Distributed Programming

Rong Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/rong_chen

Credits: some slides from P. Krzyzanowski (Rutgers) and Michael Isard (MSR) et al.

MapReduce: Simplified Data Processing on Large Clusters

Background

Traditional programming is **serial**

Parallel programming

- Break processing into parts that can be executed **concurrently** on multiple processors

Challenges

- Identify **concurrency**

  Task/Data *that can be processed **concurrently***

- Not all problems can be parallelized

Simplest Env. for Parallelism

- **No dependency** among data
- Data can be split into **equal-size** chunks
- **Each** process can work on a chunk
- **Master/worker** approach
  - Master
    1. Initialize array and splits it according to # of workers
    2. Send each worker the sub-array
    3. Receive the results from each worker
  - Worker
    1. Receive a sub-array from master
    2. Perform processing
    3. Send results to master

Why distributed computations?

Multiprocessor system don’t scale enough

Disney’s Cars 2 required 11.5 hours to render each frame (average) – some took 90 hours

- 12,500 cores on Dell render blades
- ~152,640 frames = 1,755,360 hours = 200ys

Google

- Approximately 1 billion queries per day
- Index > 25 billion web pages
- Hundreds of thousands of servers

Distributed Computations

Cluster computing

- Hundreds or thousands of PCs connected by high speed LANs

Grid computing

- Hundreds of supercomputers connected by high speed network

. . .

1000 nodes potentially give 1000X speedup

Challenges

Sending **data** to/from nodes

Coordinating among nodes

Recovering from node **failure**

Optimizing for **locality**

Debugging

Same for
all problems

MapReduce Programming Model

Created by **Google** (OSDI'04)

- Jeffrey Dean and Sanjay Ghemawat

Inspired by **LISP** (function language)

- **Map** (function, set of values)
  - Applies function to each value in the set

```
(map 'length' (() (a) (a b) (a b c))) → (0 1 2 3)
```

- **Reduce** (function, set of values)
  - Combines all the values using a function (e.g., +)

```
(reduce #'+' (1 2 3 4 5)) → 15
```

MapReduce Programming Model

Allows user to process *huge* amounts of data on *thousands* of nodes

Framework for data-parallel computing

Programmers get **simple** API

Don't have to **worry** about handling

- ~~Parallelization~~
- ~~Data distribution~~
- ~~Load balancing~~
- ~~Fault tolerance~~

