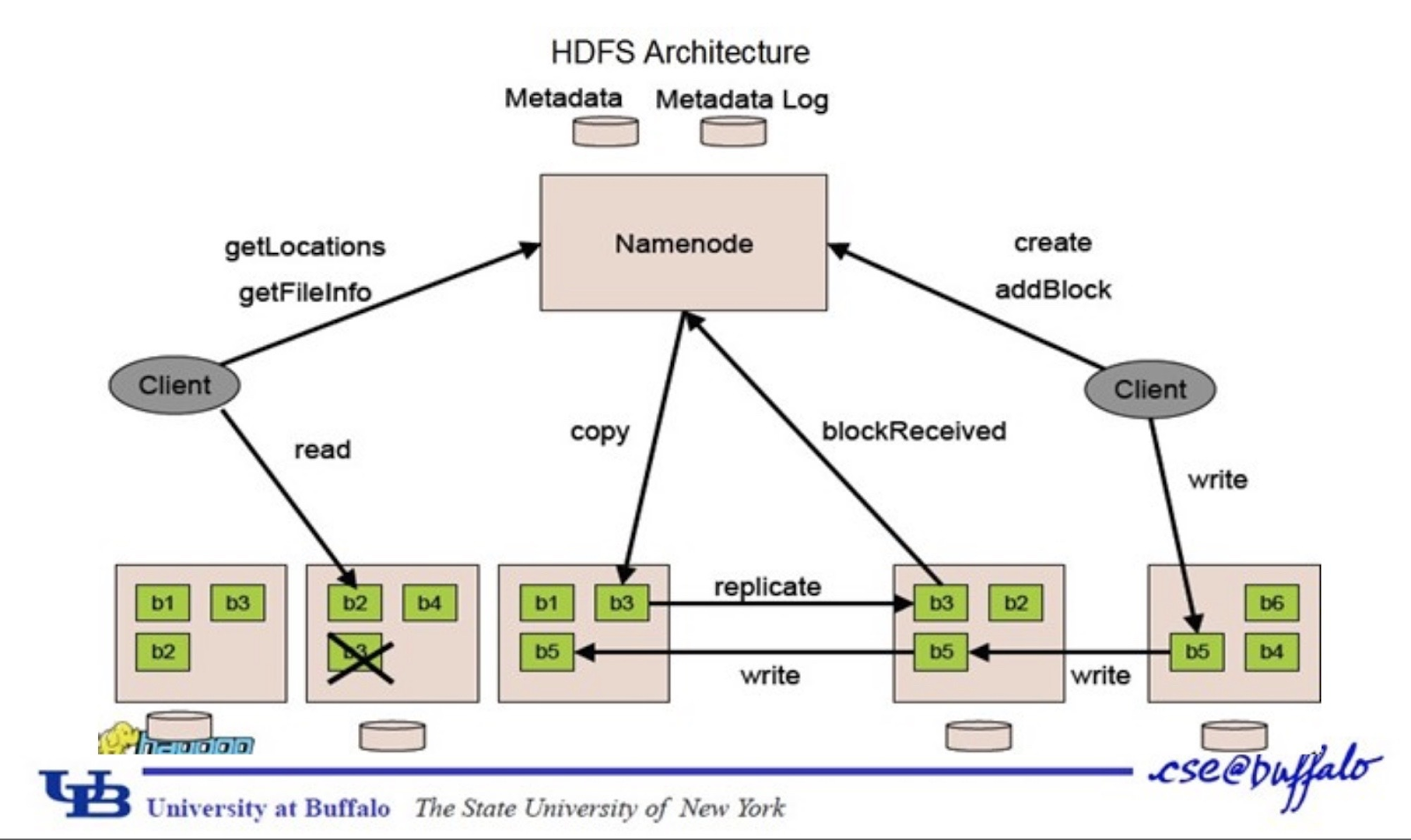
**Part1. Summary of All**

**1.** **Hadoop Distributed File System**



**Goals of HDFS:** (1) Store large datasets reliability (2) Stream those datasets at high bandwidth to user applications (high throughput rather than low latency) (3) Run on commodity hardware (low-cost) (4) Write-once-read-many access model to files

**Benefits:** (1) Scalable - easily grown by adding more nodes (2) Fault Tolerance (3) Fault Tolerance.

**Architecture**: (1) Files broken into blocks of 128 MB (default) (2) File can be made up of several blocks (3) Each block replicated across machines to prevent loss of data (4) Metadata stored on NameNode and application data on DataNodes.

**NameNode:** (1) Metadata stored on dedicated server called the NameNode (2) Maintains namespace tree and mapping of file blocks to DataNodes. (3) “Image” - metadata of the name system. Contains the inodes data and list of blocks belonging to each file (4) “Checkpoint” - persistent record of image stored in host’s native file system. (5) “Journal” - modification log of image stored on NameNode.

**DataNode:** (1) Stores block data (and some metadata like checksums and generation stamp) (2) On startup (a) Connects to NameNode and performs handshake -- verifies namespace ID and software version. (b) Registers with NameNode with its unique storage ID. (3) Block reports to NameNode. (4) Heartbeats to NameNode to confirm is operating. NameNode can reply with various instructions.

**Client:** (1) Reading (a) Client contacts NameNode for locations of data blocks comprising the file (b) Client reads block contents from the DataNode. (2) Writing (a) For each block, client requests NameNode to nominate 3 DataNodes to host the block replicas. (b) Writes to DataNodes in a pipeline fashion.

**2.MapReduce**

It is a programming model used in web search, inverted indexes etc, it wants automatic parallelization, network, disk optimization, it can handle machine failures.

