Programming Models and Frameworks:

Iterative Computation

Advanced Cloud Computing

15-719/18-847b

Garth Gibson
Greg Ganger
Majd Sakr

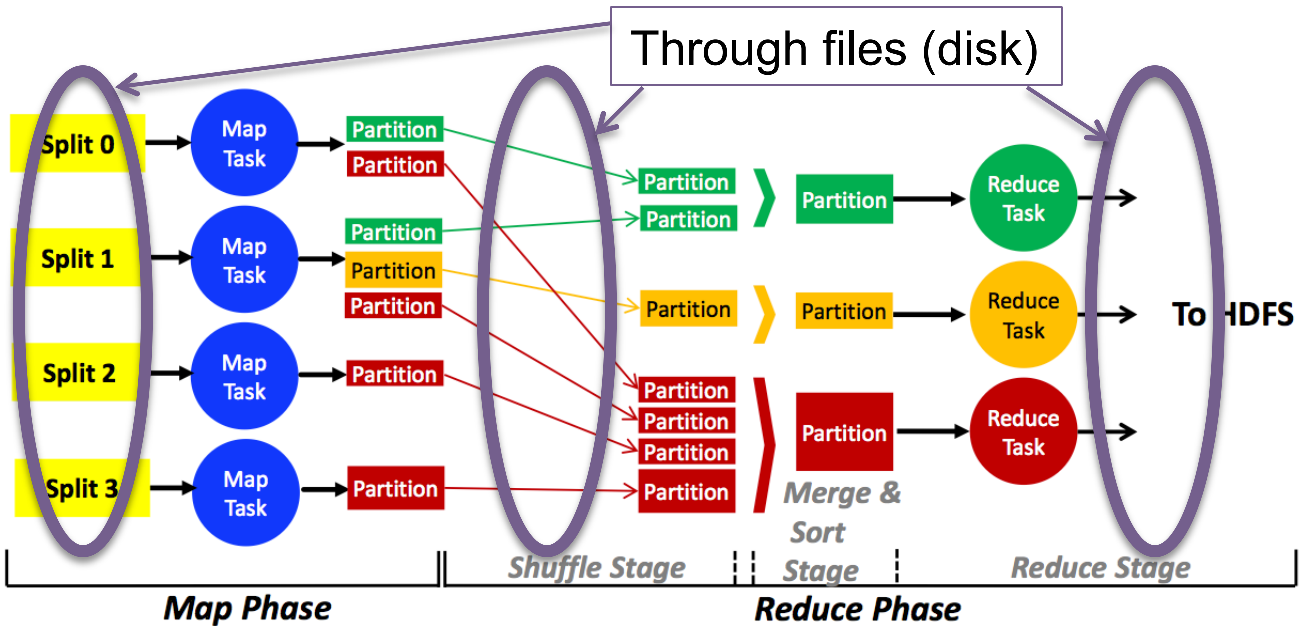
Advanced Cloud Computing Programming Models

- Ref 1: Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein (2010). "GraphLab: A New Parallel Framework for Machine Learning." Conf on Uncertainty in Artificial Intelligence (UAI). http://www.select.cs.cmu.edu/publications/scripts/papers.cgi
- Ref 2: Spark: cluster computing with working sets. Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, Ion Stoica. USENIX Hot Topics in Cloud Computing (HotCloud'10). http://www.cs.berkeley.edu/~matei/papers/2010/hotcloud_spark.pdf

Advanced Cloud Computing Programming Models

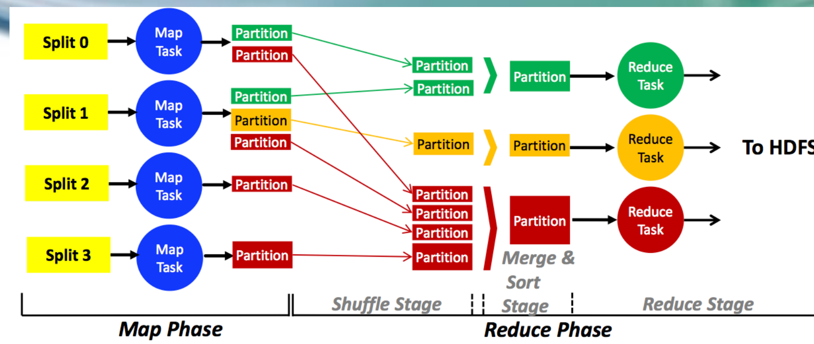
- Optional
- Ref 3: DyradLinQ: A system for general-purpose distributed data-parallel computing using a high-level language. Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey. OSDI'08. http://research.microsoft.com/en-us/projects/dryadlinq/dryadlinq.pdf
- Ref 5: TensorFlow: A system for large-scale machine learning. Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeff Dean, Matthieu Devin, Sanjay Ghemawatt, Geoffrey Irving, Michael Isard. OSDI'16. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