MapReduce Programming Model

user

Functionality

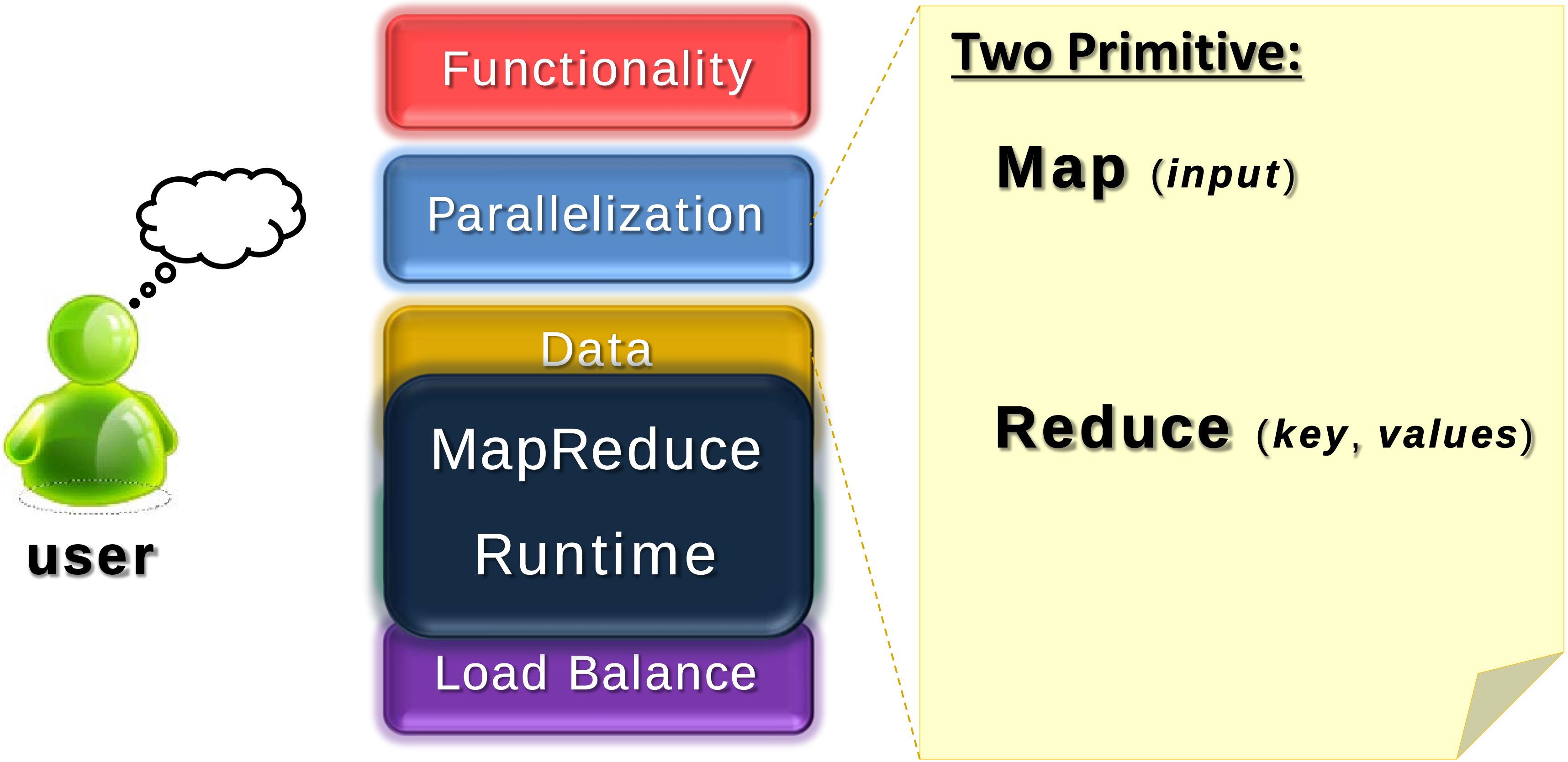
Parallelization

Data Distribution

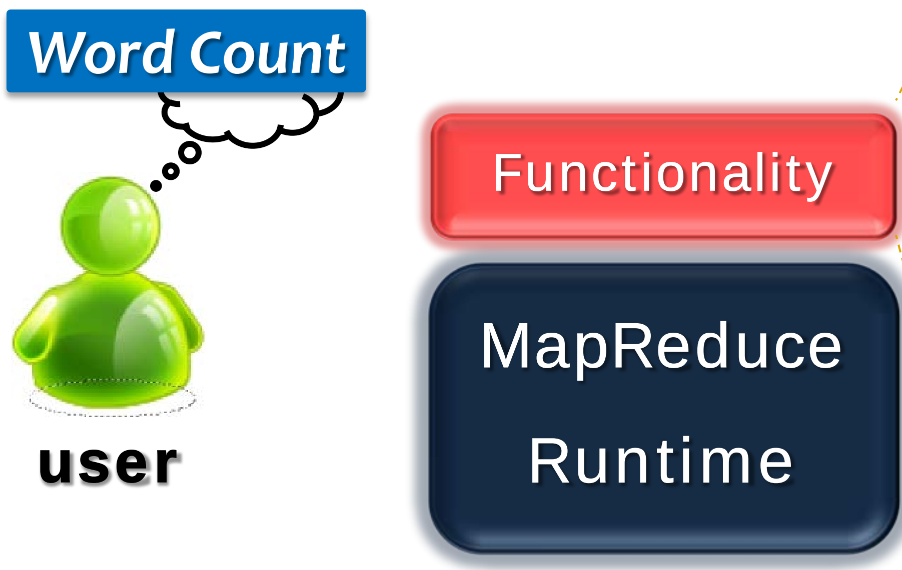
Fault Tolerance

Load Balance

MapReduce Programming Model



MapReduce Programming Model



<u>Two Primitive:</u>

Map (***input***)

for each ***word*** in ***input***

emit (***word***, ***1***)

Reduce (***key***, ***values***)

int ***sum*** = 0;

for each ***value*** in ***values***

sum += ***value***;

emit (***word***, ***sum***)

Programming Interface

Map: (input shard) → **intermediate** (k/v pairs)

- Partition the input data into M "shards"
- Group all intermediate values associated with the same key
- Pass them to the *Reduce* function

Reduce: **intermediate** (k/v pairs) → (results)

- Partition the key space into R "pieces" using a *Partition* function (e.g., hash(key) mod R)
- Accept a key & a set of values
- Merge these values to form result of the key

Execution Flow

Map

Grab the relevant data from the source
User function gets called for each chunk of input