MapReduce: Mapper((Kin,Vin) -> list(Kinter,Vinter)) + Reduce ((Kinter, list(Vinter)) -> list(Kout, Vout))

**Below is classic example of word count.**

1. MapReduce library splits the input int M pieces and starts up many copies of the program

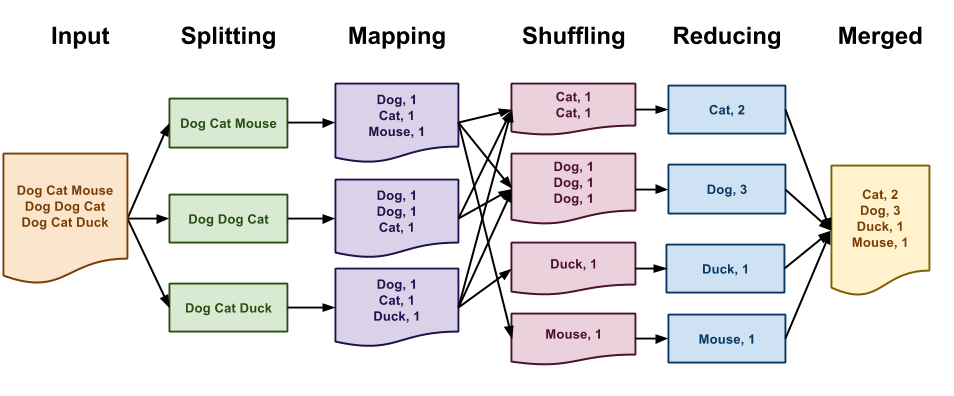
2. One is the master. It assigns map and reduce tasks

3. Map worker reads corresponding input, runs map function, buffers key/value pairs

4. Periodically buffered pairs are written to local disk, partitioned into R regions. These locations are sent back to master who forwards these to reducers

5-6. Reducer uses RPCs to read data. Once it has all of it, it sorts it, then runs the reduce function, and the output is appended to its output file

7. When all completed, wakes up user program.



For MapReduce,

it can fault recovery, (1) Crashed task is retried on another node (2) Master failure aborts the program.

its locality attempts to schedule tasks near corresponding data.

Its combiners - reduce locally before sending to reducer to lower network traffic.

Its stragglers - launch backup tasks when job close to completion.

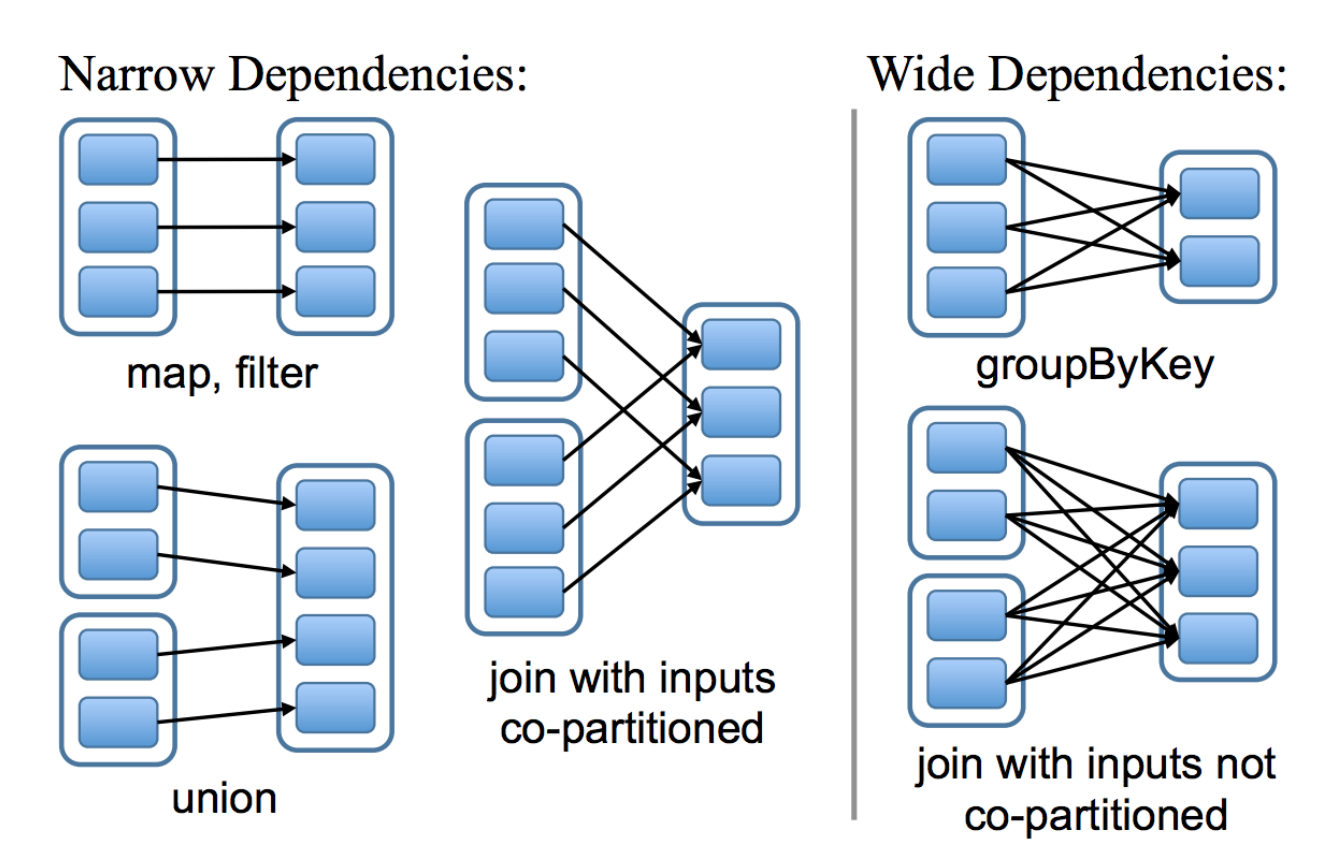
The shortcomings of Map Reduce:

1. Most applications require multiple MR steps.
2. MR only provides one pass of computation A. Data must be written to file system in between B. Expensive for data reuse.

