Big Data Systems

Big Data Parallelism

- Huge data set
  - crawled documents, web request logs, etc.
- Natural parallelism:
  - can work on different parts of data independently
  - image processing, grep, indexing, many more

- Assume that you ran a large data analysis program
  - it took 10 hours on 1 node
  - it took 1 hour on 100 nodes
- What reasons can you come up with for this "suboptimal" performance? How would you debug?

Challenges

- Parallelize application
  - Where to place input and output data?
  - Where to place computation?
  - How to communicate data? How to manage threads? How to avoid network bottleneck?
- Balance computations
- Handle failures of nodes during computation
- Scheduling several applications who want to share infrastructure

Goal of MapReduce

- To solve these distribution/fault-tolerance issues once in a reusable library
  - To shield the programmer from having to re-solve them for each program
- To obtain adequate throughput and scalability
- To provide the programmer with a conceptual framework for designing their parallel program

Map Reduce

- Overview:
  - Partition large data set into M splits
  - Run map on each partition, which produces R local partitions; using a partition function R
    - Hidden intermediate shuffle phase
  - Run reduce on each intermediate partition, which produces R output files

Details

- Input values: set of key-value pairs
  - Job will read chunks of key-value pairs
  - “key-value” pairs a good enough abstraction
- Map(key, value):
  - System will execute this function on each key-value pair
  - Generate a set of intermediate key-value pairs
- Reduce(key, values):
  - Intermediate key-value pairs are sorted
  - Reduce function is executed on these intermediate key-values

Count words in web-pages

```
Map(key, value) {
   // key is url
   // value is the content of the url
   For each word W in the content
      Generate(W, 1);
}

Reduce(key, values) {
   // key is word (W)
   // values are basically all 1s
   Sum = Sum all 1s in values

   // generate word-count pairs
   Generate (key, sum);
}
```

Reverse web-link graph

Go to google advanced search:
"find pages that link to the page:" cnn.com

```
Map(key, value) {
   // key = url
   // value = content
   For each url, linking to target
      Generate(output target, url);
}

Reduce(key, values) {
   // key = target url
   // values = all urls that point to the target url
   Generate(key, list of values);
}
```

- Question: how do we implement “join” in MapReduce?
  - Imagine you have a log table L and some other table R that contains say user information
  - Perform Join (L.uid == R.uid)
    - Say size of L >> size of R
    - Bonus: consider real world zipf distributions

Implementation

- Depends on the underlying hardware: shared memory, message passing, NUMA shared memory, etc.
- Inside Google:
  - commodity workstations
  - commodity networking hardware (1Gbps at node level and much smaller bisection bandwidth)
  - cluster = 100s or 1000s of machines
  - storage is through GFS

MapReduce Input

- Where does input come from?
  - Input is striped+replicated over GFS
  - Typically, Map reads from a local disk
- Tradeoff:
  - Good: Map reads at disk speed (local access)
  - Bad: only 2-3 choices of where Map task can be run

Intermediate Data

- Where does MapReduce store intermediate data?
  - On the local disk of the Map server (not GFS)
- Tradeoff:
  - Good: fast local access
  - Bad: only one copy, problem for fault-tolerance, load-balance

Output Storage

- Where does MapReduce store output?
  - In GFS: replicated, separate file per Reduce task
  - Output requires network communication — slow
  - Used for subsequent MapReduce tasks

Scaling

- Map calls probably scale
- Reduce calls also probably scale
  - But must be mindful of keys with many values
- Network may limit scaling
- Stragglers could be a problem

Fault Tolerance

- Main idea: map, reduce are deterministic, functional, and independent
  - Simply re-execute
- What if a worker fails while running map?
- What if Map has started to produce output, then crashed?
- What if a worker fails while running Reduce?