Map/Reduce as Elastic Big Data Processing



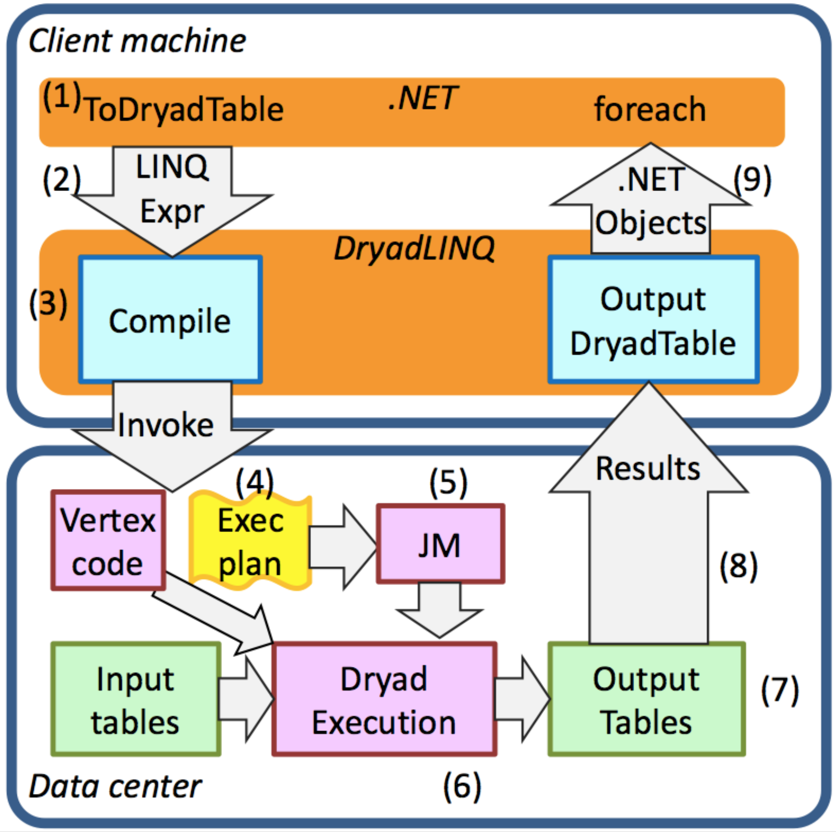
- Big data has lots of input: divide into many splits to be 'map'ed
- Queue map tasks on virtual cores
- Partition map task output to load balance work in reduce tasks
- Effective elastic exploitation of more data on map task side
  - Critical to scalability: partition function & reduce function
    - Unfortunate partition -> imbalanced load, degrade to little parallelism
    - Unfortunate reduce -> may need pre-sort (out of core), highly sensitive to real memory availability (too little -> more out of core; too much -> thrashes)

Spark as Map/Reduce 2.0



- Abstract as a sequential program in one machine (driver)
  - Driver sends work to a separate cluster (workers) to do map/reduce
- Combine map functions (spark calls these transformations)
  - E.g. rdd_x.map(foo).map(bar) is 2 passes of MR with null reduces
  - Spark creates function foo_bar() that combines foo() & bar() in map task
  - Spark transforms combine this way until a shuffle is unavoidable (stage)
- Is big data big? (100X prior examples is big, but might only be GBs)
  - Cache reduce outputs in memory (or discard & recompute as needed)
    - ‘cat <in | wc >out2’ versus ‘cat <in >out1; wc <out1 >out2’
    - for thin map() and reduce() functions, capturing out1 can be costly
- Automate splitting/partitioning (unless overridden)