**3.Spark**

**Resilient Distributed Datasets** (1) Read-only, partitioned collections of records (2) Provide restricted programming model by allowing coarse-grained transformations (maps, filters, joins) (3) This abstraction allows for efficient data reuse but general enough to describe different parallel applications.

**RDD Advantages** (1) Fault tolerance with lineage graph instead of replication or checkpointing. (2) Not materialized until needed. (3) Intermediate RDDs can be stored in memory instead of written to disk (great for iterative computations).



Narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD. Allows for pipelining.

Wide dependencies: where multiple child partitions may depend on it.

**Apache Hadoop YARN**



(Yarn) Separates resource management from the programming model, requirements including (Scalable, Locality aware, programming model diversity), resource manager, application master and node manager.

**Resource Manager**

1.per-cluster 2. Runs on dedicated machine 3. Acts as central authority arbitrating resources. 4. Exchanges heartbeats with node managers to keep consistent global view of cluster. 5. Doesn’t provide status for running applications or serving framework-specific reports -- just a scheduler.

**Node Manager**

1.Per-machine agent 2. Managers life-cycle of container 3. Monitors health of node

**Application Master**

1. Coordinates the application’s execution

2. Encodes preferences in heartbeat to RM and receives lease in subsequent heartbeats.

**DRF: Dominant Resource Fairness**

Example Problem: Consider a system with of 10 CPUs, 20 GB RAM, and two users, where user A runs tasks with demand vector <3 CPU, 2 GB>, and user B runs tasks with demand vector <1 CPU, 5 GB> each. How many tasks will be allocated by DRF to each user?

Key Properties: (1) Sharing Incentive: Sharing is better than user having 1/n of all resources

(2) Strategy-proofness: User cannot get better allocation by lying about its demands

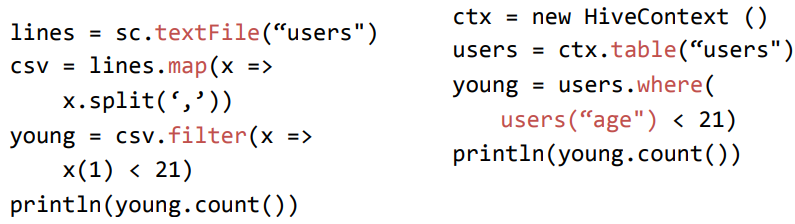
(3) Envy-freeness: User does not prefer some other users’ allocation

(4) Pareto Efficiency: Maximal resource utilization subject to satisfying above properties

**Spark-SQL**

(1) Main Abstraction: Data frame = RDD + Schema

(2) Data frame enables relational operations - projection (select), filter (where), join, and aggregations (groupBy) - that take in expressions in a limited DSL

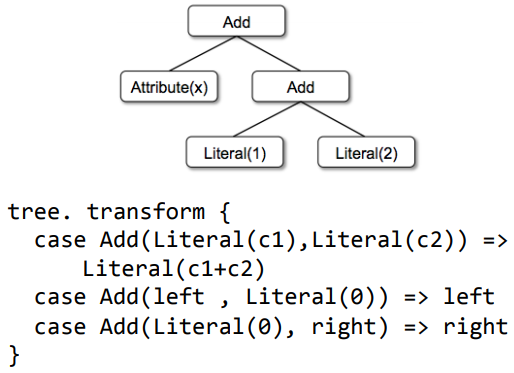


(3) Build an Abstract Syntax Tree (AST)

**SparkSQL - Catalyst Optimizer**

1. Logical Plan Optimization: Take in trees (AST) and applies rules to simplify

2. Rules = Pattern Match on subtree and replace by simplified subtree



3.Physical Plan Optimization: Generate multiple plans and use plan with the least cost

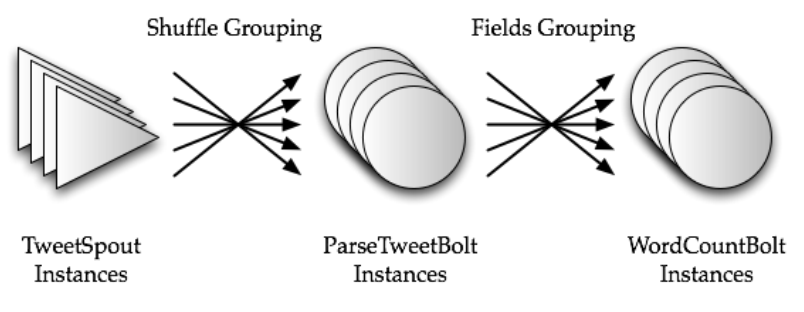
Clarinet - Geo-Distributed Analytics

1.Motivation: Cost of a physical plan is dependent on WAN-link bandwidths and how inter-datacenter over-the-WAN data transfers are scheduled