Load Balance

- What if some Map machines are faster than others?
  - Or some input splits take longer to process?
  - Need more input splits than machines
- Stragglers:
  - load balance only balances newly assigned tasks
  - Always schedule multiple copies of very last tasks

Discussion

- What are the constraints imposed on map and reduce functions?
- How would you like to expand the capability of map reduce?

Map Reduce Criticism

- “Giant step backwards” in programming model
- Sub-optimal implementation
- “Not novel at all”
- Missing most of the DB features
- Incompatible with all of the DB tools

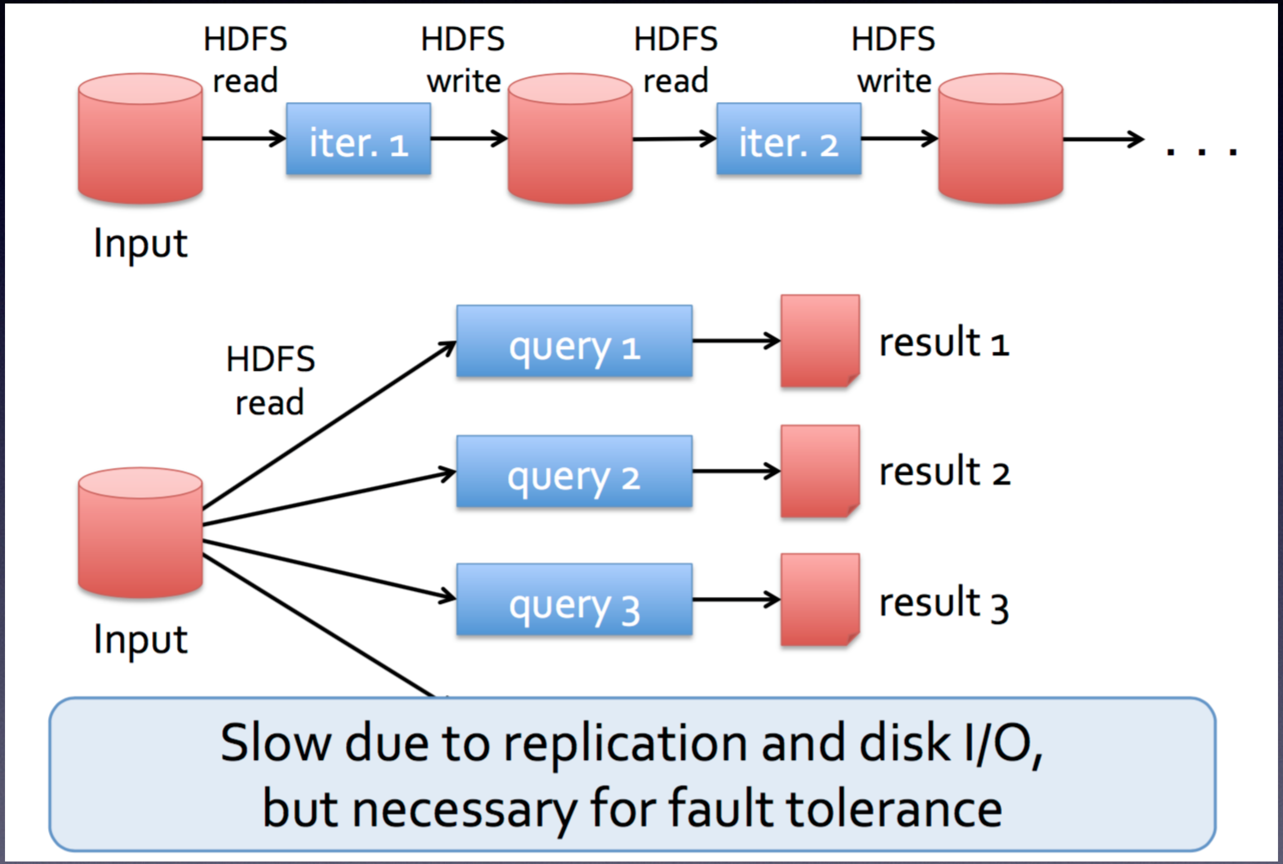
Comparison to Databases

- Huge source of controversy; claims:
  - parallel databases have much more advanced data processing support that leads to much more efficiency
    - support an index; selection is accelerated
    - provides query optimization
  - parallel databases support a much richer semantic model
    - support a scheme; sharing across apps
    - support SQL, efficient joins, etc.

