MapReduce and Dryad

CS227

Li Jin, Jayme DeDona

Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

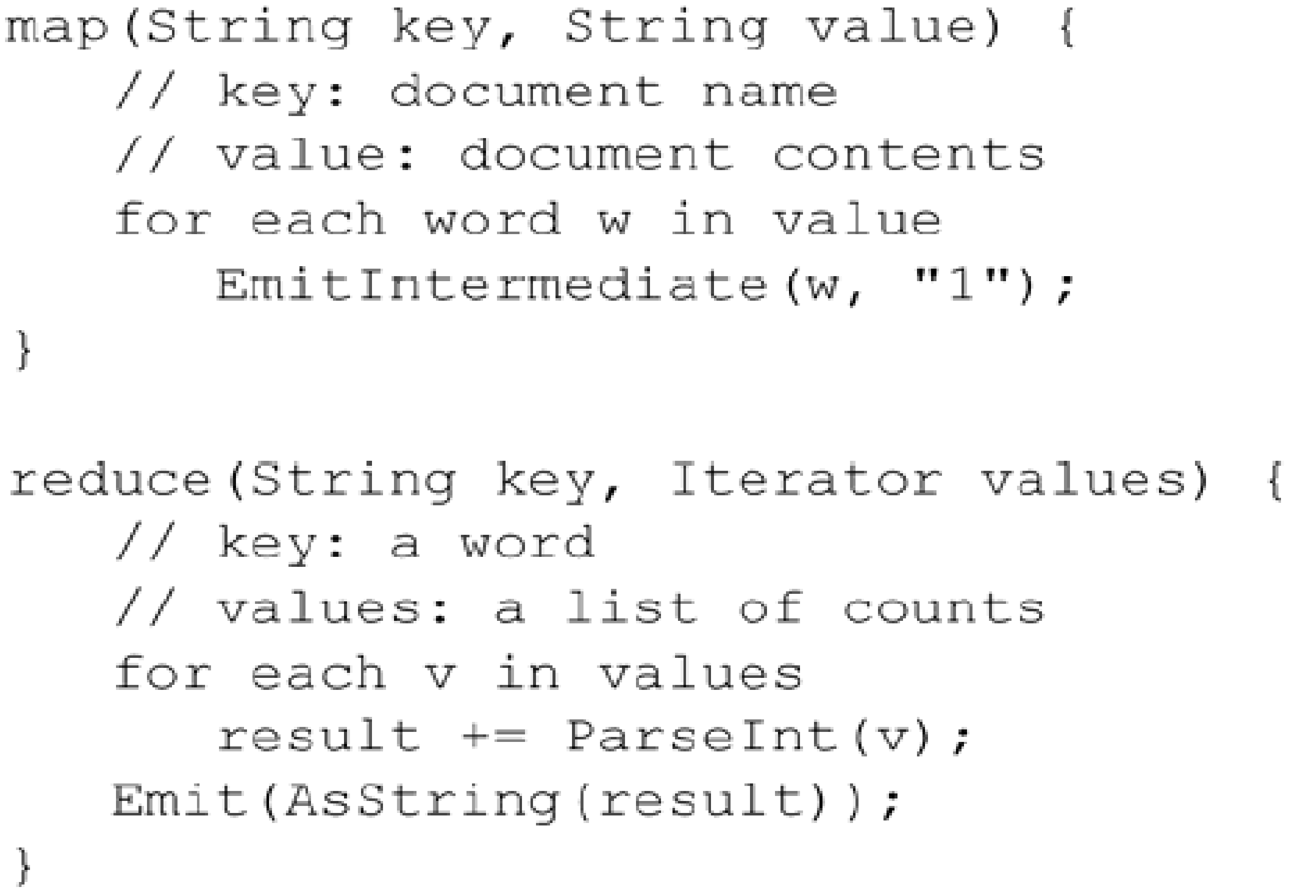
Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

Map/Reduce function

- Map
  - For each pair in a set of key/value pairs, produce a new key/value pair.
- Reduce
  - For each key
    - Look at all the values associated with that key and compute a new value.

Map/Reduce Function Example



```
map(String key, String value) {
    // key: document name
    // value: document contents
    for each word w in value
       EmitIntermediate(w, "1");
}

reduce(String key, Iterator values) {
    // key: a word
    // values: a list of counts
    for each v in values
       result += ParseInt(v);
    Emit(AsString(result));
}
```

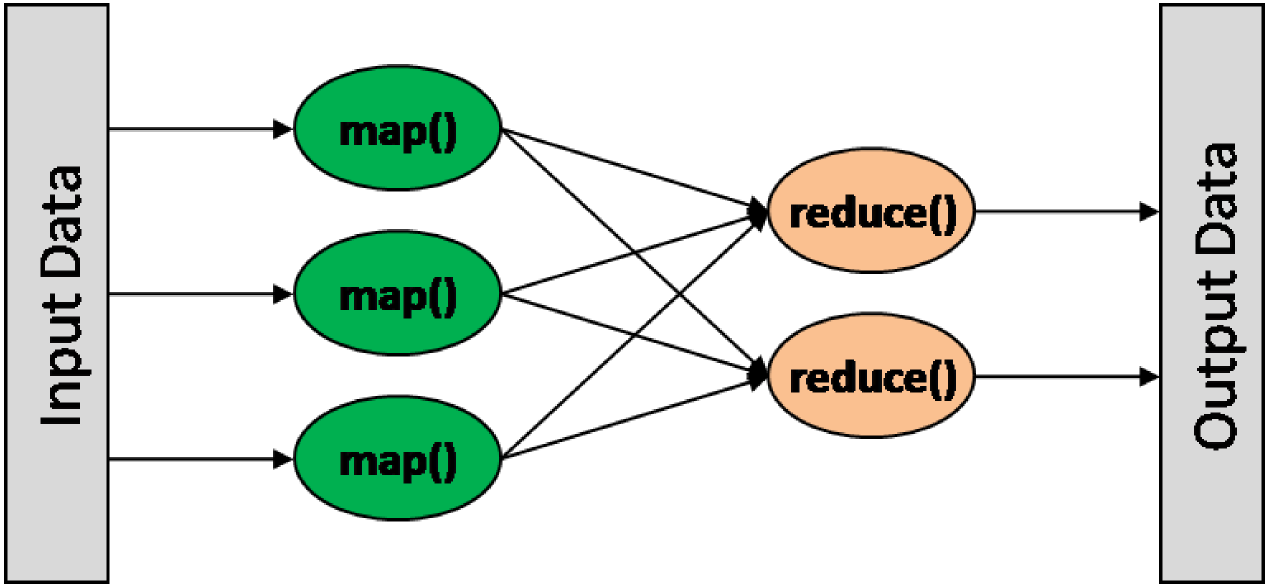
Implementation Sketch

- Map’s input pairs divided into M splits
  - stored in DFS
- Output of Map divided into R pieces
- One master process is in charge: farms out work to W worker processes.
  - each process on a separate computer

Implementation Sketch

- Master partitions splits among some of the workers
  - Each worker passes pairs to map function
  - Results stored in local files
    - Partitioned into R pieces
  - Remaining works perform reduce tasks
    - The R pieces are partitioned among them
    - Place remote procedure calls to map workers to get data
    - Put output to DFS