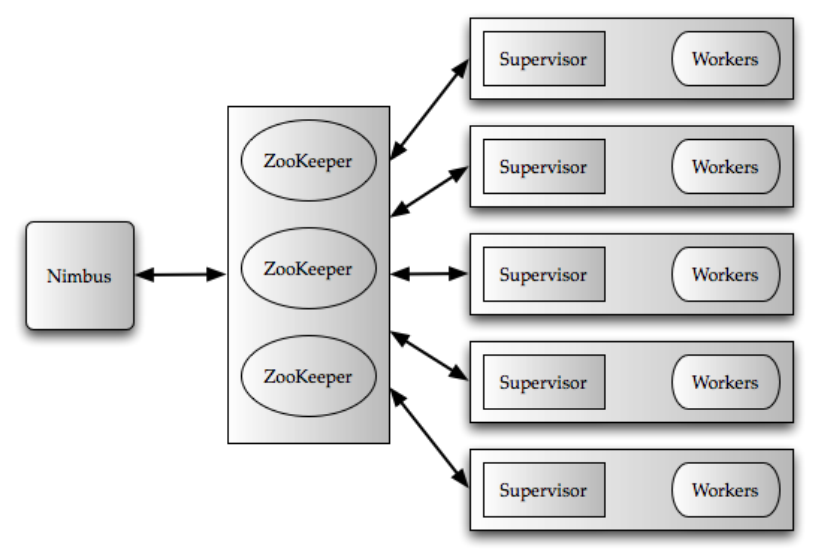
2.For a Single Query – a. Cost = Estimated running time with optimal data transfer schedule given current WAN-link bandwidths (lower is better)

3.For Multiple Queries – a. Co-optimize physical plan selection for all queries (evaluate exponential number of choices) b. Cost has to be contention-aware c. Use shortest job first (SJF) heuristic - Pick shortest physical plan from across all queries, reserve resources, and repeat.

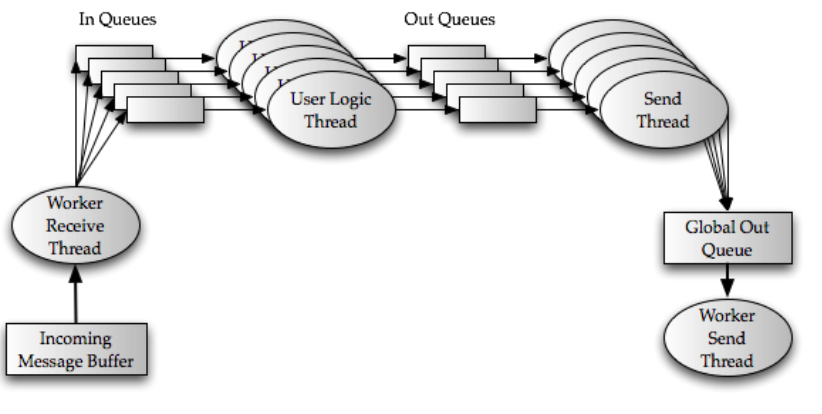
**Storm - Programming Model**



**Storm - Architecture Overview**



**Storm - Message Flow @ Worker**



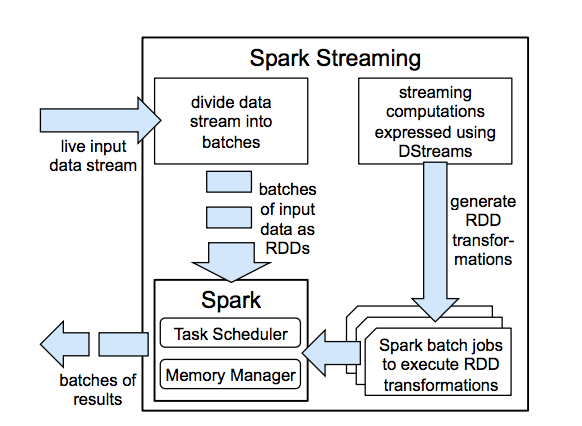
**Spark Streaming**

**Goals** 1. Scale to hundreds of nodes2.Second scale processing latencies 3**.** Second scale recovery from faults as well as stragglers.

**Prior systems based on continuous operator model**

1. Makes it hard to deal with faults as well as stragglers in a timely fashion. 2. Approaches a. Replication b. Upstream Backup

**Discretized Streams**



**Fault Tolerance**

1.Extend RDDs lineage across time and leverage RDD’s recovery mechanisms in the presence of failures

2.Recovery of the failed partitions can be done in parallel -- leading to fast recovery

**Processing Semantics**

Exactly once semantics, Computation is deterministic, stateless and ordered by time

**Machine learning:**

It includes (1) Data parallelism (2) Parameter server (3) Model parallelism (4) Consistency (5) Consistency (6) Inference.

**Main challenges, ideas:**

How to scale training to large data sets (n) and large models (w)? Fast training?

Ideas: 1. Data parallelism 2. Model parallelism 3. Consistency models 4. Systems tricks to speed things up.

**Data parallelism**

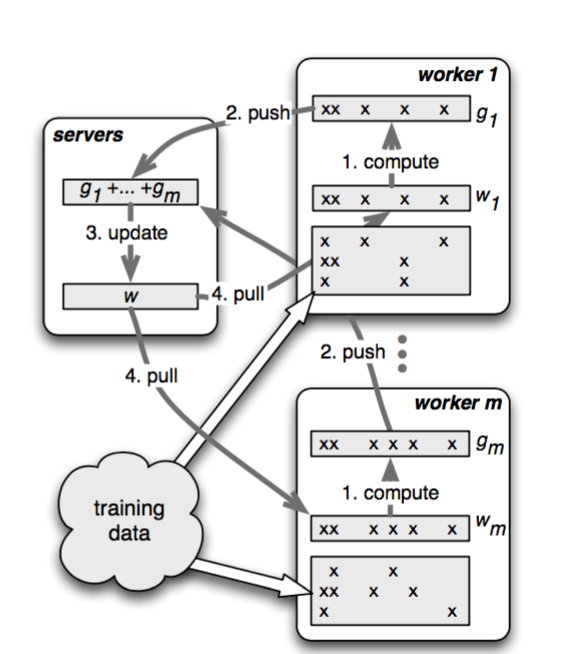
Distributes mini-batch gradient computation across workers → data parallelism helps speed up stochastic gradient descent based training. Updates to the model can happen in one of two different ways → structuring the system appropriately can help make this efficient.