Map Worker

Reduce Worker

Reduce

Aggregate the results
User function gets called for each unique key

Execution Flow

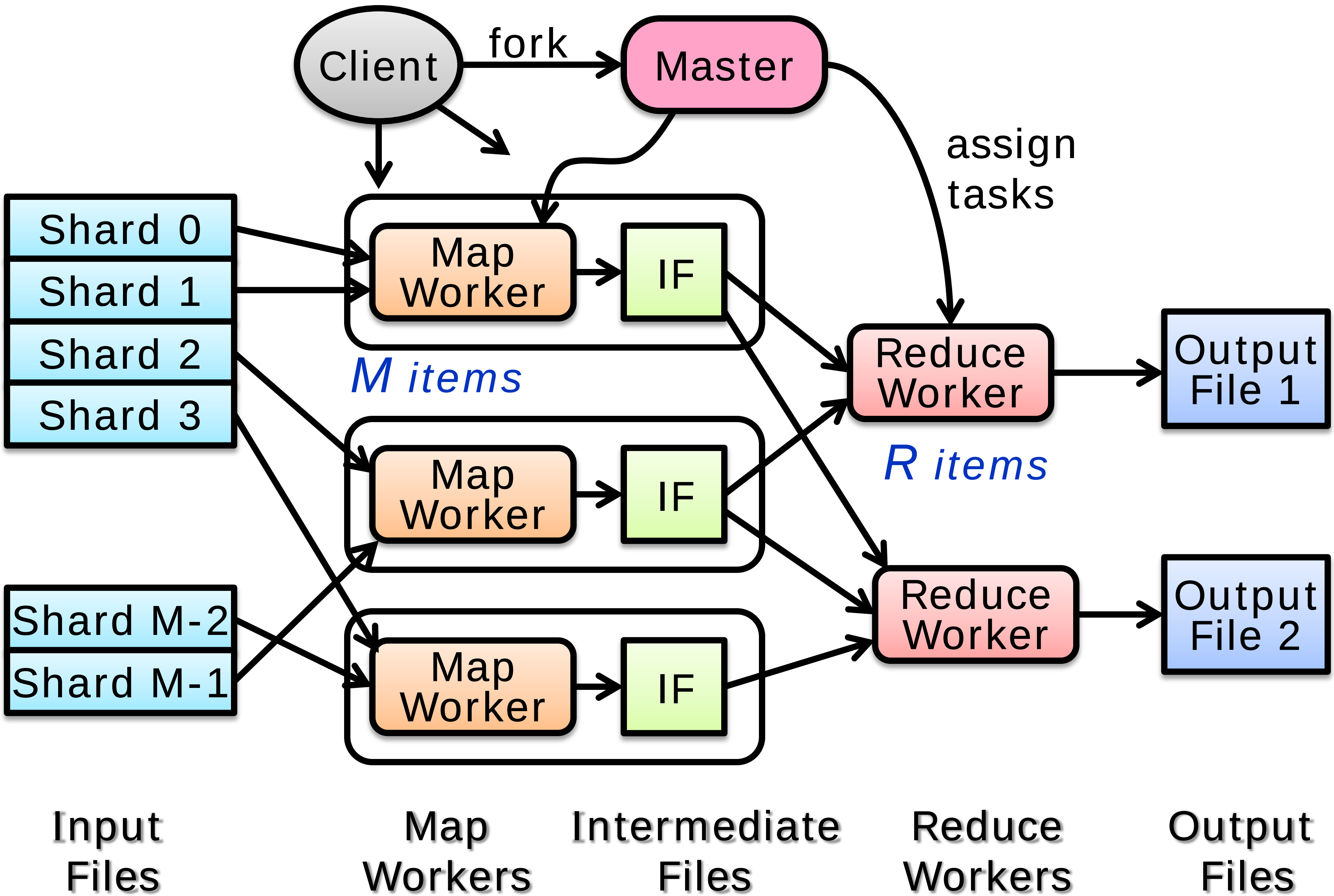
Map Worker

Map	Grab the relevant data from the source User function gets called for each chunk of input
Partition	Identify which of Reducers will handle which keys Map partitions data to target Reducers

Reduce Worker

Sort	Fetch the relevant partition data from all mappers Sort by keys
Reduce	Aggregate the results User function gets called for each unique key

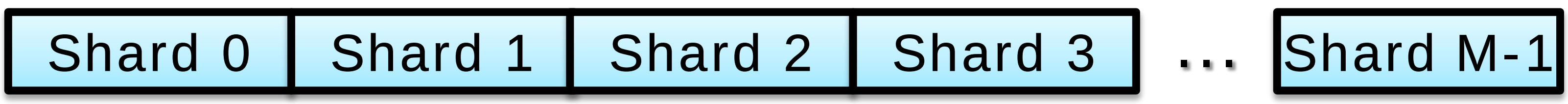
The Complete Picture



The Complete Picture

Step1: split input files into chunks (shards)