DryadLinq



- Simplify efficient data parallel code
  - Compiler support for imperative and declarative (eg., database) operations
  - Extends MapReduce to workflows that can be collectively optimized
- Data flows on edges between processes at vertices (workflows)
- Coding is processes at vertices and expressions representing workflow
- Interesting part of the compiler operates on the expressions
  - Inspired by traditional database query optimizations – rewrite the execution plan with equivalent plan that is expected to execute faster

DryadLinq

- Data flowing through a graph abstraction
  - Vertices are programs (possibly different with each vertex)
  - Edges are data channels (pipe-like)
  - Requires programs to have **no side-effects** (no changes to shared state)
  - Apply function similar to MapReduce reduce – open ended user code
- Compiler operates on expressions, rewriting execution sequences
  - Exploits prior work on compiler for workflows on sets (LINQ)
  - Extends traditional database query planning with less type restrictive code
    - Unlike traditional plans, virtualizes resources (so might spill to storage)
  - Knows how to partition sets (hash, range and round robin) over nodes
  - Doesn't always know what processes do, so less powerful optimizer than database – where it can't infer what is happening, it takes hints from users
  - Can auto-pipeline, remove redundant partitioning, reorder partitionings, etc

Example: MapReduce (reduce-reorderable)

- DryadLinq compiler can pre-reduce, partition, sort-merge, partially aggregate
- In MapReduce you “configure” this youself