Implementation Sketch

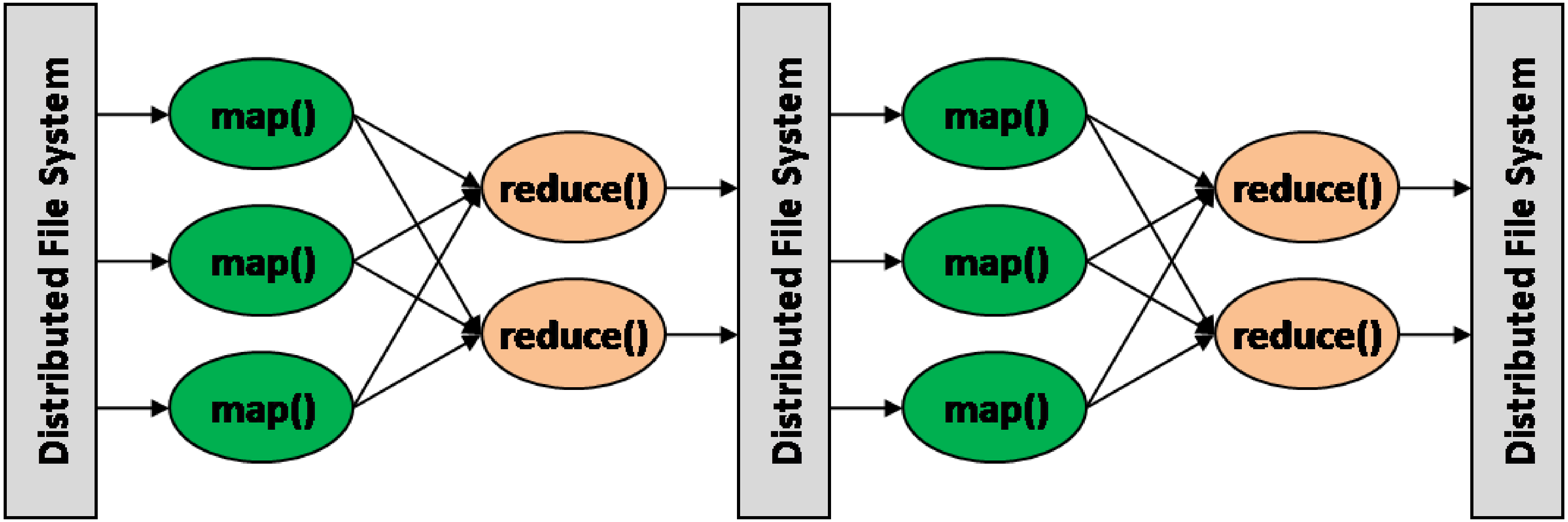


Split
[k1, v1]

Sort
by k1

Merge
[k1, [v1, v2, v3 ...]]

Implementation Sketch



More Details

- Input files split into M pieces, 16MB-64MB each.
- A number of worker machines are started
  - Master schedules M map tasks and R reduce tasks to workers, one task at a time
  - Typical values:
    - M = 200,000
    - R = 5000
    - 2000 worker machines.

More Details

- Worker assigned a map task processes the corresponding split, calling the map function repeatedly; output buffered in memory
- Buffered output written periodically to local files, partitioned into R regions.
  - Locations sent back to master

More Details

- Reduce tasks
  - Each handles one partition
  - Access data from map workers via RPC
  - Data is sorted by key
  - All values associated with each key are passed to the reduce function
  - Result appended to DFS output file

Coping with Failure

- Master maintains state of each task
  - Idle (not started)
  - In progress
  - Completed
- Master pings workers periodically to determine if they’re up

Coping with Failure

- Worker crashes
  - In-progress tasks have state set back to idle
    - All output is lost
    - Restarted from beginning on another worker
  - Completed map tasks
    - All output is lost
    - Restarted from beginning on another worker
    - Reduce tasks using output are notified of new worker

Coping with Failure

- Worker crashes(continued)
  - Completed reduce tasks
    - Output already on DFS
    - No restart necessary
- Master crashes
  - Could be recovered from checkpoint
  - In practice
    - Master crashes are rare
    - Entire application is restarted

Counterpoint

- MapReduce: A major step backwards
  - http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/
    - A giant step backward in the programming paradigm for large-scale data intensive applications
    - Sub optimal. Use brute force instead of indexing
    - Not novel at all – it represents a specific implementation of well known techniques nearly 25 years ago
    - …

Countercounterpoint

- Mapreduce is not a database system, so don’t judge it as one
- Mapreduce has excellent scalability; the proof of Google’s use
- Mapreduce is cheap and databases are expensive. (As a countercountercounterpoint to this, a Vertica guy told me they ran 3000 times faster than a hadoop job in one of their client’s cases)

Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

Dryad goals

- General-purpose execution environment for distributed, data-parallel applications
  - Concentrates on throughput not latency
  - Assumes private data center
- Automatic management of scheduling, distribution, fault tolerance, etc.

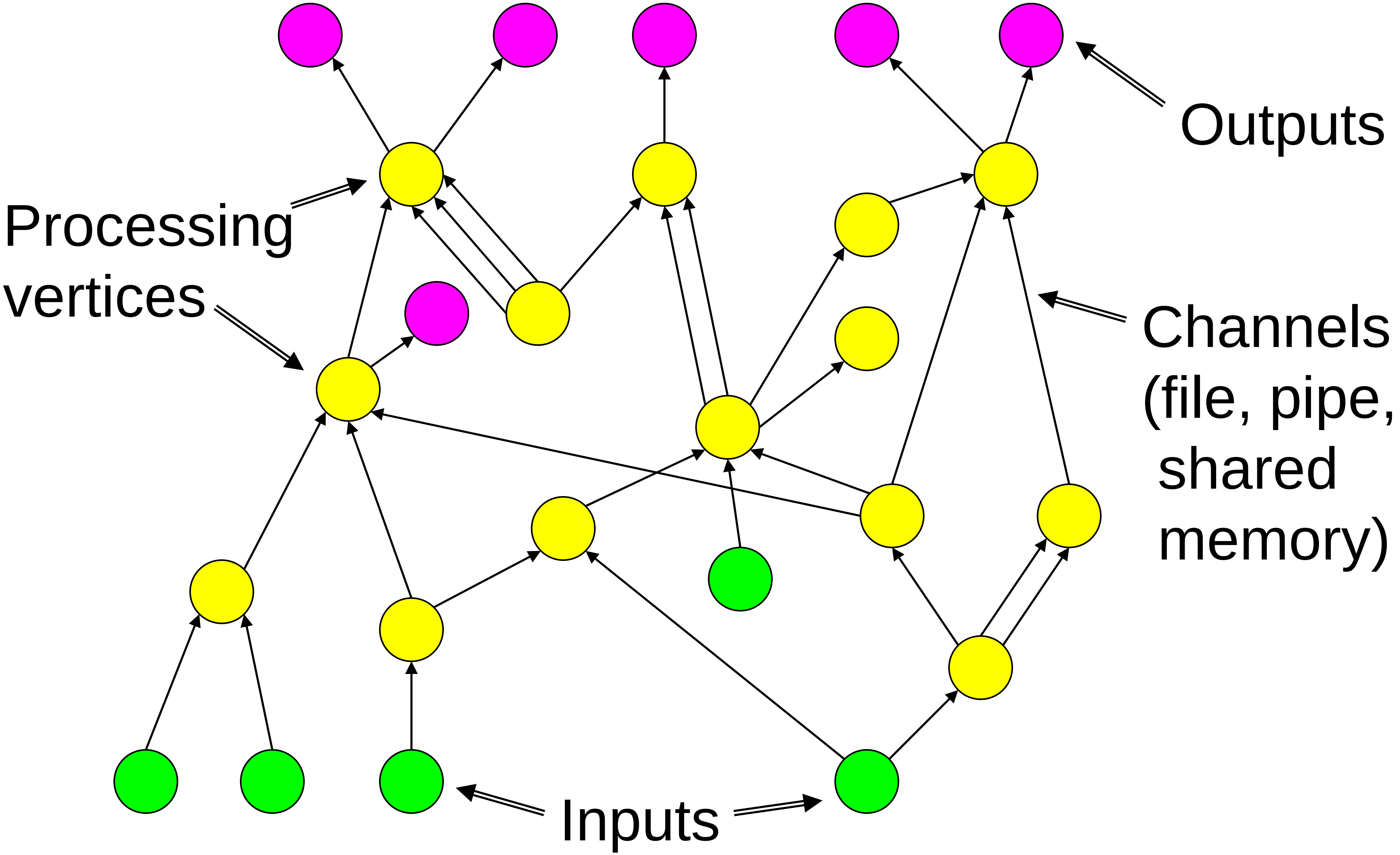
Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

Where does Dryad fit in the stack?

