support *incremental computation*
  - Maintain a view updated by deltas. Run computation periodically with small changes in the input.
Arrive Naiad (2013)

- Bulk computation is so 2012. Now is the time for *timely data flow*

- Need for a universal system that can do
  - Iterative processing on real-time data stream
  - Interactive queries on a consistent view of results

- Argue that currently
  - Streaming systems cannot deal with iteration
  - Batch systems iterate synchronously, so have high latency. Cannot send data increments
The black magic: Timely Dataflow

- Timely dataflow properties
  - Structured loops allowing feedback in the dataflow
  - Stateful dataflow vertices capable of consuming/producing data without coordination
  - Notifications for vertices once a “consistent point” has been reached (ex: end of iteration)

- Dataflow graphs are directed and can be cyclic

- Stateful vertices asynchronously receive messages + notifications of global progress

- Progress is measured through “timestamps”
Timestamps in Naiad (Construction)

- Timestamps are key to nodes “tuning” the degree of asynchrony/consistency desired in the system within different epochs/iterations.

- Dataflow graphs have specific structure (ingress/egress nodes, loop contexts).

- Encode path they have taken in DAG.

Timestamp: \((e \in \mathbb{N}, (c_1, \ldots, c_k) \in \mathbb{N}^k)\)

- **Vertex**
  - **Ingress** \((e, (c_1, \ldots, c_k))\)
  - **Egress** \((e, (c_1, \ldots, c_k, c_{k+1}))\)
  - **Feedback** \((e, (c_1, \ldots, c_k))\)

- **Input timestamp**: \((e, (c_1, \ldots, c_k))\)
- **Output timestamp**: \((e, (c_1, \ldots, c_k, 0))\) (Ingress), \((e, (c_1, \ldots, c_k))\) (Feedback), \((e, (c_1, \ldots, c_{k+1}))\) (Egress)
Timestaps in Naiad (Use)

- Timestamps are used to track forward progress of the computation
  - Helps a vertex determine when it wants to synchronise with other vertices
  - Vertex can receive timestamps from different epochs/iterations (no longer synchronous)
  - $T_1$ could result in $T_2$ if path from $T_1$ to $T_2$

- Every node implements methods `OnRecv/SentBy`, and `OnNotify/NotifyAt`
  - Notify only sent when will never send a smaller timestamp to that node

- Every node must reason about the possibility of receiving "future messages"
  - Set of possible timestamps constrained by set of unprocessed events + graph structure
  - Used to determine when safe to deliver notification
Timestamps in Naiad (Use)

- How do you compute the frontier?

  Pointstamp: \((t \in \text{Timestamp}, l \in \text{Edge} \cup \text{Vertex})\).

- Pointstamps have occurrence count + precursor count
  - Occurrence count: number of concurrently unprocessed events for that pointstamp
  - Precursor Count: number of unprocessed events that could result-in that pointstamp

- When pointstamp \(p\) becomes active:
  - Increment occurrence count + initialise precursor count to number of pointstamps that could result-in \(p\) + increment precursor count of pointstamps that \(p\) could result-in
  - When remove pointstamp (occurrence count = 0), decrement precursor count for pointstamps that \(p\) could result in
  - If precursor count = 0, then \(p\) is on the frontier
Timely Dataflow example

class DistinctCount<S, T> : Vertex<T>
{
    Dictionary<T, Dictionary<S, int>> counts;
    void OnRecv(Edge e, S msg, T time)
    {
        if (!counts.ContainsKey(time)) {
            counts[time] = new Dictionary<S, int>();
            this.NotifyAt(time);
        }

        if (!counts[time].ContainsKey(msg)) {
            counts[time][msg] = 0;
            this.SendBy(output1, msg, time);
        }

        counts[time][msg]++;
    }

    void OnNotify(T time)
    {
        foreach (var pair in counts[time])
        {
            this.SendBy(output2, pair, time);
            counts.Remove(time);
        }
    }
}