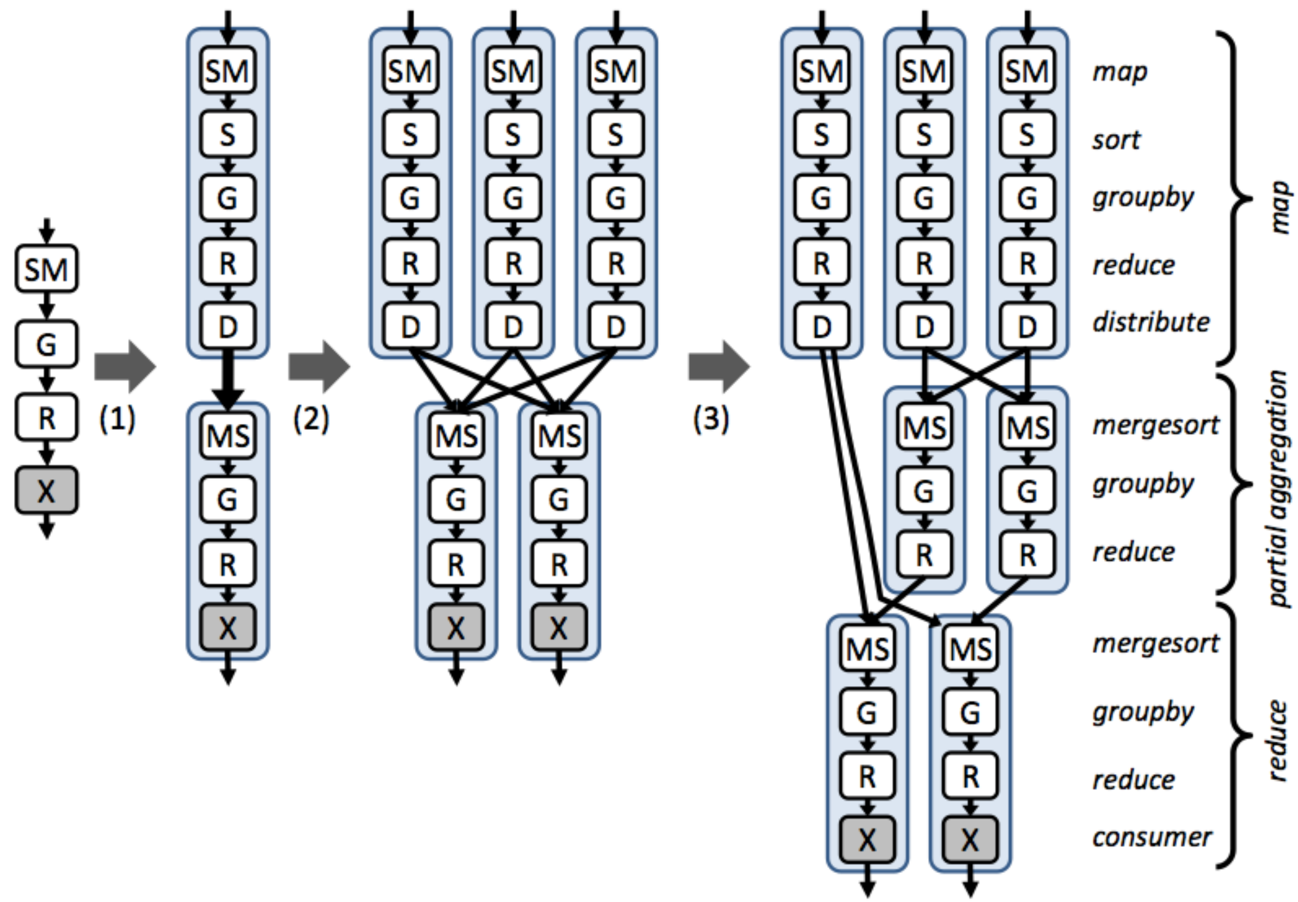


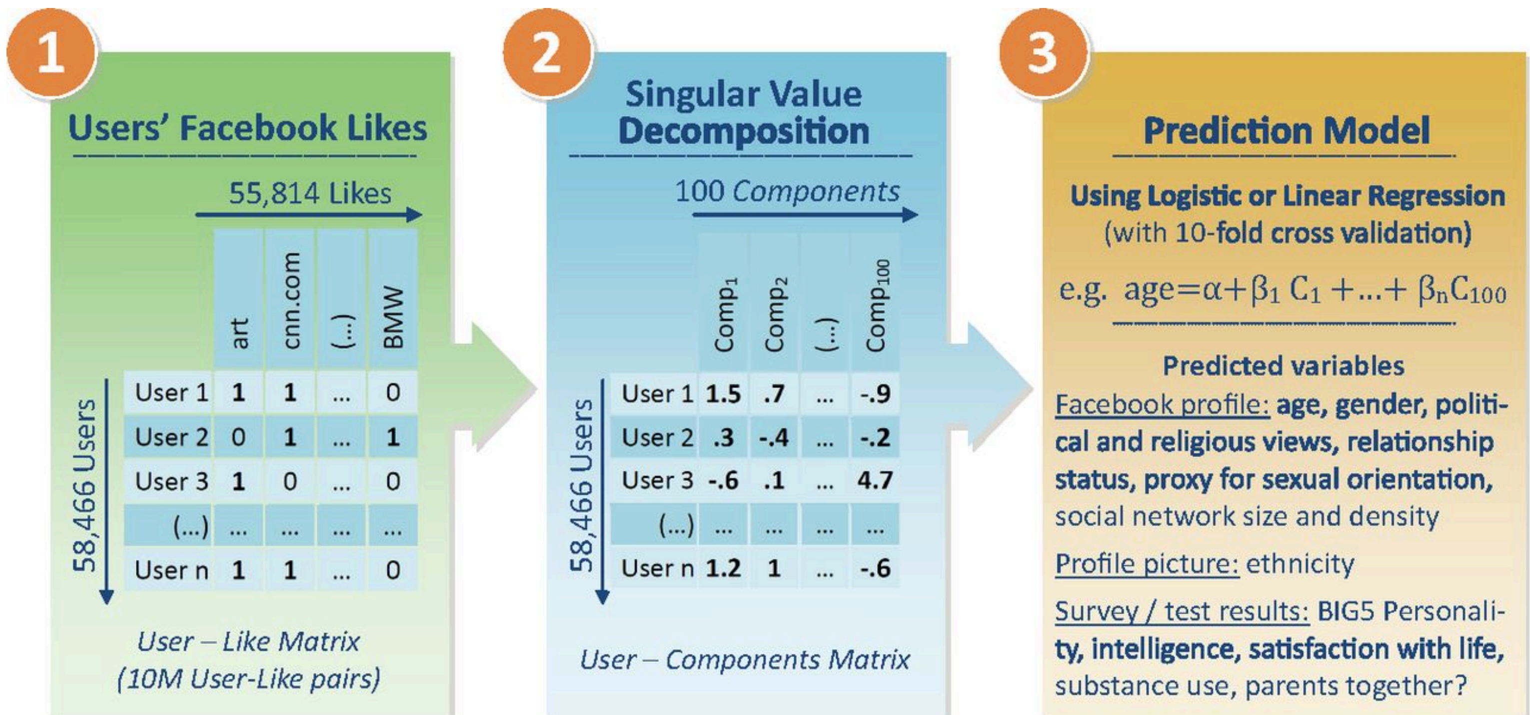
Figure 6: *Execution plan for MapReduce, described in Section 4.2.4.*