Where does MR win?

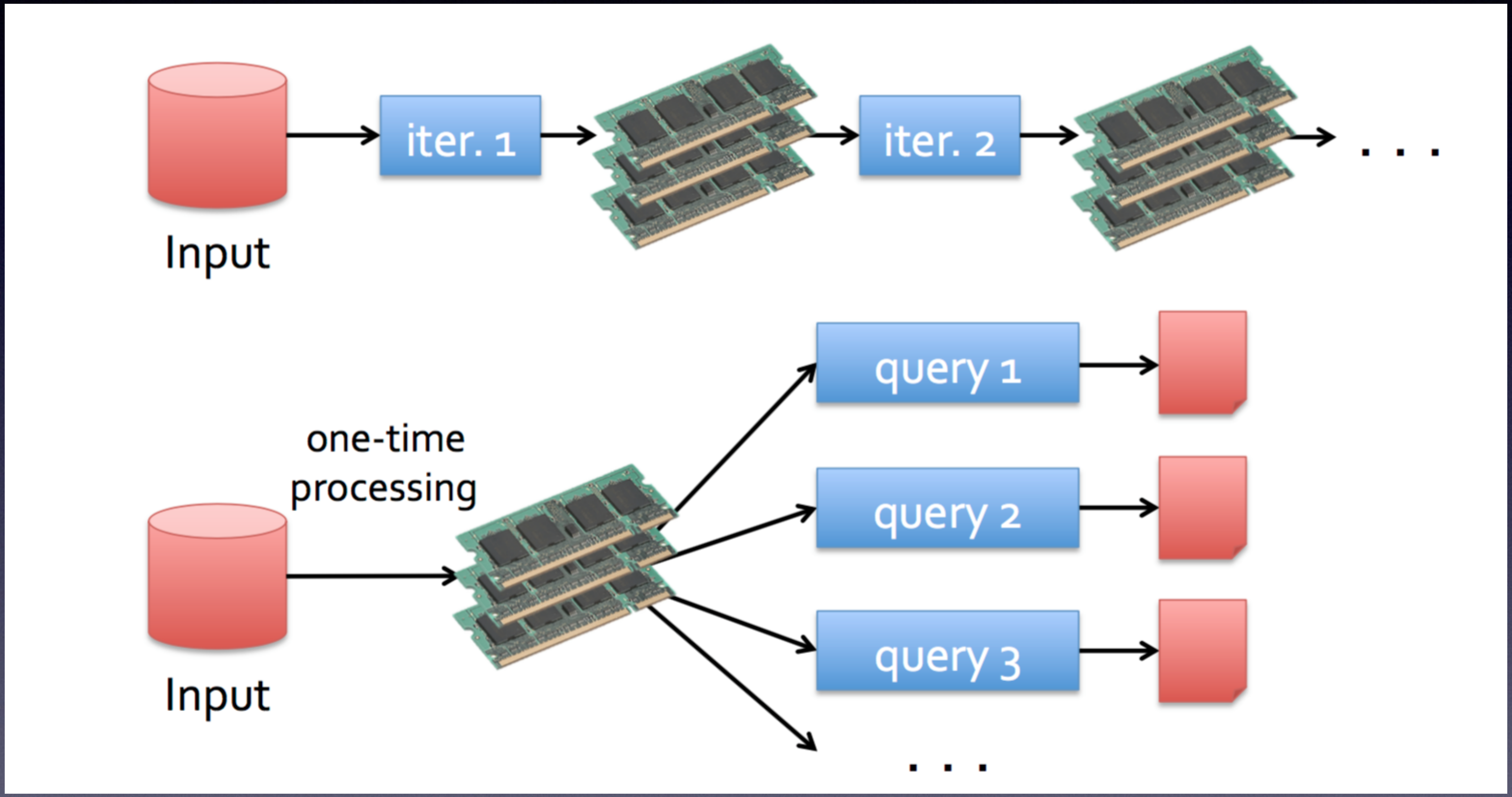
- Scaling
- Loading data into system
- Fault tolerance (partial restarts)
- Approachability