- Typically 64MB



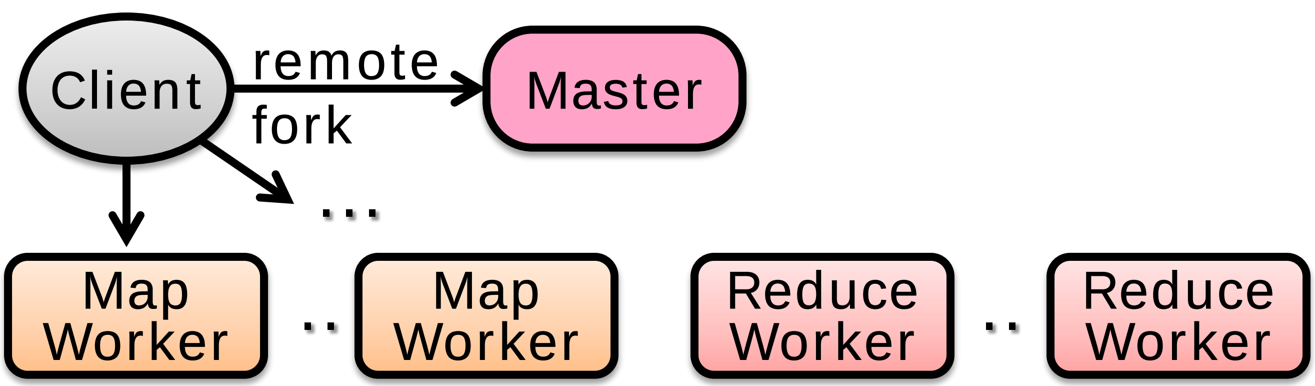
Input Files

Divided into M shards

The Complete Picture

Step2: fork processes

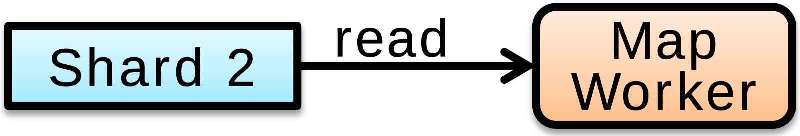
- Start up many copies of the program on cluster
  - 1 master: scheduler & coordinator
  - Lots of workers
- Idle workers are assigned either
  - Map tasks (each works on a shard)
  - Reduce tasks (each works on intermediate files)



The Complete Picture

Step3: map task

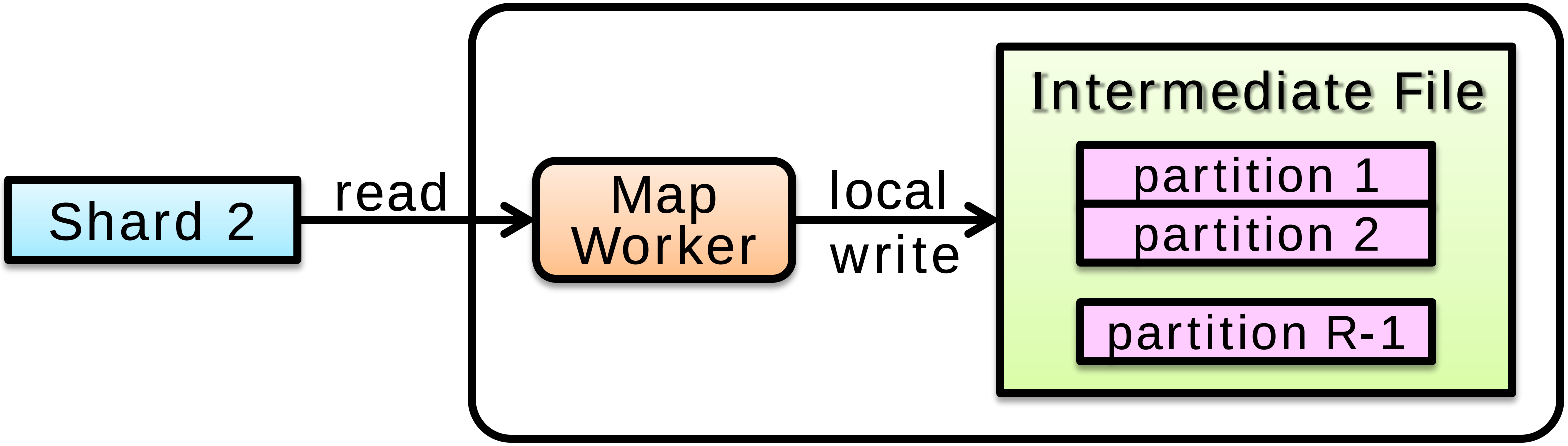
- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined *Map* function
  - Produces intermediate key/value pairs
  - These are buffered in memory



The Complete Picture

Step4: create intermediate files

- Intermediate k/v pairs buffered in memory and periodically written to the local disk
  - Partitioned into *R* regions by a *Partition* function
- Notifies master when complete
  - Passes location of intermediate data to the master
  - Master forwards locations to the Reduce worker