- Many programs can be represented as a distributed execution graph
- Dryad is middleware abstraction that runs them for you
  - Dryad sees arbitrary graphs
    - Simple, regular scheduler, fault-tolerance, etc.
    - Independent of programming model
  - Above Dryad is graph manipulation

Job = Directed Acyclic Graph



Processing vertices

Outputs

Channels (file, pipe, shared memory)

Inputs

Inputs and Outputs

- “Virtual” graph vertices
- Extensible abstraction
- Partitioned distributed files
  - Input file expands to set of vertices
    - Each partition is one virtual vertex
  - Output vertices write to individual partitions
    - Partitions concatenated when outputs completes

Channel Abstraction

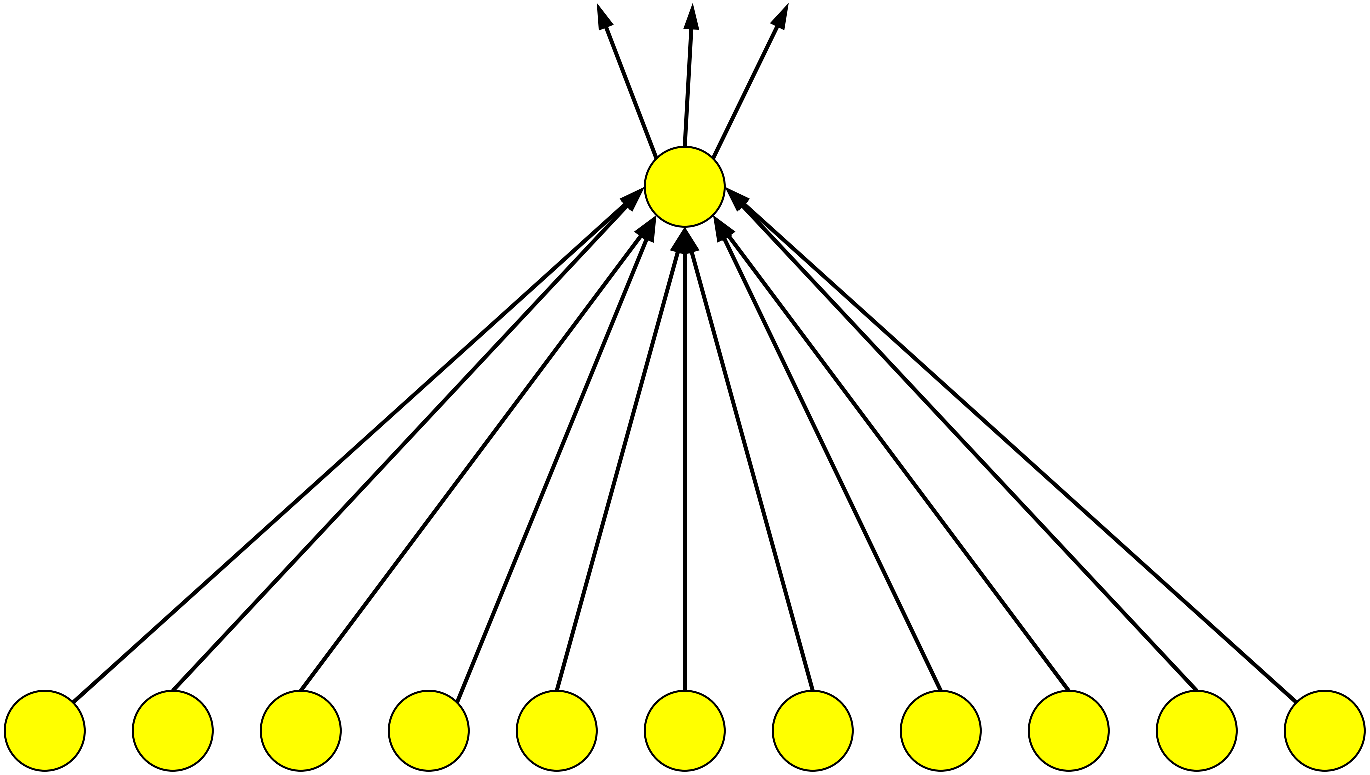
- Sequence of structured (typed) items
- Implementation
  - Temporary disk file
    - Items are serialized in buffers
  - TCP pipe
    - Items are serialized in buffers
  - Shared-memory FIFO
    - Pass pointers to items directly
- Simple, general data model

Why a Directed Acyclic Graph?

- Natural “most general” design point
- Allowing cycles causes trouble
- Mistake to be simpler
  - Supports full relational algebra and more
    - Multiple vertex inputs or outputs of different types
  - Layered design
    - Generic scheduler, no hard-wired special cases
    - Front ends only need to manipulate graphs

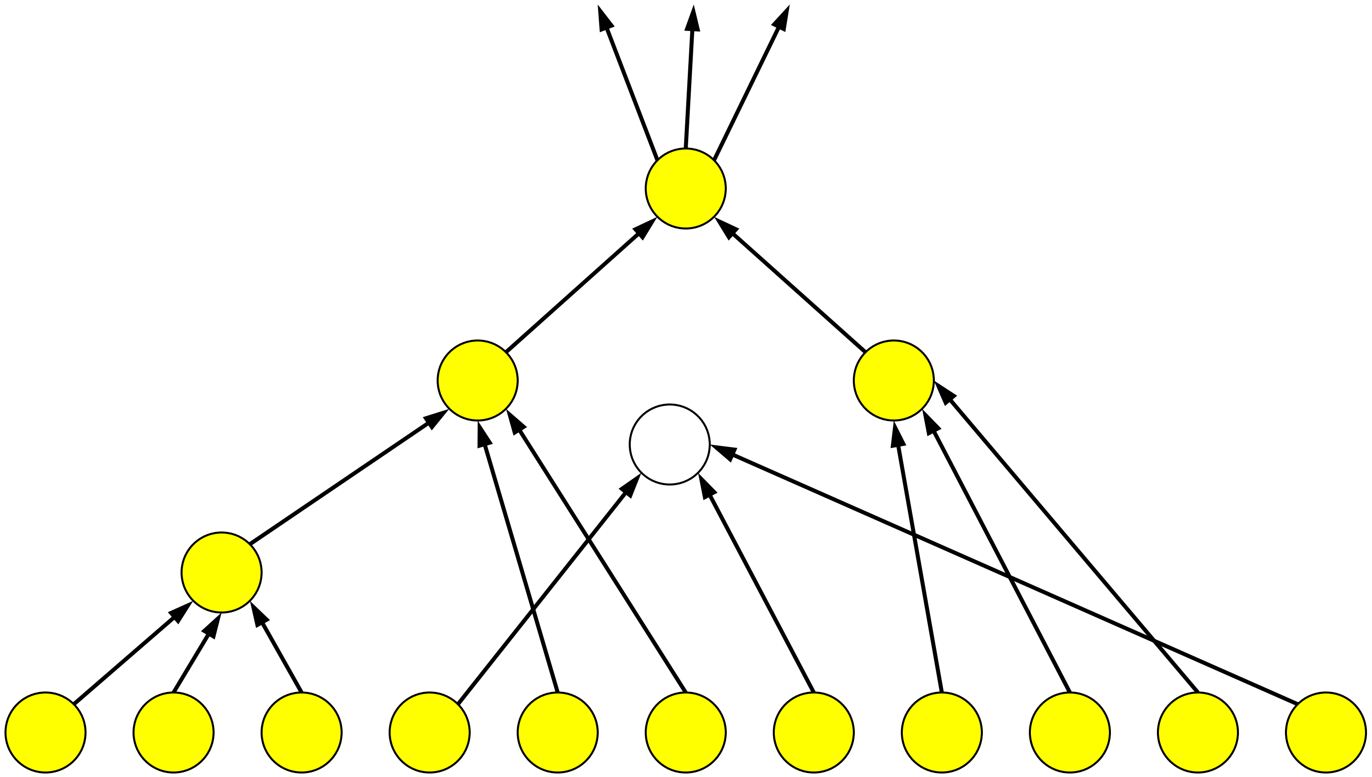
Why a general DAG?