"Killer App" for Big Data:

Machine Learning

Machine Learning (ML) works

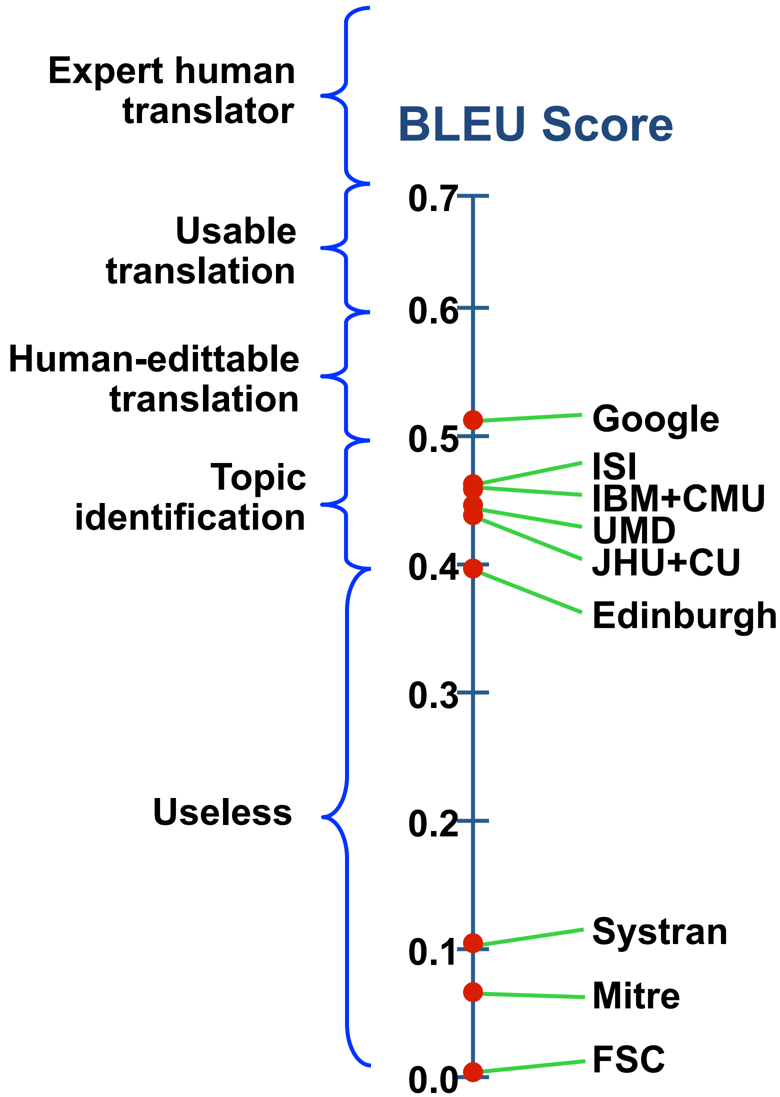
"… easily accessible digital records of behavior, Facebook Likes, can be used to automatically and accurately predict a range of highly sensitive personal attributes … model correctly discriminates between homosexual and heterosexual men in 88% of cases, African Americans and Caucasian Americans in 95% of cases, and between Democrat and Republican in 85% of cases."



"The study is based on a sample of 58,466 volunteers from the United States, obtained through the myPersonality Facebook application (www.mypersonality.org/wiki), which included their Facebook profile information, a list of their Likes (n = 170 Likes per person ..."

Kosinski M et al. PNAS 2013;110:5802-5805

2005 NIST Arabic-English Competition



Translate 100 articles

- 2005 : Google wins!