The Complete Picture

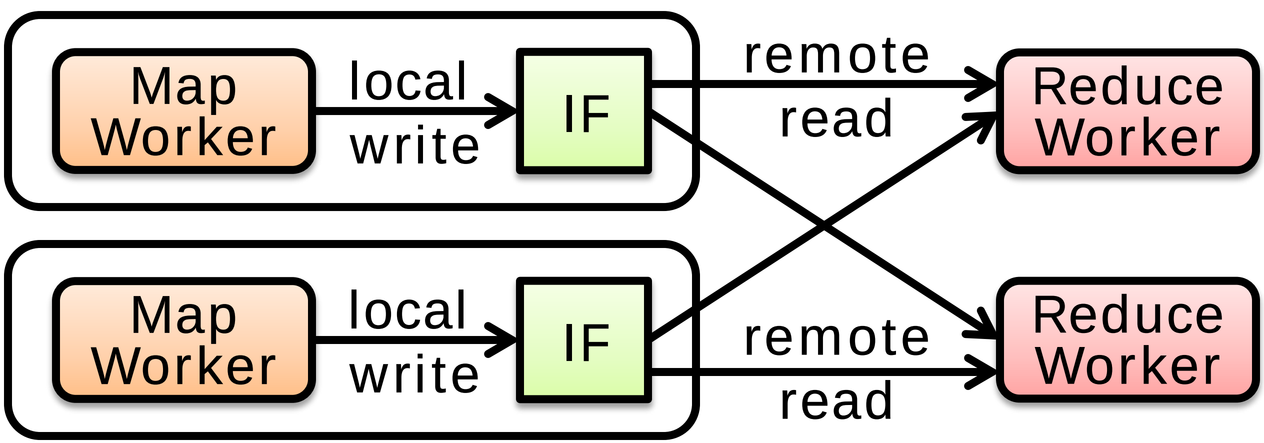
Step4a: partitioning

- Map data will be processed by reduce workers
  - The user's *Reduce* function will be called once per unique key
- Sort all the (key, value) data by keys
- *Partition* function: decides which of R reduce workers will work on which key
  - Default function: *hash(key) mod R*
  - Map worker partitions the data by keys
- Each reduce worker will read their partition from every map worker

The Complete Picture

Step5: sorting intermediate data

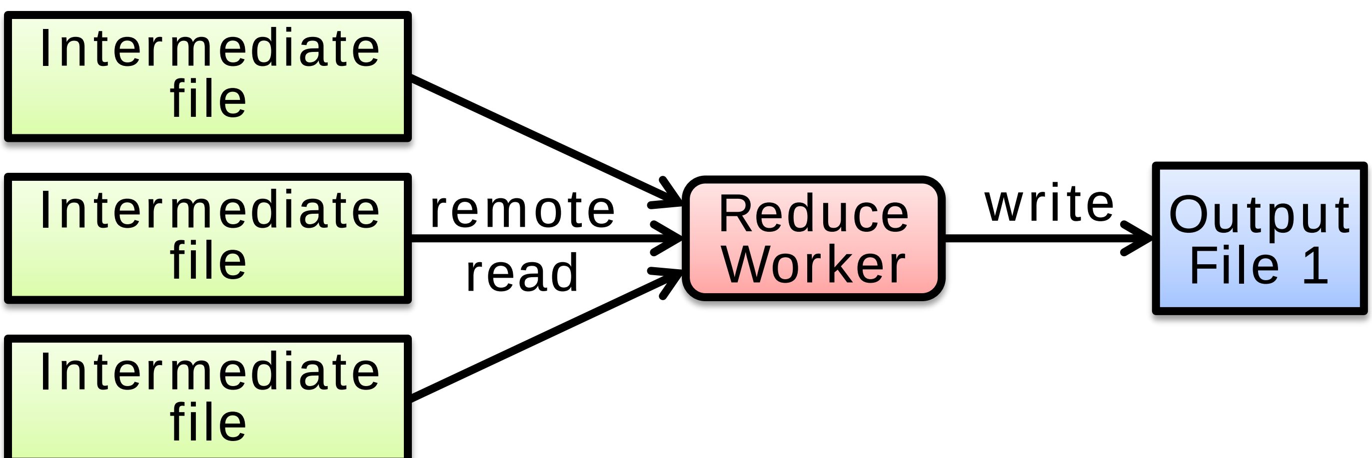
- Notified by master about the location of intermediate files for its partition
- Uses RPCs to read the data from the local disks of map workers
- Sorts the data by the intermediate keys
- All occurrences of the same key are grouped



The Complete Picture

Step6: reduce task

- Groups data with a unique intermediate key
- User's *Reduce* function is given the key and the set of intermediate values for that key
  - <key, (value1, value2, ...)>
- The output is appended to an output file



The Complete Picture

Step7: return to user

- When all map and reduce tasks have completed
- Master wakes up the user program
- The *MapReduce* call in the user program returns and the program can resume execution
  - Output of *MapReduce* is available in R output files

Example: Word Count

Word Count: Count # occurrences of each word in a collection of documents

Map:

- Parse data and emit each word with a count (1)

Reduce:

- Sort: sort by keys (words)
- Reduce: sum together counts each key (words)

```
map (String key, String value)
    for each word w in value
        EmitIntermediate (w, "1");
```

```
Reduce (String key, Iterator values)
    int result = 0;
    for each v in values
        result += ParseInt (v);
    emit (AsString (result));
```

Example: Word Count

Input file

It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and streetstalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously ...