Map Reduce Performance



In MapReduce, the only way to share data across jobs is stable storage -> **slow**!

Spark Goal: In-Memory Data Sharing

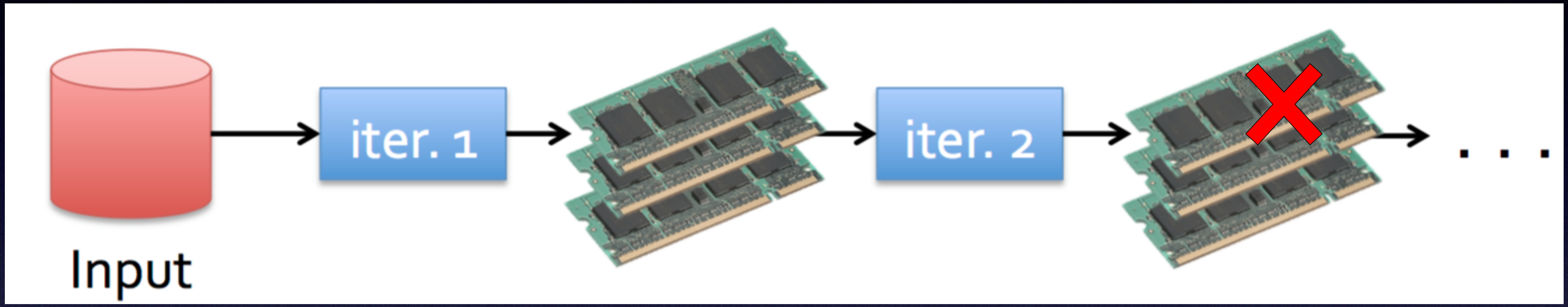


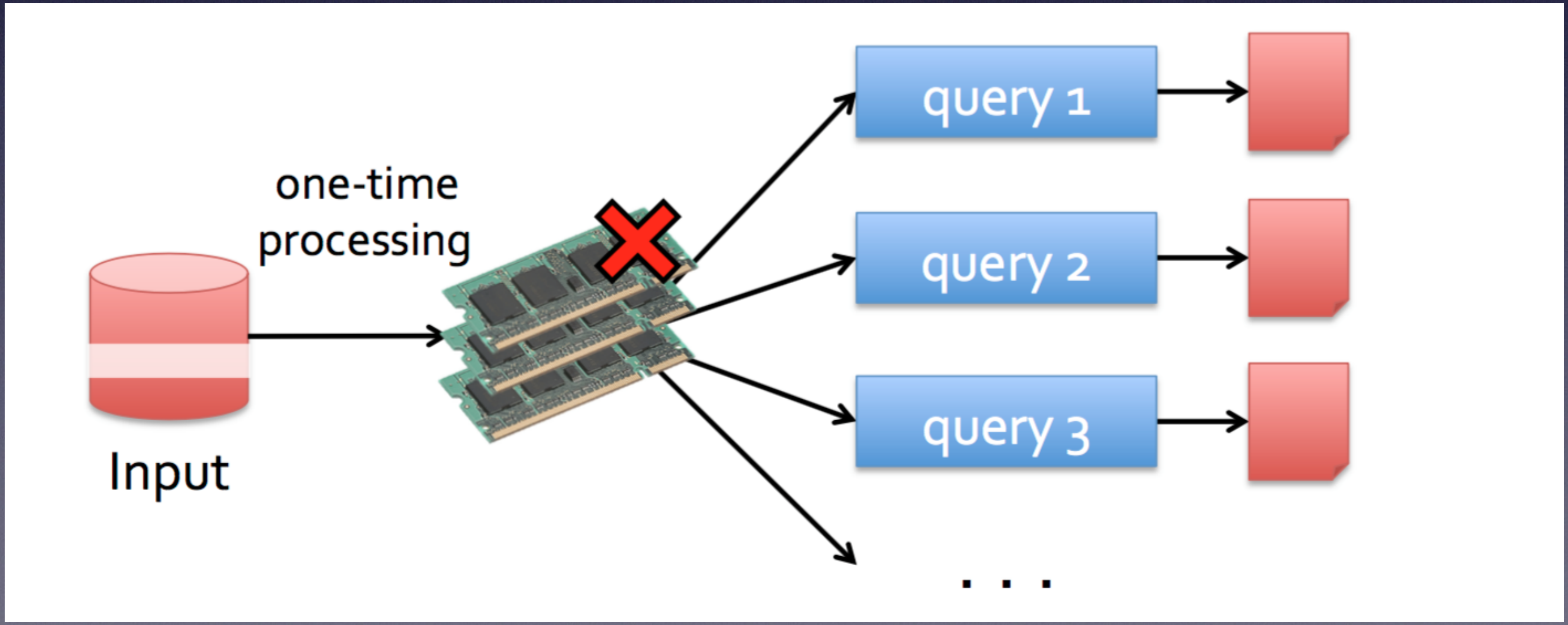
How to build a distributed memory abstraction that is fault tolerant and efficient?

Resilient Distributed Datasets (RDDs)