Qualitatively better 1st entry

Not most sophisticated approach

No one knew Arabic

Brute force statistics

But more data & compute !!

200M words from UN translations

1 billion words of Arabic docs

1000 processor cluster

➔ Can't compete w/o big data

Stages of Machine Learning

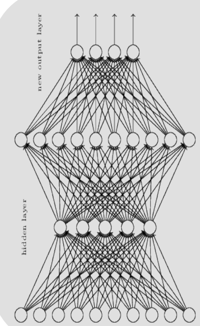
- Data collection
  - Logistics, cleaning, ….
- Model selection
  - Domain knowledge
- Data engineering
  - Extract, transform, ….
- Model training
  - Fit parameters to data
- Model inferencing
  - Predict/label outcome from model

YouTube

100+ hours video uploaded every minute

twitter

645 million users
500 million tweets / day



Google Brain
Deep Learning
for images:
1 Billion
model
parameters

Collaborative filtering
for Video recommendation:

NETFLIX

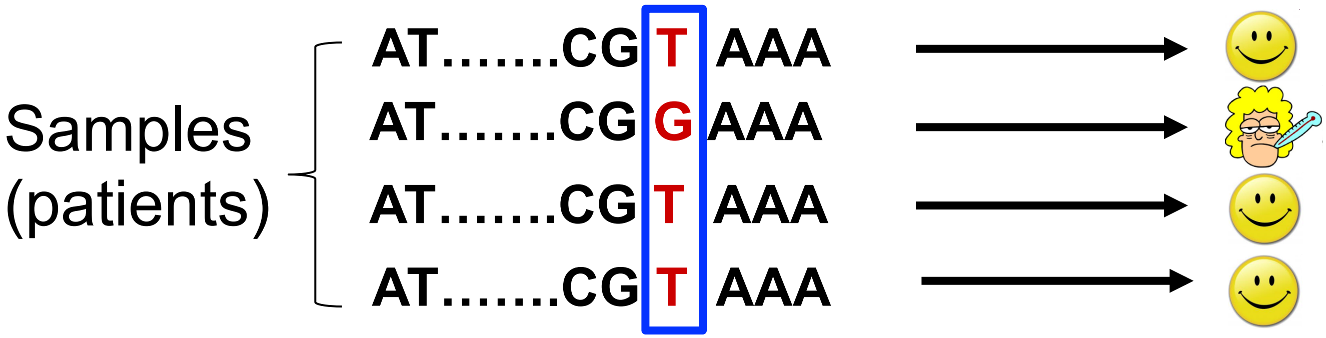
1 Billion
model
parameters

Stages of Machine Learning

- Data collection
  - Done mostly away from machine learning data center, then aggregated
- Model selection
  - Done offline from collection/engineering/training/inference
- Data engineering (project 2 part 1)
  - Multiple data passes (Map/Reduces), large data reduction
- Model training (project 2 part 2)
  - Start with a guess of parameters, test against recorded input and output data, adjust parameters, iterate many times (many data passes)
- Model inferencing
  - For one input, apply model and return one predicted output (no data passes)

Eg. Medical Research

- Collect human genome and disease outcome for lots of people
- Model disease probability as a linear model of presence of gene pairs



- Millions to 10^{11} (pair-wise genes) parameters; thousands of patients
- Model training is solving for “best” parameter weights
  - Under-determined set of equations for learning model of gene influence on disease; infinite number of parameter sets match observed outputs
  - Add figure of merit (objective function) to value a solution and search solution space for best merit

Model Training

$$\arg\max_{\vec{\theta}} \equiv \mathcal{L}(\{\mathbf{x}_i, \mathbf{y}i\}_{i=1}^{N} \; ; \; \vec{\theta}) + \Omega(\vec{\theta})$$

Model Data Parameter

Solved by an iterative convergent algorithm on vectors & matrices

```
for (t = 1 to T) {
  doThings()
```

$$\vec{\theta}^{t+1} = g(\vec{\theta}^{t}, \; \underline{\Delta_f \vec{\theta}(\mathcal{D})})$$

```
  doOtherThings()
}
```

This computation needs to be parallelized!