Map

After Map

[Intermediate]

...
it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking 1
burrower 1
and 1
grub-worm 1
of 1
a 1
poor 1
devil 1
of 1
a 1
sub-sub 1
...

Reduce

After Sort

[In-memory]

...
a 1
a 1
aback 1
aback 1
abaft 1
abaft 1
abandon 1
abandon 1
abandon 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abased 1
abased 1
...

After Reduce

a 4736
aback 2
abaft 2
abandon 3
abandoned 7
abased 2
abasement 1
abashed 2
abate 1
abated 3
abatement 1
abating 2
abbreviate 1
abbreviation 1
abeam 1
abed 2
abednego 1
abel 1
abhorred 3
...

Locality

- Input and Output files are on GFS
  - Google File System (SOSP'03)
  - Each chunk is replicated on multiple servers (3)
- MapReduce runs on GFS chunkservers
  - Collocate computation and storage
- Master tries to schedule map worker on one of the nodes that has a copy of the input chunk it need

Fault Tolerance

- Master pings each worker periodically
  - If no response is received within a certain time, the worker is marked as failed
  - Map or reduce tasks given to this worker are reset back to the initial state and rescheduled for other worker (re-execution)
- Master failure
  - State is checkpointed to GFS
  - Recover master from GFS and continue

Straggler

- Slow workers (straggler) significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data slowly
  - Weird things: processor caches disabled ☹
- Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first “wins” ☺
  - Dramatically shortens job completion time

Intermediate Key/Value Pairs

Question:

- Compare save intermediate results on **mapper**'s local disk with **reducer**'s local disk

Mapper-side:

1. Pre-aggregation → save network resource
2. Map task failure → no partial data
3. Reduce task failure → just pull data again

Reducer-side:

1. Pipeline → overlap the transfer time

Paper: *Tyson Condie et al*, MapReduce Online, NSDI, 2010

Other Examples

Distributed grep

- *Search for words in lots of documents*
- **Map**: emit a line if it matches a given pattern
- **Reduce**: just output the intermediate data

Count URL access frequency

- *Find the frequency of each URL in web logs*
- **Map**: process logs of web page access;
  output <URL, 1>
- **Reduce**: add all values for the same URL

Other Examples

Inverted index

- *Find what documents contain a specific word*
- **Map**: parse document, emit <word, doc-ID>
- **Reduce**: sort the doc-ID for each word, and output <word, list(doc-ID)>

Reverse web-link graph

- *Find where page links come from*
- **Map**: output <target, source> for each link to target in a page source
- **Reduce**: concatenate all source for each target, output <target, list(source)>

Summary

Get a lot of data from input files

Map

- Parse & extract items of interest

Partition & Sort

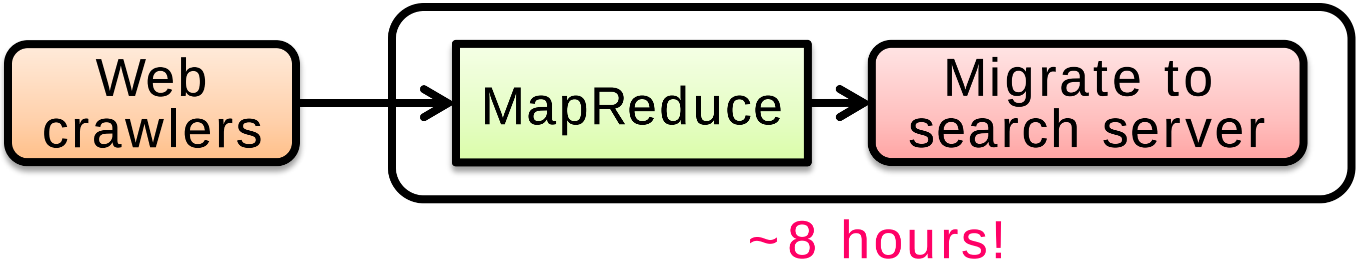
Reduce

- Aggregate results

Write results to output files

All is not Perfect

- MapReduce was used to process webpage data collected by Google's crawlers
  - It would extract the links and metadata needed to search the pages
  - Determine the site's PageRank
- The process took around eight hours
  - Results were moved to search servers
  - This was done continuously



All is not Perfect

Web has become more dynamic

- An 8+ hour delay is a lot for some sites

Goal: refresh certain pages within seconds

MapReduce

- Batch-oriented
- Not suited for near-real-time processes
- Cannot start a new phase until the previous has completed
- This was done continuously

New search framework: Caffeine

Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks

Dryad

Created by Microsoft (EuroSys'07)