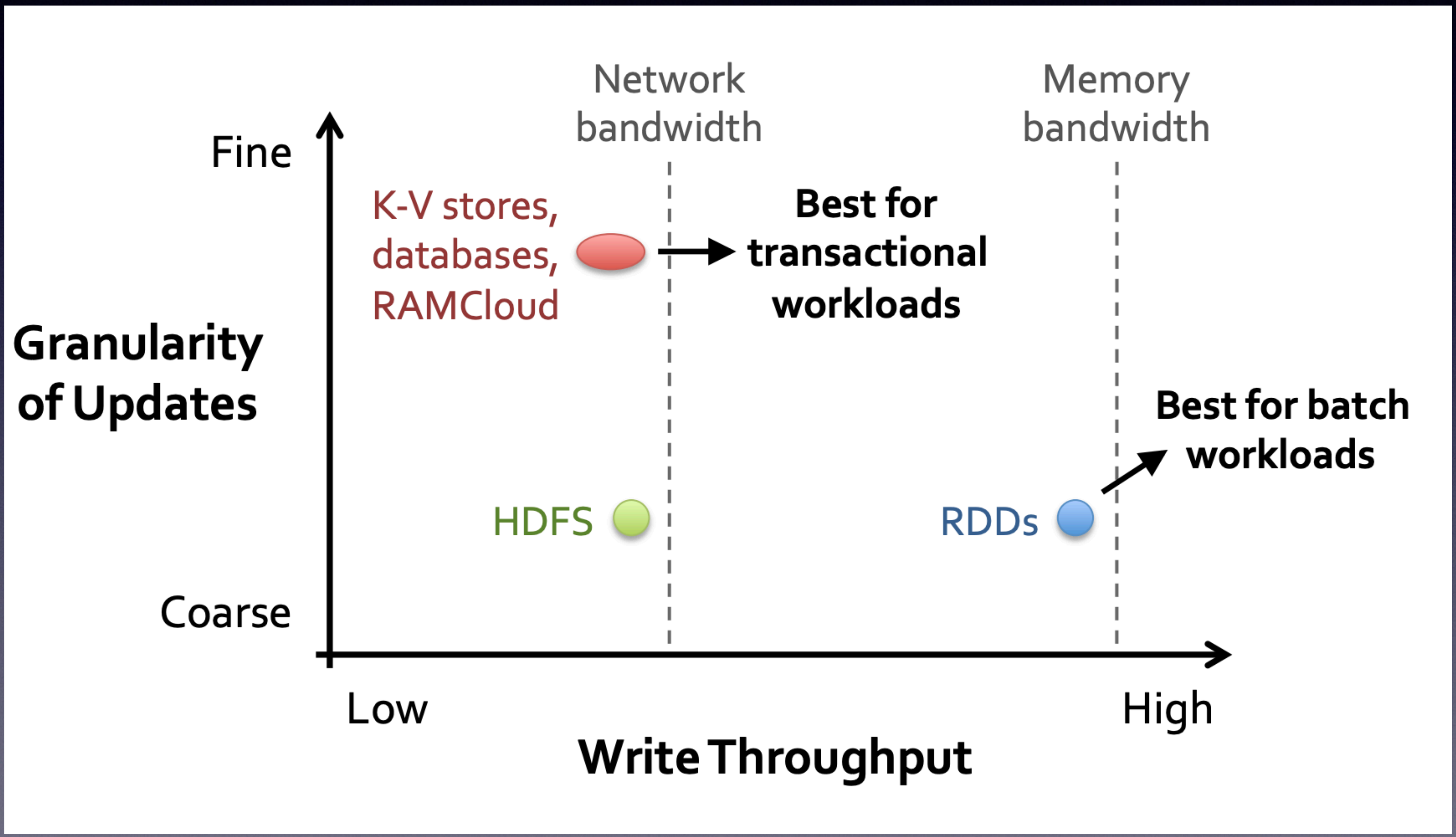
- Restricted form of distributed shared memory
  - Immutable, partitioned collection of records
  - can only be built through coarse-grained deterministic transformations (map, filter, join)
- Efficient fault tolerance through lineage
  - Log coarse-grained operations instead of fine-grained data updates
  - RDD has enough information about its derivation
  - Recompute lost partitions on failure

Fault-tolerance





Design Space



Example: Console Logs

Base RDD

Transformed RDD

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(‘\t’)[2])
messages.persist()
```

Action

```
messages.filter(lambda s: “foo” in s).count()
messages.filter(lambda s: “bar” in s).count()
. . .
```

RDD Fault Tolerance

- Track lineage to recompute lost data