- “Uniform” stages aren’t really uniform



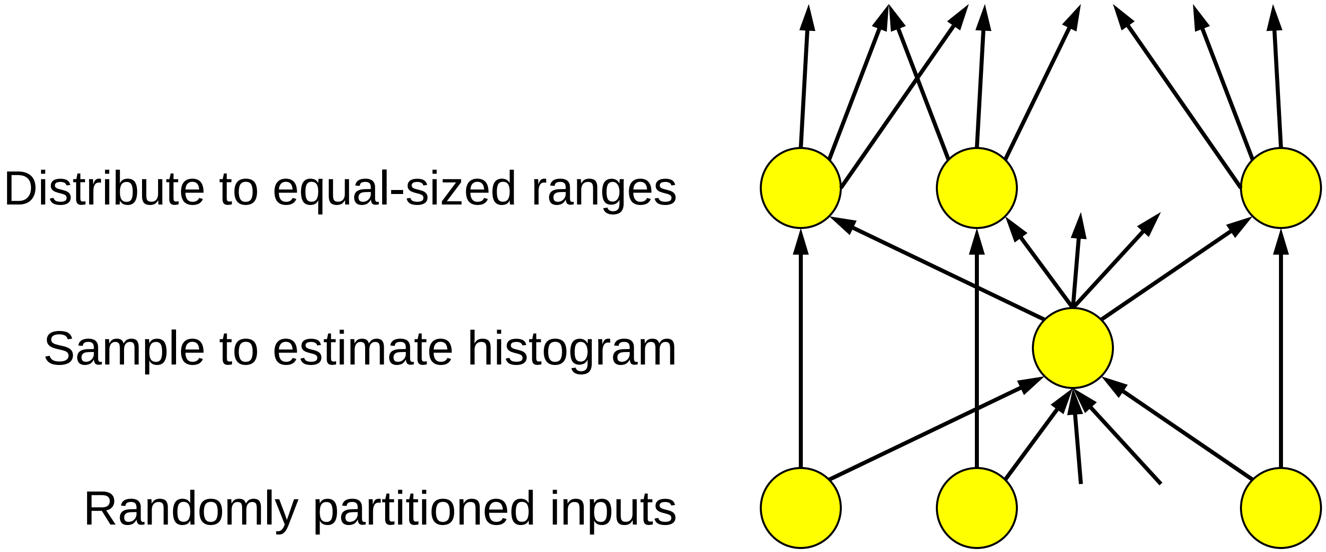
Why a general DAG?

- “Uniform” stages aren’t really uniform



Graph complexity composes

- Non-trees common
- E.g. data-dependent re-partitioning
  - Combine this with merge trees etc.



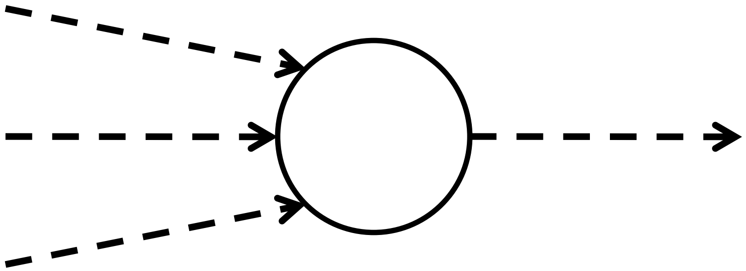
Distribute to equal-sized ranges

Sample to estimate histogram

Randomly partitioned inputs

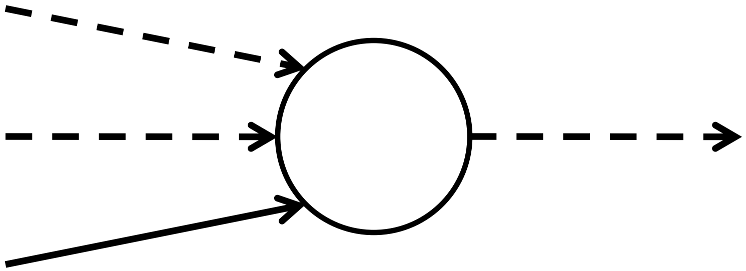
Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.



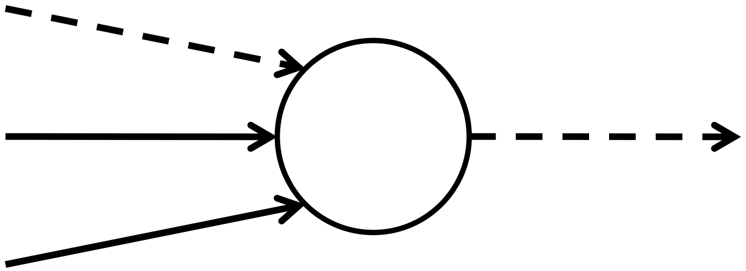
Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.



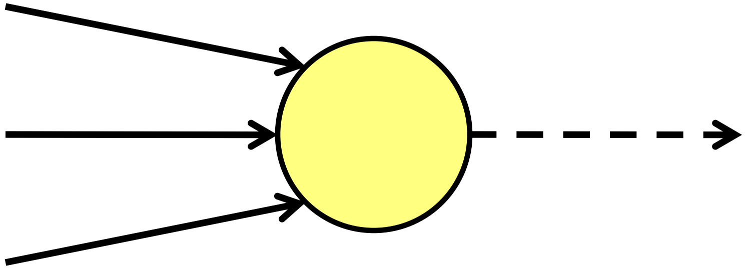
Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.



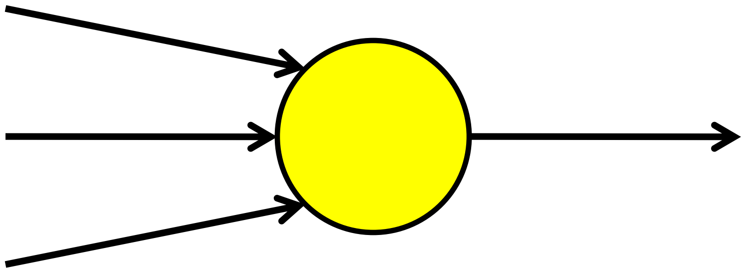
Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.



Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.



Why no cycles?

- Scheduling is easy
  - Vertex can run anywhere once all its inputs are ready.
  - Directed-acyclic means there is no deadlock
  - Finite-length channels means vertices finish.
- Fault tolerance is easy (with deterministic code)