- Michael Isard, Andrew Birrell, et al

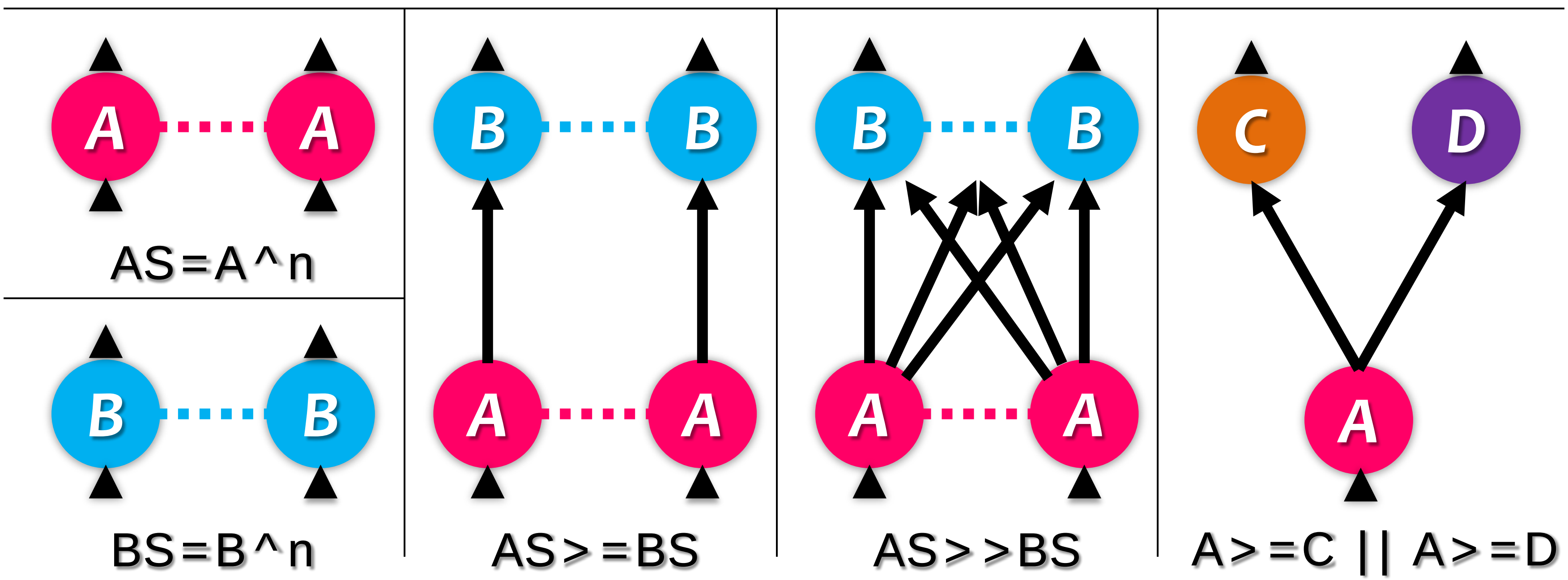
Similar goals as MapReduce

- A general-purpose distributed execution engine for coarse-grain data-parallel applications
- Focus on throughput, not latency
- Automatic management of scheduling, distribution, and fault tolerance

Dryad

Computations expressed as a **graph**

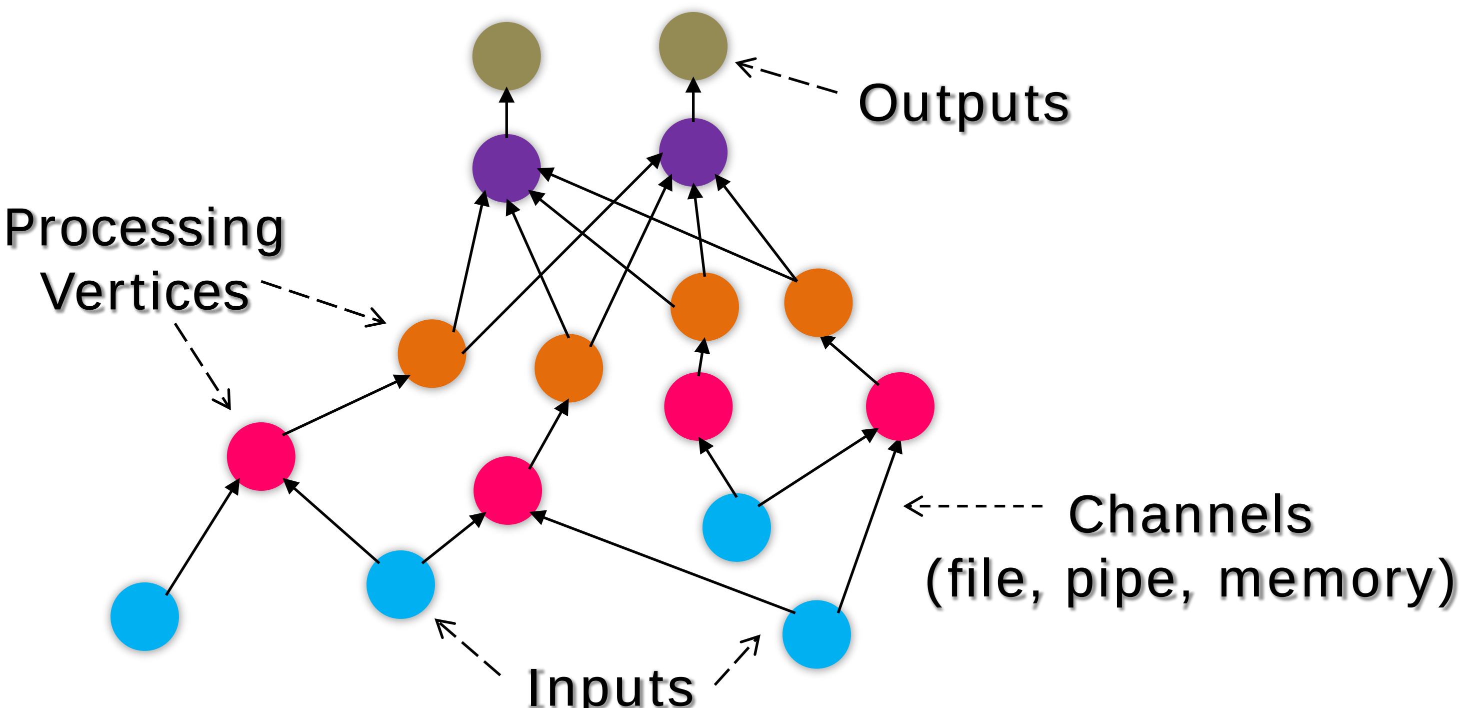
- **Vertices** are **computations**
- **Edges** are **communication** channels
- Each vertex has several **input** and **output** edges