```
messages = textFile(...).filter(lambda s: s.contains(“ERROR”))
                        .map(lambda s: s.split(‘\t’)[2])
```

RDD Implementation

- List of partitions
- Parent partition
    - Narrow: depends on one parent (e.g., map)
    - Wide: depends on several parents (e.g., join)
- Function to compute (e.g., map, join)
- Partitioning scheme
- Computation placement hint

RDD Computations

- Spark uses the lineage to schedule job
  - Transformation on the same partition form a stage
    - Joins, for example, are a stage boundary
    - Need to reshuffle data
  - A job runs a single stage
    - pipeline transformation within a stage
  - Schedule job where the RDD partition is

Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors} rank_i / |neighbors_i|$$

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

What are the performance and fault tolerance issues in this code?

PageRank

- Co-locate ranks and links
- Each iteration creates two new RDDs: ranks, temp
- Long lineage graph!
  - Risky for fault tolerance.
  - One node fails, much recomputation
- Solution: user can replicate RDD
  - Programmer pass "reliable" flag to persist()
  - Replicates RDD in memory
  - With REPLICATE flag, will write to stable storage (HDFS)

Tensorflow: System for ML

- Open source, lots of developers
- Used in RankBrain, Photos, SmartReply

Three types of ML

- Large scale training
- Low latency inference
- Testing new ideas (single node prototyping systems)

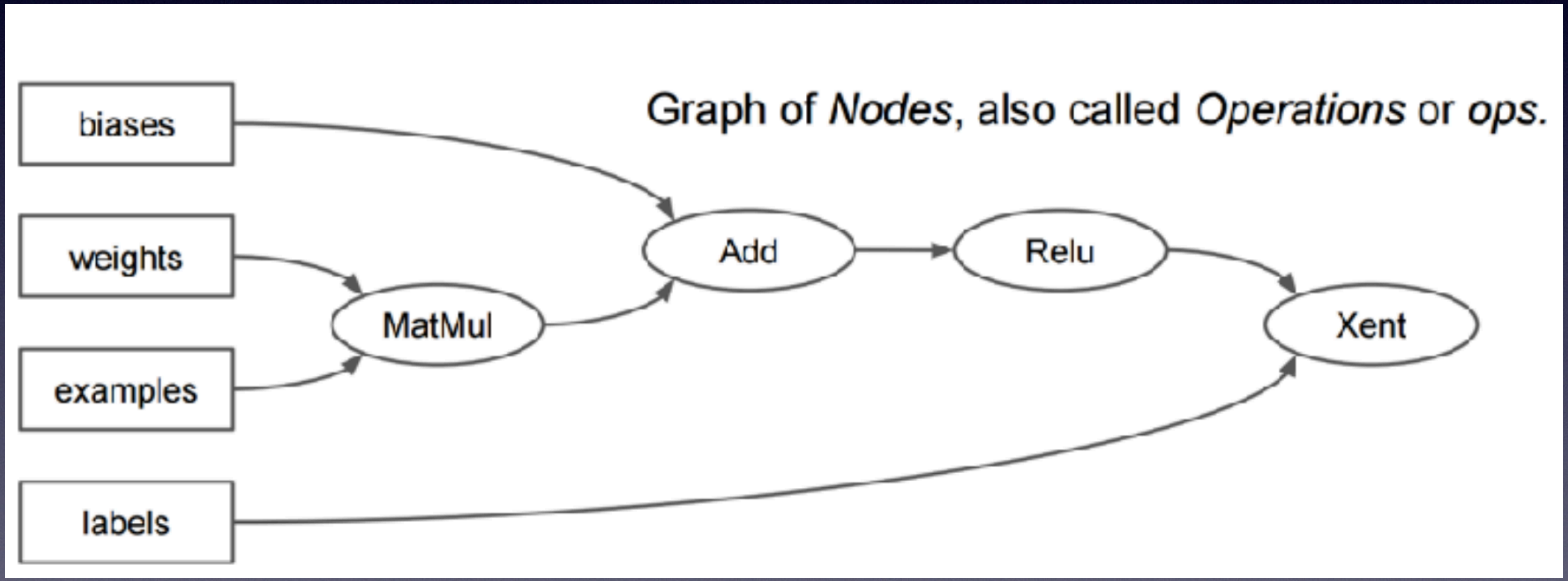
TensorFlow

- Common way to write programs
- Dataflow + Tensors
- Mutable state
- Simple mathematical operations
- Automatic differentiation

Background: NN Training

- Take input image
- Compute loss function (forward pass)
- Compute error gradients (backward pass)
- Update weights
- Repeat

Dataflow Graph



System Architecture

- Parameter server architecture
  - Stateless workers, stateful parameter servers (DHT)
  - Commutative updates to parameter server
- Data parallelism vs. model parallelism
  - Every worker works on the entire data flow graph (data parallelism)
  - Model and layers split across workers (model parallelism)
- What are the tradeoffs of different types of parallelism?

Synchrony

- Asynchronous execution is sometimes helpful (stragglers)
- Asynchrony causes consistency problems
- TensorFlow pursues synchronous execution
  - But adds k backup nodes to address straggler problems

Open Research Problems

- Automatic data placement
- Efficient code generation from data flow graph