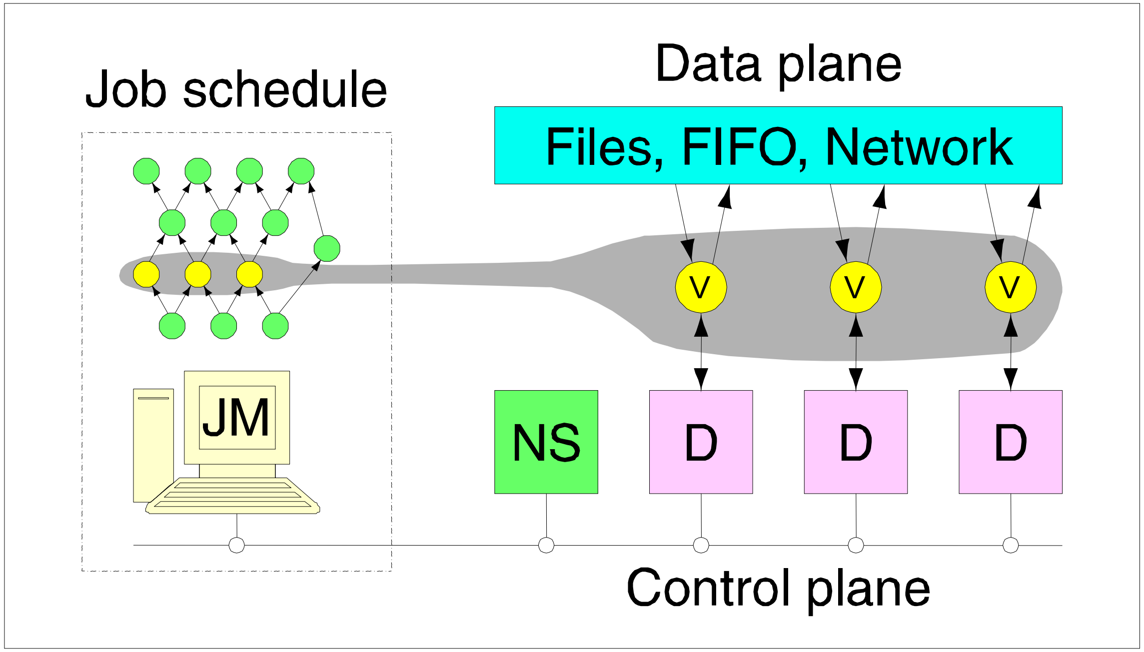
Optimizing Dryad applications

- General-purpose refinement rules
- Processes formed from subgraphs
  - Re-arrange computations, change I/O type
- ***Application code not modified***
  - System at liberty to make optimization choices
- High-level front ends hide this from user
  - SQL query planner, etc.

Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

Runtime



- Services
  - Name server
  - Daemon
- Job Manager
  - Centralized coordinating process
  - User application to construct graph
  - Linked with Dryad libraries for scheduling vertices
- Vertex executable
  - Dryad libraries to communicate with JM
  - User application sees channels in/out
  - Arbitrary application code, can use local FS

Scheduler state machine

- Scheduling is independent of semantics
  - Vertex can run anywhere once all its inputs are ready
    - Constraints/hints place it near its inputs
  - Fault tolerance
    - If A fails, run it again
    - If A’s inputs are gone, run upstream vertices again (recursively)
    - If A is slow, run another copy elsewhere and use output from whichever finishes first

Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

SkyServer DB Query

- 3-way join to find gravitational lens effect
- Table U: (objId, color) 11.8GB
- Table N: (objId, neighborId) 41.8GB
- Find neighboring stars with similar colors:
  - Join U+N to find

    T = U.color,N.neighborId where U.objId = N.objId
  - Join U+T to find

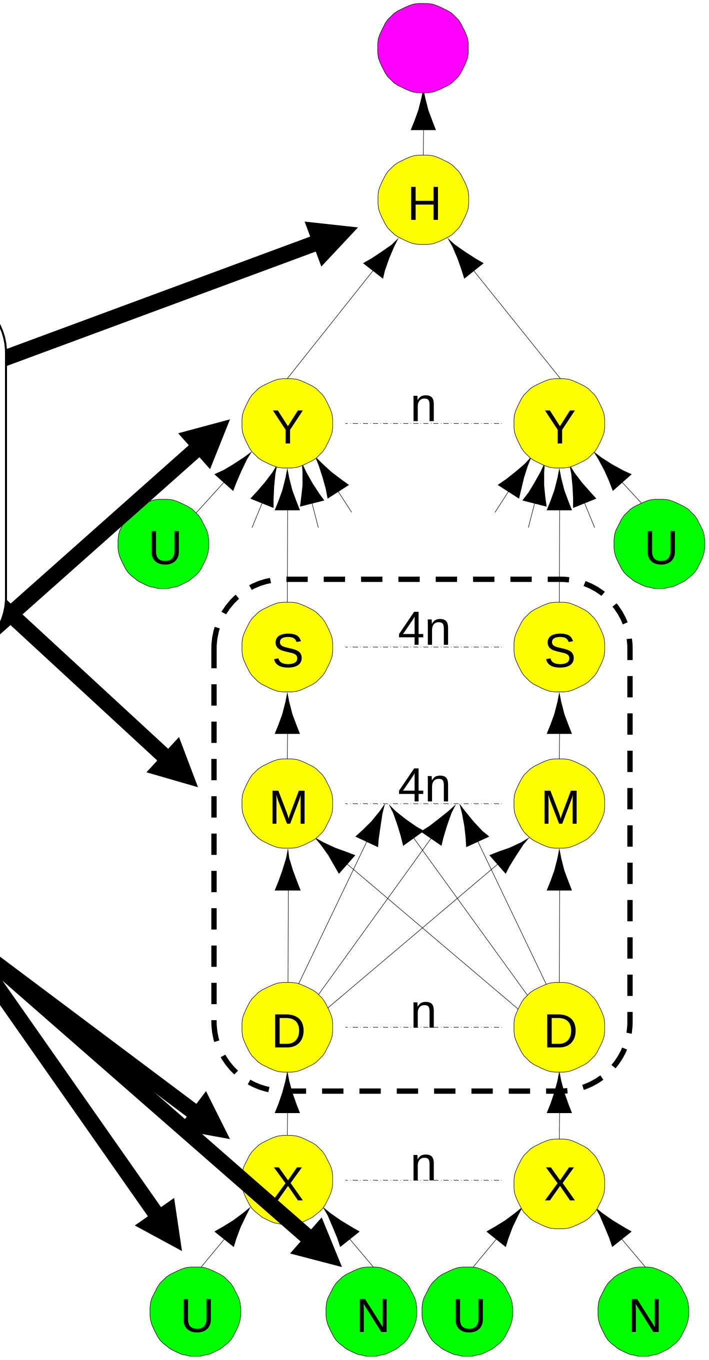
    U.objId where U.objId = T.neighborID

    and U.color ≈ T.color

SkyServer DB query

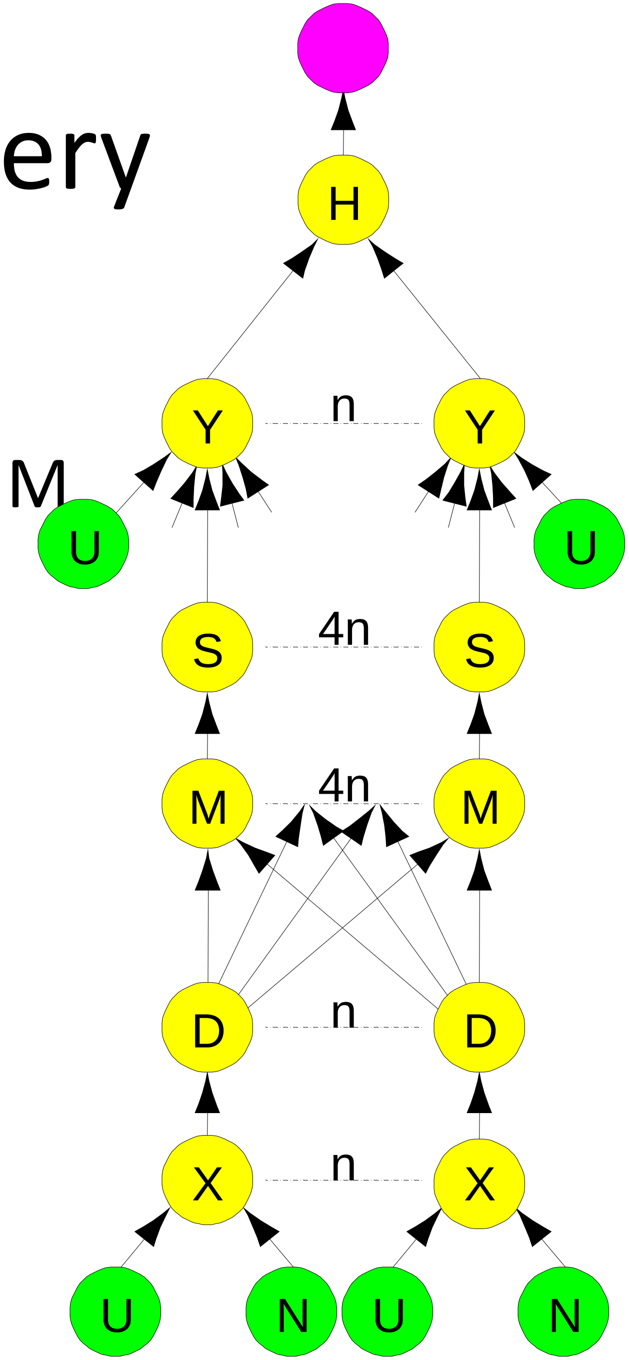
[distinct]
[merge outputs]