(1) Peer-to-peer, or some aggregation topology a. Each worker stores model; receives gradient updates from everyone else; b. When gradient updates are synchronized (received from everyone), model is updated c. Coordination is hard -- complex local worker logic d. Somewhat difficult to control communication overhead

(2) Parameter server

a. Separates out aggregating and synchronizing model params and updates.

**Parameter server**



Positive side-effects

1. Fault tolerance -- nothing ML-specific; just becomes easier
2. Can implement different consistency models → better compute/comm overlapping
3. Elasticity and fault tolerance semantics are simpler

**Consistency**

**Synchronous** -- wait for all workers updates. Update at server. Then workers pull update and move to next iteration.

(1) Converges, but wastes resources and stragglers can hurt → how to avoid/overcome stragglers.

(2) workers idle in between iterations.

**Asynchronous** -- updates from a worker applied immediately. Workers can pull right after.

(1) Makes more efficient use of resources.

(2) But workers can operate on models of unbounded staleness.

(3) May not converge.

**Stale synchronous** -- in between the two extremes

**Asynchronous vs synchronous**

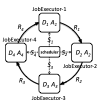
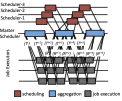
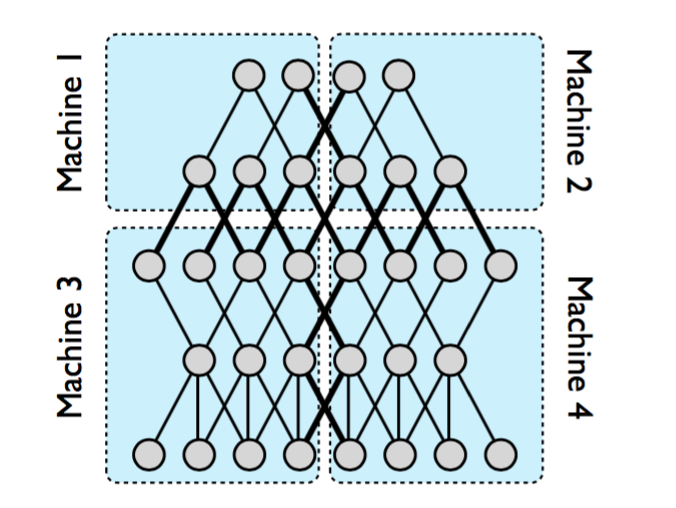
It is not a systems problem! Asynchronous and synchronous are different algorithms.

(1) Synchronous execution is clear → updates from all workers are gathered before applying to the model. Convergence is known.

(2) Asynchronous execution → making updates and using updates are decoupled.

(3) Worker makes update → guaranteed to see its updates; some recent updates of other workers updates may be seen. Known to diverge.

**Model Parallelism (DistBelief, STRADS)**

Two main observations:

(1) parameters may be dependent, thus naive concurrent updates can introduce errors that slow convergence;

(2) model parameters converge at different rates; hence a small subset of parameters can bottleneck ML algorithm completion.

**Programming model (TensorFlow)**

1.TensorFlow, e.g. → largely tied to algorithm specifics, as opposed to system-level requirements. Contrast with Spark’s use of RDDs.

(1) RDDs were about efficient fault tolerance and in-memory computation.

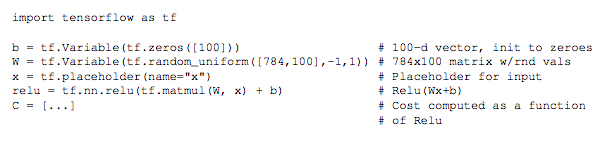
(2) It is application agnostic.

(3) Allows multiple different applications to be written atop and benefit from RDDs.

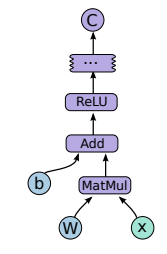
2. ML programming models push the level of abstraction closer to apps. Lose generality but better meet application-level programmer goals.

**Programming Model**

Directed Dataflow Graph to express computation.



Introduction about the programming model;

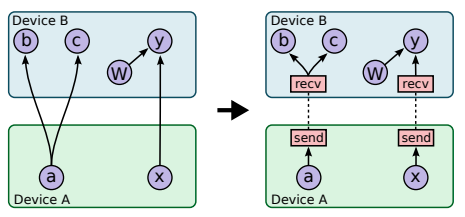


1. Sessions to interact with the TensorFlow System
2. s = tf.session() | s.run(C, feed\_dict={x: input})
3. TensorFlow takes care of executing the graph on the underlying machines
4. Operators - Kernels
5. Directed Computation Graph – Symbolic.
6. Variables - Nodes can be stateful
7. Each node is an operation
8. TensorFlow along edges