Machine Learning (ML) via MapReduce (MR)

0) Store engineered data and initial model parameters in files

1) Split engineered data to map tasks; replicate/broadcast parameters (this is known as “data parallel” decomposition)

2) Each map task tests model against data inputs & outputs and computes changes in model parameters; send changes to reducers

3) Reducers combine changes from different map splits of data and write a new model parameters file (and decide if training is over)

4) If training is not over, go to (1)

Problems with ML via MR

- If Hadoop, each map task and each reduce task are Java VM launch
- Iteration is in external scripts repeating Hadoop invocations
- Amount of compute per data item is not much
- No need to issue parameter update per data item; could

  pre-combine updates for same parameter in memory of each map
  - So shuffle is not a flow, but a single set of parameter updates per map
- Reducer function is simple add updates for each parameter
  - Most work is communication through file system
- It may scale but overhead is high

Spark for ML via MR

- Don't write reducer output to file system; cache in memory
- Don't re-read engineered data from file system; cache in memory
- For small numbers of parameters, driver collect & broadcast
- Combine map transformations to try for one shuffle per iteration
- Don't launch separate Java VMs for each map task; retain one VM for all tasks across all iterations

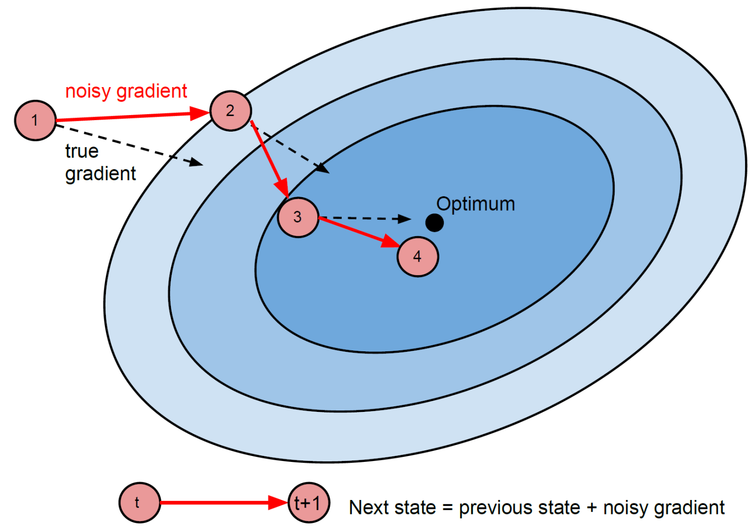
- Potential speedup is large – 10X in Spark paper

Parameter Servers

- ML via MR model moves parameter updates through MR shuffle to reducers, then combines all parameters into an RDD (possibly collected/broadcast by driver)
- Parameter Servers use a shared memory model for parameters
  - All map tasks can cache any/all parameters; changes are pushed to them
  - All reducers are replaced with atomic “add to parameter in shared mem”
  - Less data transmitted and less task overhead
  - Engineered data easily avoids repartitioning in the next iteration

Does ML via MR need to be synchronized

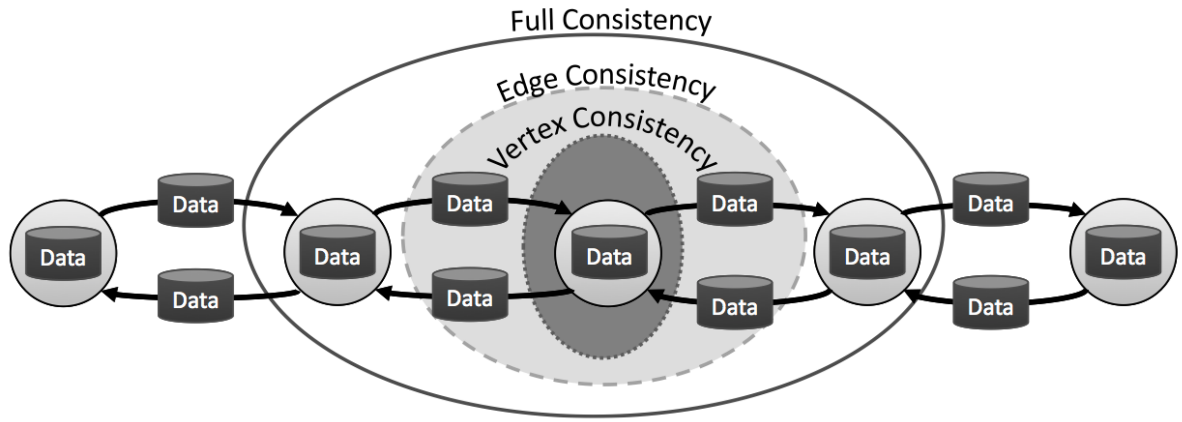
- Basic MR is functional; inputs are read-only, outputs write-only
  - All communication occurs through RDDs/file systems after one complete MR when a later MR reads the output file (RDD) of a prior MR
  - This separation of write-only output becoming read-only input is a barrier synchronization
- Parameter servers can be used synch or allowed to run asynch
  - Async works because ML is iterative approximation, converging to optimum provided async error is bounded