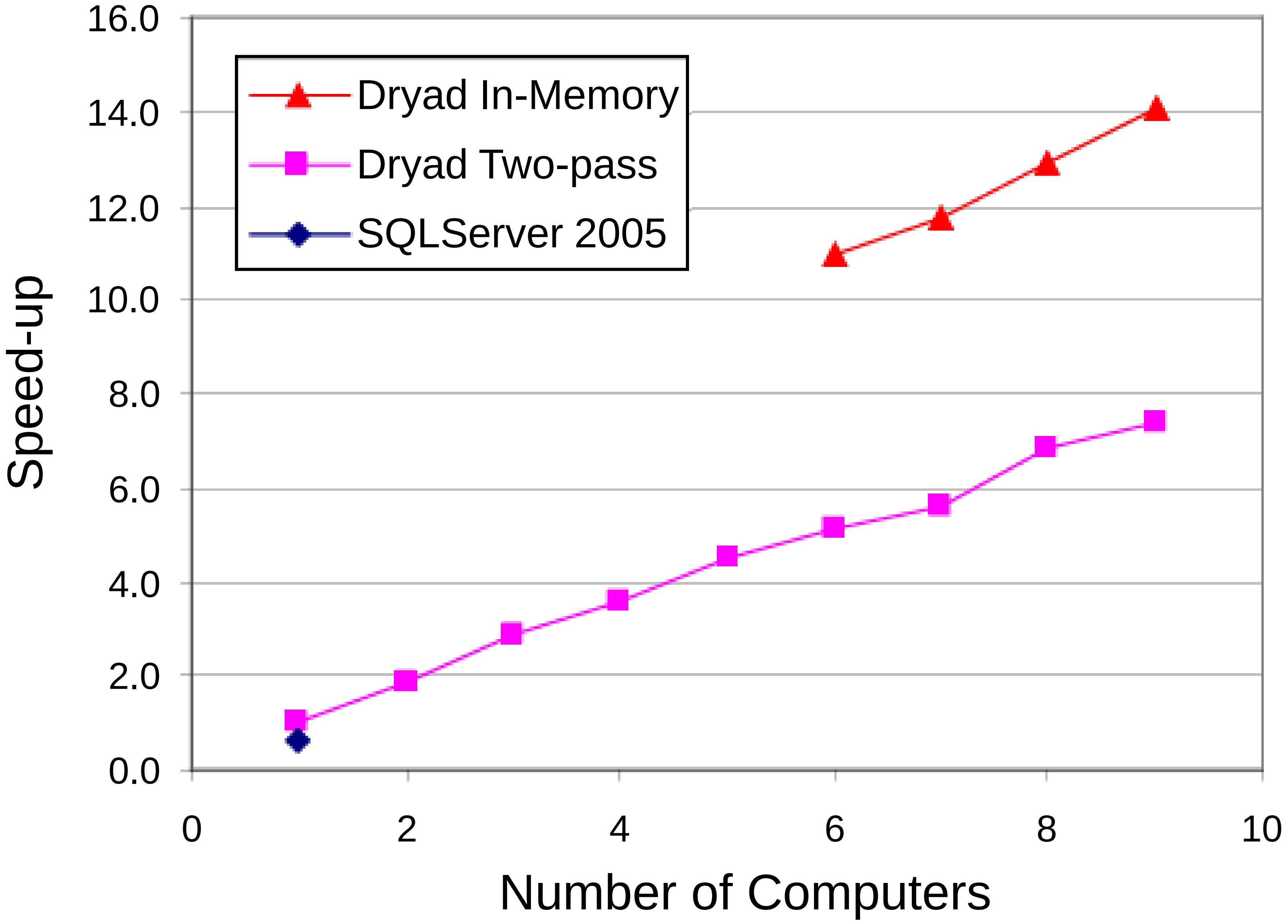
select
u.objid
from u join <temp>
where
u.objid = <temp>.neighborobjid and
|u.color - <temp>.color| < d



SkyServer DB query

- M-S-Y : SHM
  - “in-memory” : D-M is TCP and SHM
  - “2-pass” : D-M is Temp Files.
- Other Edges:
  - Temp Files