**Execution of programming model.**

1. Placement of different modes on to underlying devices / machines: Greedy.

2. Example



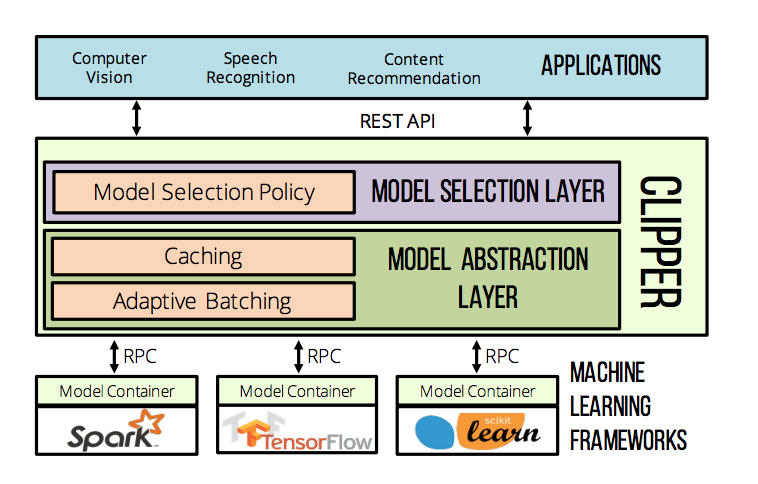
**Scheduling (****Gandiva)**

1.Leverages predictable iterative nature of ML. Ensures affinity as much as possible.

2.Mechanisms: Migration, suspend-resume, elasticity, profiling (used to trigger different mechanisms)

3.Introspection: Migration, time slicing, and elasticity triggered in the background based on constant profiling.

**Training vs Inference**

****

**Graph Analytics Summary.**

1.High level definitions.

2.Pregel

3.GraphLab

4.GraphX

**Graph Processing**

1.How to compute: iterative, vertex-centric processing. “Think like a vertex”

2.Vertex runs a program, with local state, or “value”

3.Each iteration, vertex gets input from other vertices about some function of their state. It accumulates this (“gather”), modifies its state (“apply”), and prepares sending this to other vertices (“scatter”)

4.Two sets of ways of doing this (1) message passing, distributed shared memory; (2) synchronous (all iterations proceed in lock step); asynchronous (each vertex can process whatever partial messages it receives or data it reads)

**Pregel**

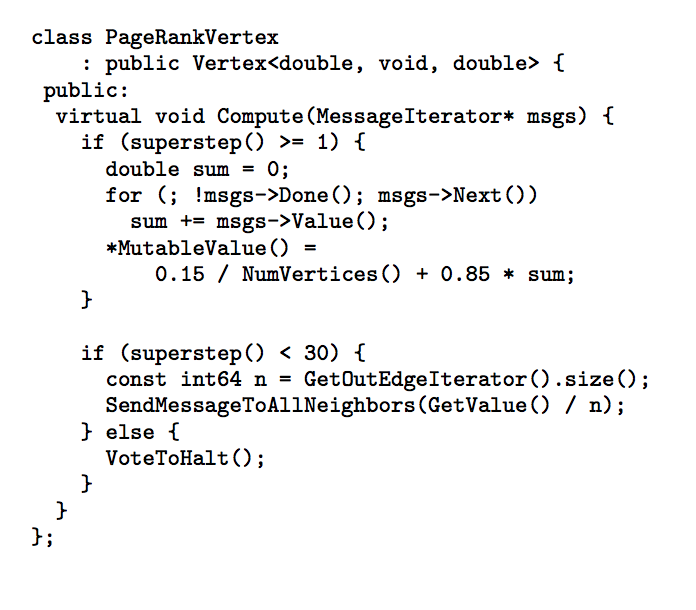
1.Synchronous message passing

2.Vertexes start iteration by gathering; then they apply, and then they scatter.

3.Messages carry values along edge (E.g., page rank/degree)

4.Scattered messages are buffered at recipient vertices.

5.Iterations coordinated by a Master.



**GraphLab**

1.Uses distributed shared memory.

2.In gather phase, a vertex reads values from neighboring vertices

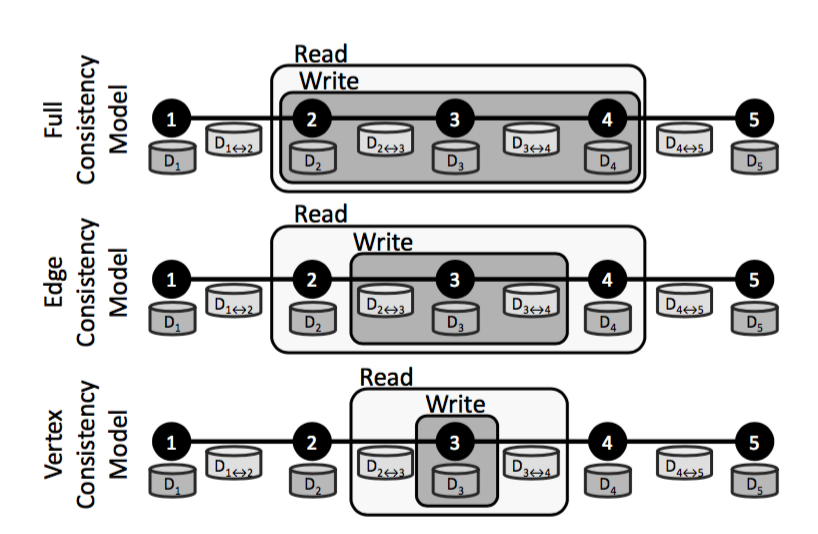
3.Systems ideas for performance: delta caching, conflict free vertex activations

4.Vertex has access to neighbor data any time; in Pregel need messages explicitly

5.Good for dynamic algorithms (page rank, where converged nodes can “stop executing”)