- Timely dataflow is hard to write. (McSherry’s implementation has 700 lines)
- Introduced two new high-level front-ends that leverage timely dataflow
  - GraphLINQ
  - Lindi
edges = edges.PartitionBy(x => x.source);
// capture degrees before trimming leaves.
var degrees = edges.Select(x => x.source).CountNodes();
var trim = false;
if (trim)
edges = edges.Select(x => x.target.WithValue(x.source)).FilterBy(degrees.Select(x => x.node))
.Select(x => new Edge(x.value, x.node));

// initial distribution of ranks.
var start = degrees.Select(x => x.node.WithValue(0.15f))
.PartitionBy(x => x.node.index);

// define an iterative pagerank computation, add initial values, aggregate up the results
var iterations = 10;
var ranks = start.IterateAndAccumulate((lc, deltas) => deltas.PageRankStep(lc.EnterLoop(degrees),
lc.EnterLoop(edges)), x => x.node.index, iterations, "PageRank").Concat(start)
// add initial ranks in for correctness.
.NodeAggregate((x, y) => x + y) // accumulate up the ranks.
.Where(x => x.value > 0.0f); // report only positive ranks.

// start computation, and block until completion.
computation.Activate();
computation.Join();
Results

- Claim: perform as well as different specialised systems

(a) PageRank on Twitter follower graph (§6.1)  
(b) Logistic regression speedup (§6.2)
Taking a step back: are universal frameworks the way to go?

- Spark is the de-facto default in industry
  - GraphX more popular than GraphChi or PowerGraph despite better performance

- Naiad was also becoming very popular.

- I’d argue that research is moving to an “in-between”
Taking a step back: are universal frameworks the way to go? (YES)

- Data is becoming more complex
  - Workflows don’t fit neatly into graph/ML/batch, but combination of all

- Developers like simplicity
  - One system to configure and manage
  - If Spark hadn’t been written in Scala, would it have succeeded?

- Systems like Spark or Naiad have “good enough” performance in all cases compared specialized systems
Taking a step back: are universal frameworks the way to go? (NO)

- Systems make fundamental (and incompatible) tradeoffs
  - Size of input
  - Structure of data (skew, selectivity)
  - Engineering decision (cost of loading input/preprocessing)

![Graphs showing makespan comparison](image)

Figure 3: Varying makespan for PageRank on social network graphs; lower is better; error bars: ±σ of 10 runs.

(a) PROJECT on a relation. (b) JOIN two data sets.
Taking a step back: are universal frameworks the way to go? (Sort of?)

- Evidence suggests that it is possible to capture “the core structure” of what all these workloads look like. And use this intermediate representation to convert part of workloads to best framework.

(Musketeer - Eurosyst’15)  
Current Spark ecosystem  
(Weld - CIDR’17)  
(NAIAD - SOSP’13)
Taking a (more radical) step back: are distributed data processing frameworks the way to go?

- Is data really that big? What are the overheads associated with going distributed unnecessarily?
  - 80% of Cloudera customers + 80% of jobs in Facebook have < 1GB input (VLDB’12)

- What is the COST of big data systems (Configurations that outperform a single thread (McSherry, HotOS’15)
  - Parallelism doesn’t necessarily mean efficiency

Figure 3: Varying makespan for PageRank on social network graphs; lower is better; error bars: ±σ of 10 runs.
Backups
Arrive Recursion/Iteration! CIEL (2011)

- Dryad DAG is: 1) acyclic  2) static => limits expressiveness

- CIEL enables support for iterative/recursive computations by
  - Supporting data-dependent control-flow decisions
  - Spawning new edges (tasks) at runtime
  - Memoization of tasks via unique naming of objects

Lazily evaluate task:
Start from the result future and attempt to execute tasks if dependencies are both concrete references. If future references, recursively attempt to evaluate tasks charged with generating these objects.
Evaluation - Iterative Workloads

- Unclear what the takeaway are: different in result seem to be due to Hadoop engineering decisions
- Performance improvement stems from better locality of tasks in CIEL (schedule tasks with warm caches/next to data)
- No evaluation of memoization? Would have liked to see how results change with number of iterations

Figure 8: Results of the k-means experiment on Hadoop and CIEL with 20 workers (§6.2).