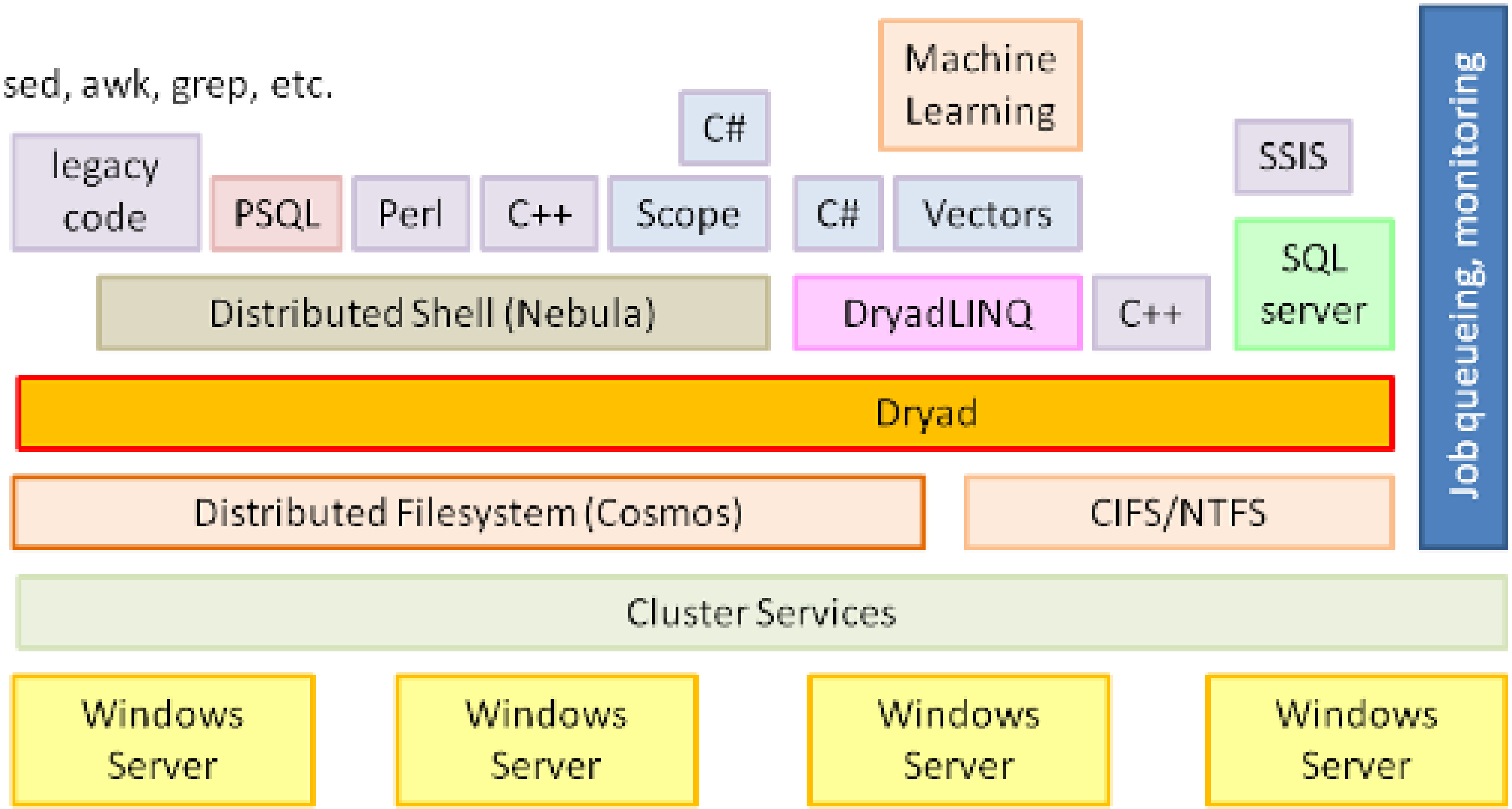



16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
Speed-up
0
2
4
6
8
10
Number of Computers
Dryad In-Memory
Dryad Two-pass
SQLServer 2005

Outline

- Map Reduce
- Dryad
  - Computational Model
  - Architecture
  - Use cases
  - DryadLINQ

Dryad Software Stack



DryadLINQ

- LINQ: Relational queries integrated in C#
- More general than distributed SQL
  - Inherits flexible C# type system and libraries
  - Data-clustering, EM, …

LINQ

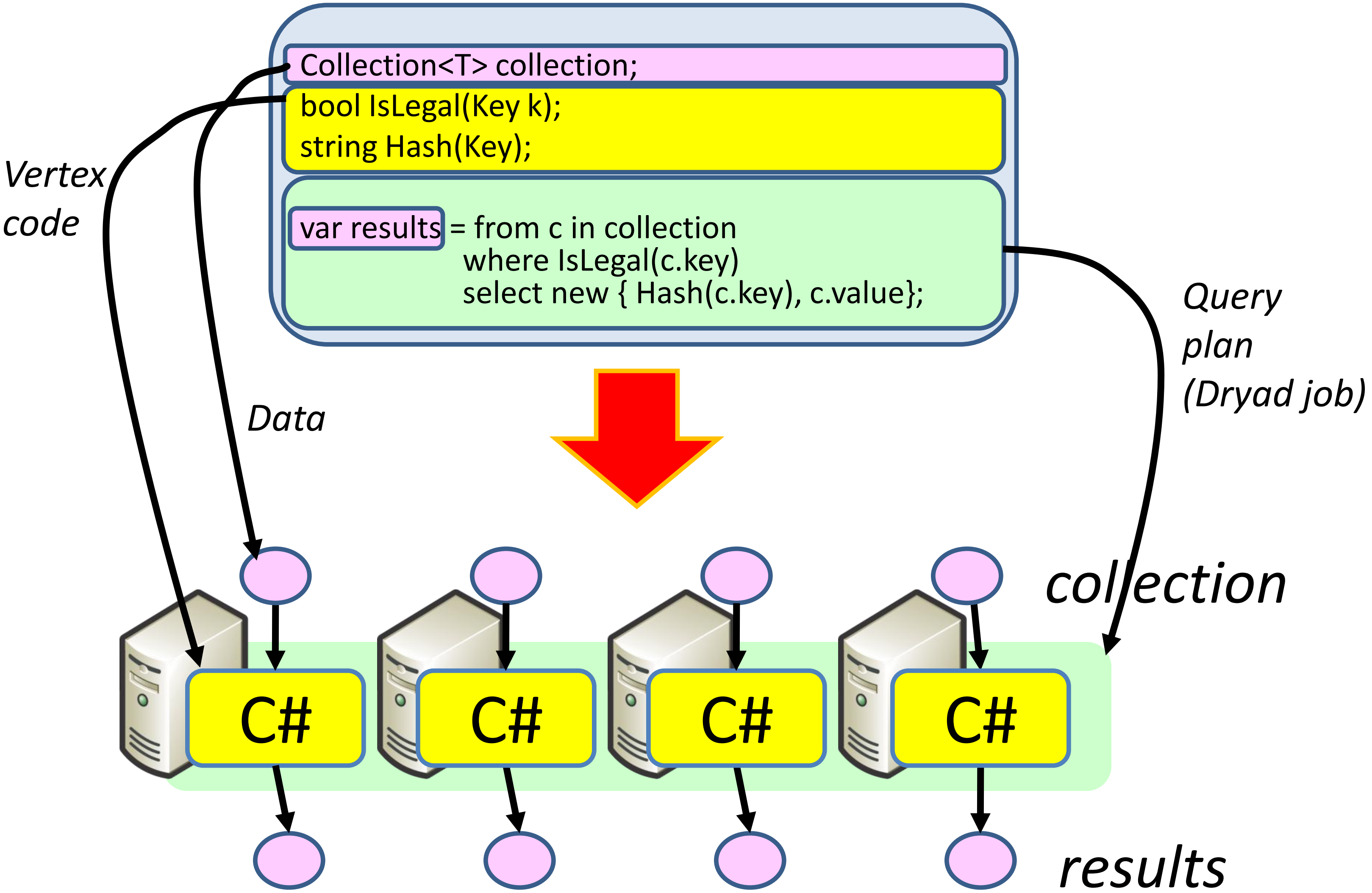
```
Collection<T> collection;
```

```
bool IsLegal(Key);

string Hash(Key);
```

```
var results = from c in collection
            where IsLegal(c.key)
            select new { Hash(c.key), c.value};
```