Also easily supports parallelism

**GraphLab consistency models**



1.Full consistency: “scopes” of concurrently executing updates don’t overlap

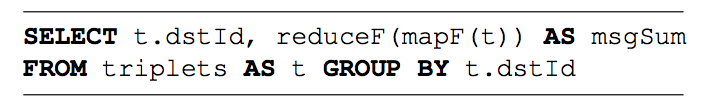
(1) “Scope” → neighborhood of a vertex

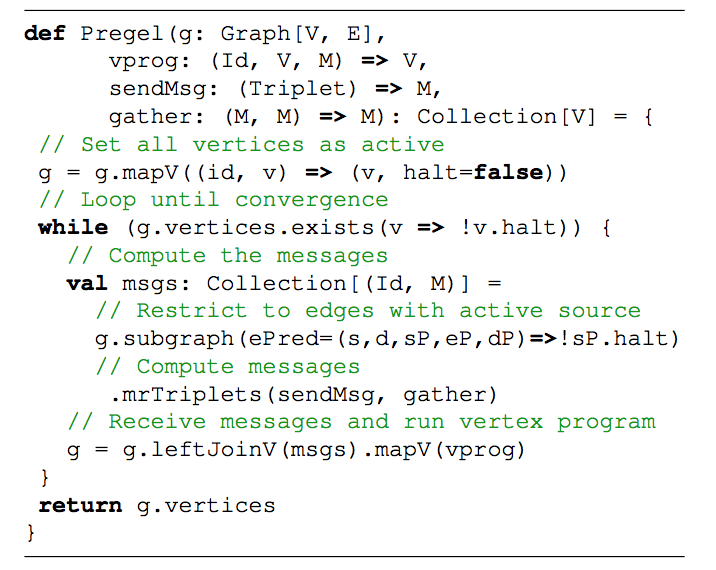
(2) Full → full read/write access to all data within a scope

2.Edge consistency: a vertex has read/write access to itselfs and edges incident and read-only to adjacent vertices

3.Vertex consistency: all vertexes can run in parallel, enabling full parallelism

4.Updates to shared state needs locking. GraphLab has a graph-centric locking approach where vertexes take turns

**GraphX**

****

Data organization -- vertex and edge collections (tables)

1.Vertex cut partitioning for the edge collection -- introduced first by GraphLab

2.Graph processing realized using relational operators

3.Create a triplet’s view

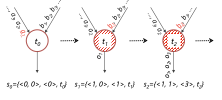
4.Run triplet.mrtriplets(map, reduce); For example, t.mrtriplets(sendmsg,gather)

5.Thinking of data as collections allows use of relational operators, e.g., use of indexes.

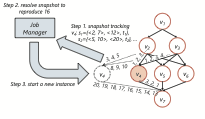
**Execution**

Similar to programming model, execution advancements are also tied to application specifics. E.g., leveraging special hardware to efficiently run ML kernels, optimize communication, etc.

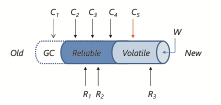
**Stream Scope**



Numbered event streams



Snapshots, recovery



Asynchronous checkpointing

**Serverless Computing**

New model of cloud computing

Programmers write functions. Infrastructure takes care of launching, and rapidly scaling up/down.

Billing per event. When an event arrives, routed to an existing lambda (if queue is small) or a new lambda is launched.

Cold start latency can be large, but rapid scale out possible.

Stateless, all I/O happens via an external store. Data is always persisted

Interesting programming model, but poor fit for “stateful” apps, e.g., batch analytics, ML, ..?

**Approximation, an area of growing interest**

A = Q(I)

Multiple approaches: approximate input, by sampling (blinkdb), or approximation computation (quickr, sketches, freeze inference)

BlinkDB key ideas:

1.Stratified sampling. Over-sample rows from rare QCSs.

2.Maintain samples for multiple QCSs that have good QCS coverage, occur often, good storage cost, good coverage of sparse QCSs

3.Build ELPs for a query, and pick a point on the ELP curve that is optimal w.r.t time or accuracy

Many issues: QCS overlap may be small; sample creation is expensive; sample may be as large as original table

**Related to Industry:**

You now know:

**Hadoop,**

**MR** (Computation expressed as “maps”, “reduces”),

**Spark** (RDDs: immutable partitioned collection of records + Computations expressed as transformations or actions on RDDs),

**Tez**(Computations expressed as a DAG. Each vertex takes generic compute logic),

batch processing, Streaming, Graph analytics, ML, TensorFlow, serverless, …

You are unique -- few people have this breadth of knowledge!

Put it on your resume!

Many companies WANT YOU: Databricks, Mesosphere, Snappydata, Hortonworks, Cloudera, Google, Azure, Microsoft, Facebook, AWS, IBM, AMFAM, Uber, ATT, Comcast, … Sell yourself well.

**Midterm Summary:**

**1.HDFS and MapReduce**

A. (1) What is the role of the NameNode in HDFS? (2) Block locations are not part of the HDFS namenode checkpoint. Given this, how is the location map recovered upon NameNode failure?

(1) Metadata, namespace tree, mapping of file blocks to DataNodes

(2) Gets block reports from DataNodes

B. Describe what functionalities would the Map and Reduce tasks perform for (1) a job that takes as input list of (book name, author name) k-v pairs and returns the pairs sorted by the author name. (2) Consider an input list of documents as (unique document id, list of words in document). Write a map-reduce program (pseudo-code) that counts the occurrence of each unique word across all documents. The expected output is (word, total number of occurrences summed across all documents).