GraphLab: early tools for parameter servers

- GraphLab started not from Hadoop MR but from shared memory transaction processing – lots of parallel updates ordered by locks
- GraphLab provides a higher level programming model
  - Data is associated with vertices and edges between vertices, inherently sparse (or we'd use a matrix representation instead)
    - Non zeroes in a matrix representation are edges or vertices
    - Lots of machine learning data sets, like social media, are very sparse
  - Update: code updates a vertex and its neighbor vertices in isolation
  - Iteration: one complete pass over the input data, calculating updates (Fold in GraphLab paper), then combine changes (Apply in GraphLab)

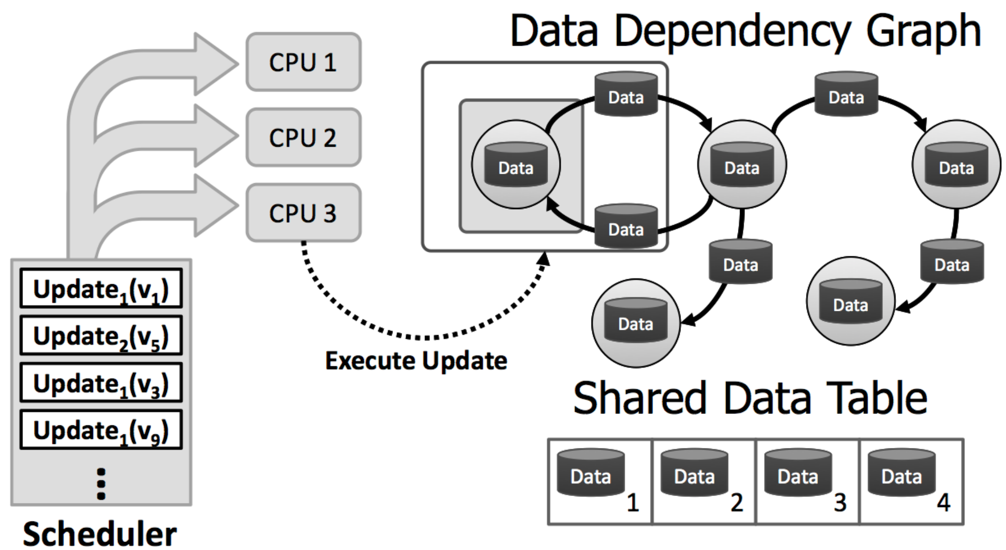
Consistency



(b) Consistency Models

- Many machine learning algorithms are tolerant of some data races
  - “converging” to good enough may depend on data and schedule
- Graphlab allows some updates to “overlap” (not fully lock)
  - Much more parallel than matrix multiply because of the sparseness
  - Totally safe if the transactions don’t update what is being overlapped
    - Ie., database serializable concurrent transactions guaranteed for edge or vertex consistency given restrictions on what the update code can do
- Natural for shared memory multithreaded update
  - Like HPC (distributed) simulation

Scheduling



- Graphlab allows some updates to do scheduling
  - Baseline is sequential execution of each vertex' update once per iteration
  - Sparseness allows non-overlapping updates to execute in parallel
  - Opportunity for smart schedulers to exploit more app properties
    - Prioritize specific updates over other updates because these communicate more information more quickly
    - Possible to execute some updates more often than others

Next day plan

- Project 2 part 2
- Cloud Storage comes next