DryadLINQ = LINQ + Dryad



```
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

Vertex code

Data

Query plan (Dryad job)

collection

results

Performance

- 10% code.(In comparison to programming directly on the Dryad middleware)
- 30% slower than “expert code”.

Summary

- General-purpose platform for scalable distributed data-processing of all sorts
- Very flexible
  - Optimizations can get more sophisticated
- Designed to be used as middleware
  - Slot different programming models on top
  - LINQ is very powerful

Yahoo! Cloud Serving Benchmark

Xiaowei

Motivation

HYPERTABLE

SQL Azure

Redis

PNUTS

HBASE

cassandra

Project Voldemort
A distributed database

amazon web services™

riak

mongoDB
{name: "mongo", type:"DB"}

apache CouchDB relax

Benchmark tiers

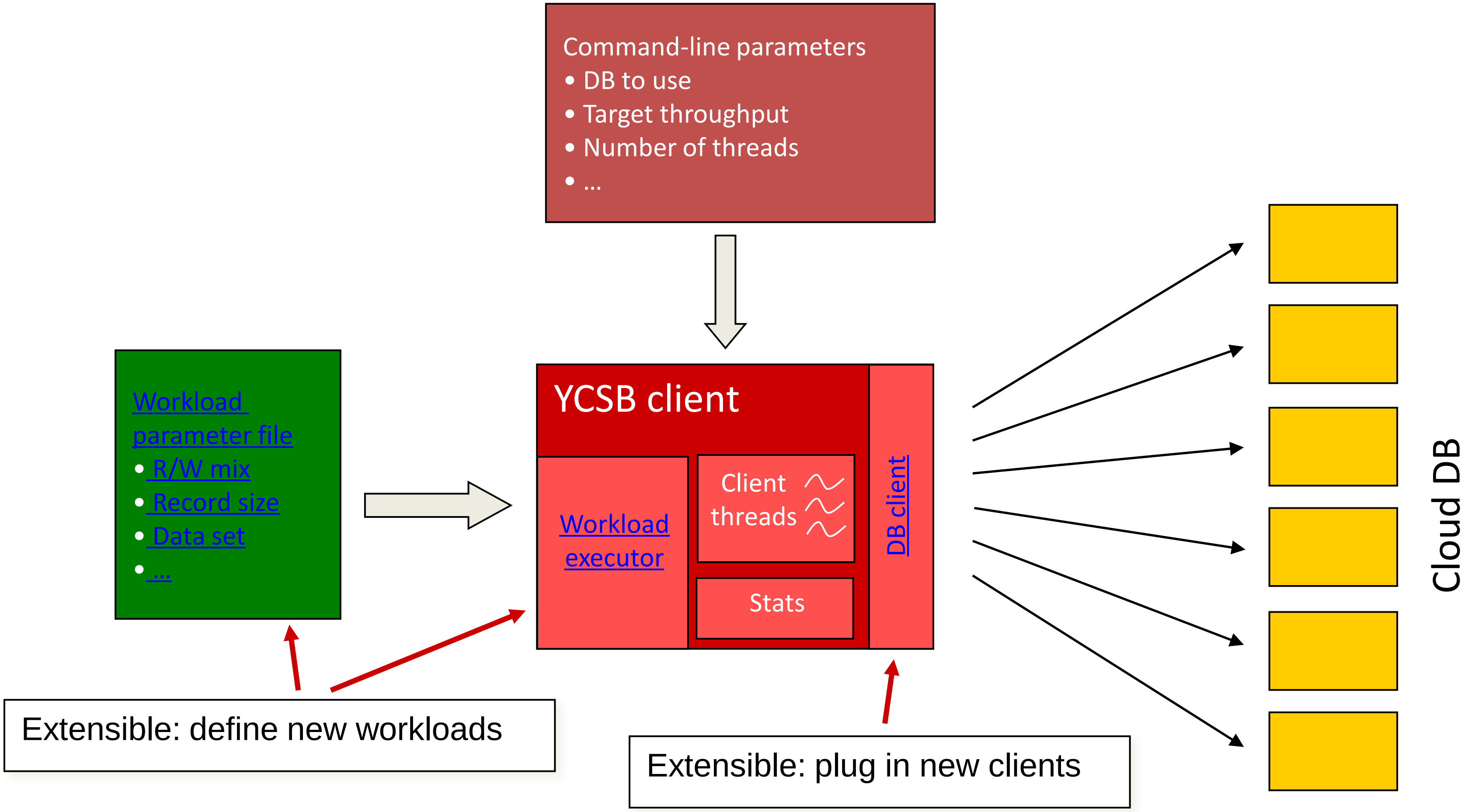
- Tier 1 – Performance
  - A system with better performance will achieve the desired latency and throughput with fewer servers
- Tier 2 – Scalability
  - Latency as database, system size increases
  - “Scaleup”

  - Latency as we elastically add servers
  - “Elastic speedup”

Benchmark tiers

- Tier 3 – Availability
  - Measure the Impact of failures on the system
- Tier 4 – Replication
  - Measure the effects of Replication Strategy on the system’s performance

Architecture



Command-line parameters
- DB to use
- Target throughput
- Number of threads
- …

Workload parameter file
- R/W mix
- Record size
- Data set
- …

YCSB client

Workload executor

Client threads

Stats

DB client

Cloud DB

Extensible: define new workloads

Extensible: plug in new clients

DB interface

- read()
- insert()
- update()
- delete()
- scan()
  - Execute range scan, reading specified number of records starting at a given record key

Test

- Setup
  - Six server-class machines
    - 8 cores (2 x quadcore) 2.5 GHz CPUs, 8 GB RAM, 6 x 146GB 15K RPM SAS drives in RAID 1+0, Gigabit ethernet, RHEL 4
  - Plus extra machines for clients, routers, controllers, etc.
  - Cassandra 0.5.0 (0.6.0-beta2 for range queries)
  - HBase 0.20.3
  - MySQL 5.1.32 organized into a sharded configuration
  - PNUTS/Sherpa 1.8 with MySQL 5.1.24
  - No replication; force updates to disk (except HBase, which primarily commits to memory)

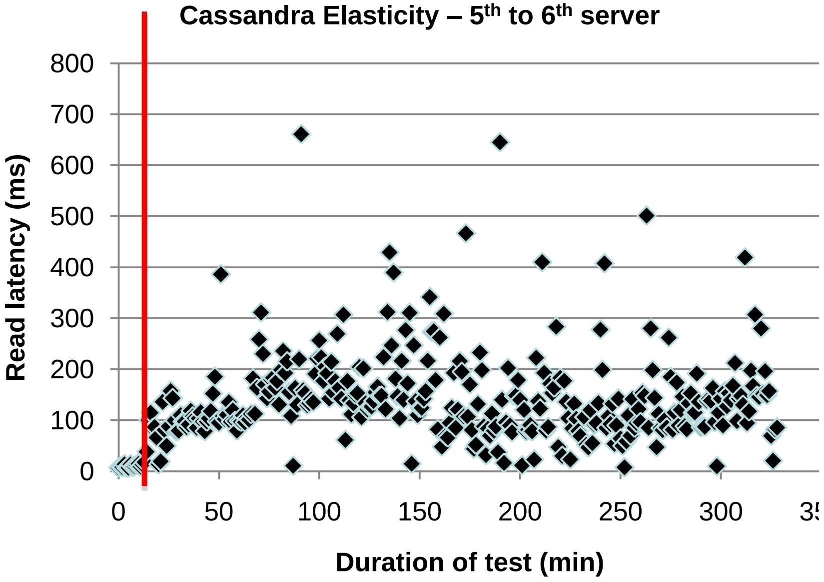
- Workloads
  - 120 million 1 KB records = 20 GB per server

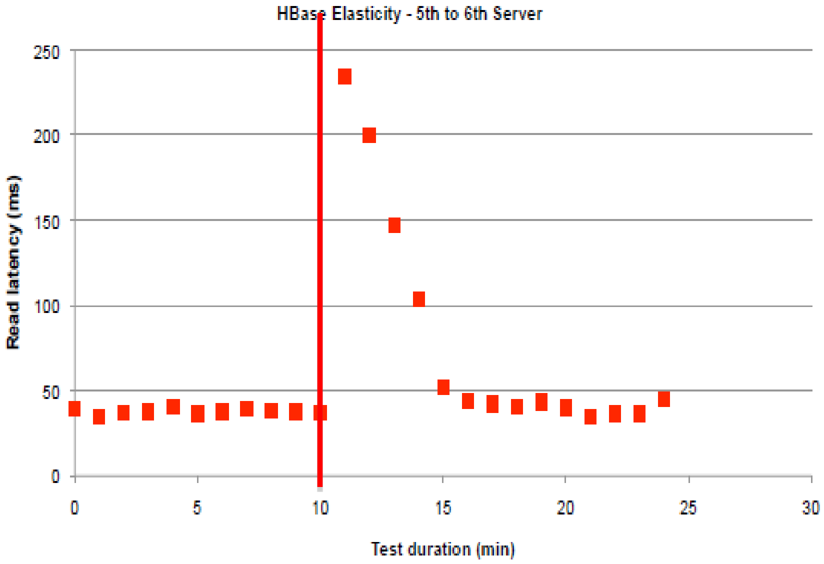
- Caveat
  - We tuned each system as well as we knew how, with assistance from the teams of developers

https://github.com/brianfrankcooper/YCSB/tree/master/workloads

Elasticity

- Run a read-heavy workload


Cassandra Elasticity – 5th to 6th server


HBase Elasticity - 5th to 6th Server

Running a workload

- Set up the database system to test
- Choose the appropriate DB interface layer
- Choose the appropriate workload
- Choose the appropriate runtime parameters (number of client threads, target throughput, etc.)
- Load the data
- Execute the workload

Tips

- Only one Tip!

Conclusions

- YCSB is an opensource benchmark for cloud serving systems

- Experimental results show tradeoffs between systems
- https://github.com/brianfrankcooper/YCSB/wiki/
- http://arunxjacob.blogspot.com/2011/03/setting-up-ycsb-for-low-latency-data.html

Thanks!