(1) Map to (author\_name, book\_name), reducer sorts

(2) Mapper: emit (word,1) for each word in the list. Reducer: reduce to (word,sum)

C. Briefly describe how does the MapReduce framework handle the following failure scenarios (Be sure to mention your assumptions, if you make any) -

(1) Completed Map task failure

(2) In-progress Map task failure

(3) In-progress Reduce task failure

(4) Completed Reduce task failure

(1) Re-executed because output is stored on local disk of failed machine

(2) Reset to idle and eligible to reschedule

(3) Reset to idle and eligible to reschedule

(4) Don’t need to be re-executed since output is stored in global file system

**2.Spark**

Iterative algorithms were one of the motivations behind Spark. For example, an iterative PageRank algorithm can be implemented atop Spark using the following code snippet -

val links = spark.textFile(...).map(...).persist()

var ranks = // RDD of (URL, rank) pairs

for (i <- 1 to 5)

{

val contribs = links.join(ranks).flatMap {

(url, (links, rank)) => links.map(dest => (dest, rank/links.size)) }

ranks = contribs.reduceByKey((x,y) => x+y).mapValues(sum => a/N + (1-a)\*sum)

}

A. Spark introduces the notion of an RDD. For the application above, how many RDDs are generated? Why? Give any one benefit of RDDs.

Sol: 12 RDDs (1 links + 6 ranks + 5 contribs). Due to the immutable nature of RDDs. Enables easier straggler mitigation by running backup tasks (would be harder to achieve if mutable).

B. Describe the default fault recovery mechanism adopted by Spark. In the context of the application above, what additional measures (and on which RDDs) need to be taken to reduce the fault recovery time? Give a potential drawback of the additional measures you outlined.

Sol: Uses lineage approach. Checkpointing of the ranks RDD is necessary since the lineage can grow. Checkpointing incurs storage overhead.

C. Briefly outline a way in which you can improve the overall execution speed of the above application (hint: think co-partitioning tables). Explain how it helps relative to a naive alternative.

Sol: Use custom partitioning and partition links and ranks in the same way. Leads to narrow dependency and thus reduces shuffle.

**3. DRF and YARN:**

A. Consider a system with 10 CPUs, 30 GB RAM, and three users, where user A runs tasks with demand vector <2 CPU, 3 GB>, user B runs tasks with demand vector <1 CPU, 6 GB> and user C runs tasks with demand vector <2 CPU, 1 GB>. Here tasks are scheduled using DRF. How many tasks will be allocated by DRF for each user? How many resources will be unused in the system? (You do not have to write the iterative steps involved in the DRF algorithm. You just need to write the output i.e. no. of tasks)

Sol: User A - 2, User B - 2, User C - 2. Unused CPU = 0, Unused Memory = 10

B. Would DRF be an ideal strategy for applications with inter-task dependencies? (e.g. a reduce task can be scheduled only after its map task). Support your answer using an example.

Sol: No, it won’t be ideal. (Something similar to the motivation of graphene)

C. Would DRF be an ideal strategy for applications with inter-task dependencies? (e.g. a reduce task can be scheduled only after its map task). Support your answer using an example.

Sol: Advantages: Is optimal on global metrics like avg. JCT, fairness, makespan.

Disadvantages: Does not scale with the number of jobs.

**4. Spark Streaming and Storm**

Streaming systems broadly adhere to two models (a) mini-batched model and (b) record-at-a-time (i.e. continuous operator) model.

A. Name two performance bottlenecks in Storm. Explain why Storm cannot support exactly-once processing.

- One slow task slows down other tasks on executor since slows down queue.

Shared use of queue by multiple topologies. Any queue builds up would cause all topologies to run at speed of slowest topology in cluster. (This is the main point)

- Overheads to support at least once semantics (Sec 3.3.2 of the paper)

- Overheads due to externalizing heartbeat state to an external storage daemon

Cannot support exactly once:

Metadata node tracks if input tuple has been processed by all relevant nodes. But if processed by some and then restarted, it may get processed multiple times.

Another reasoning is as follows - for exactly once semantics -- we essentially want to guarantee that the state updates due to an incoming message happen only once. This is tricky to do in storm as in storm we replay on failure (or when an ACK is not received). When an ACK is not received, it is non-trivial to know whether the ACK is lost or the message. Trident solves it by introducing transactions, Spark Streaming solves it by introducing mini-batches.

B. Define a discretized-stream (D-stream).

Realizing a stream processing job as a series of time-ordered batch jobs / structuring computations as a set of short, stateless, deterministic tasks instead of continuous, stateful operators.

C.One-way Spark streaming can handle delayed records (or out-of-order records) is to use a “wait time”. Outline one disadvantage of this approach.

It does not provide a guarantee of processing out-of-order record. Reason being, a record can always show up after the configured “wait time”.

D.Which model from (a) or (b) above do the systems -- Spark Streaming and Storm -- adhere to? Which system is naturally better in the following aspects and why:

a.Latency (i.e. Execution Latency - time from ingesting a record to getting a result)

b.Recovery from Failures / Straggler Mitigation

c. Network Usage

Spark Streaming is mini-batched, Storm is record-at-time.

a. Storm has better latency - because the latency in Storm is of the order of time to ingest and process a single record whereas in Spark Streaming it is of the order of ingesting and processing a mini-batch of records.

b. Spark Streaming has better recovery from failure and better Straggler Mitigation - because in Spark Streaming recovery of failed node(s) state is naturally parallelizable in both the different partitions of the operator and time

c. Spark Streaming has lesser network usage - because aggregation of output can happen naturally in a mini-batched model

1. Four big data projects
2. Other related knowledge