Why using a Dataflow Graph?

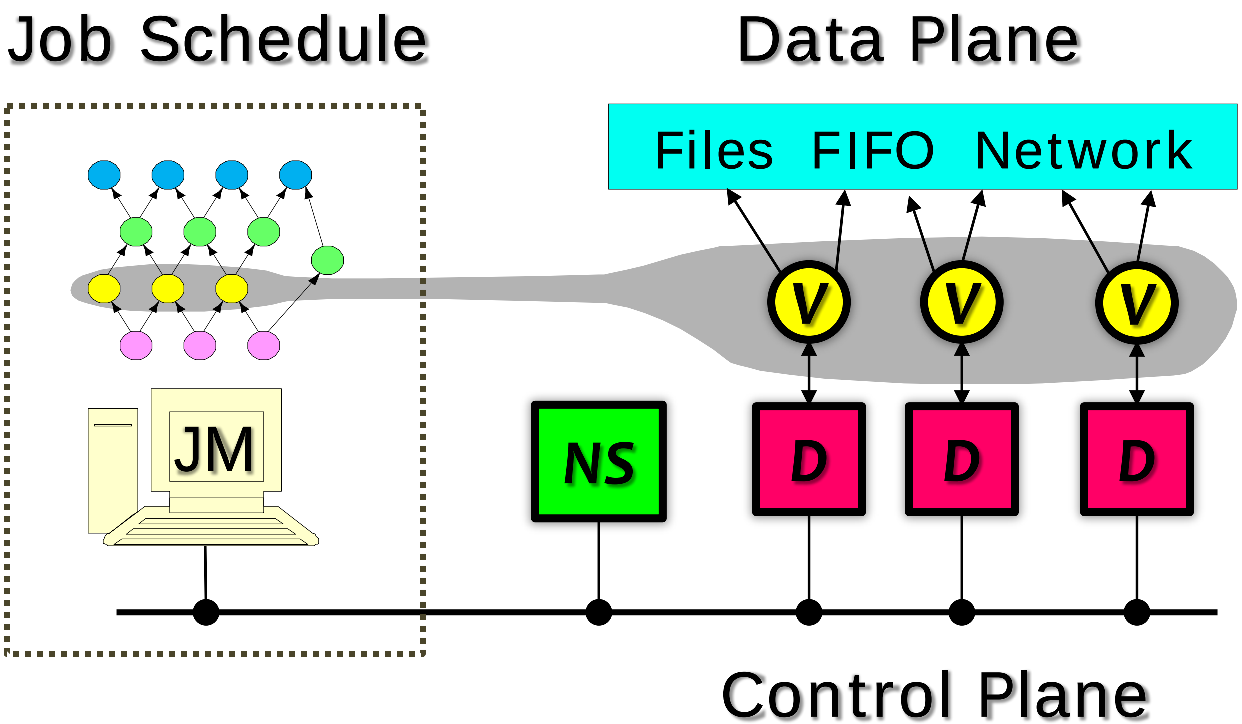
Many programs can be represented as a distributed dataflow graph

Dryad Job = Directed Acyclic Graph (DAG)



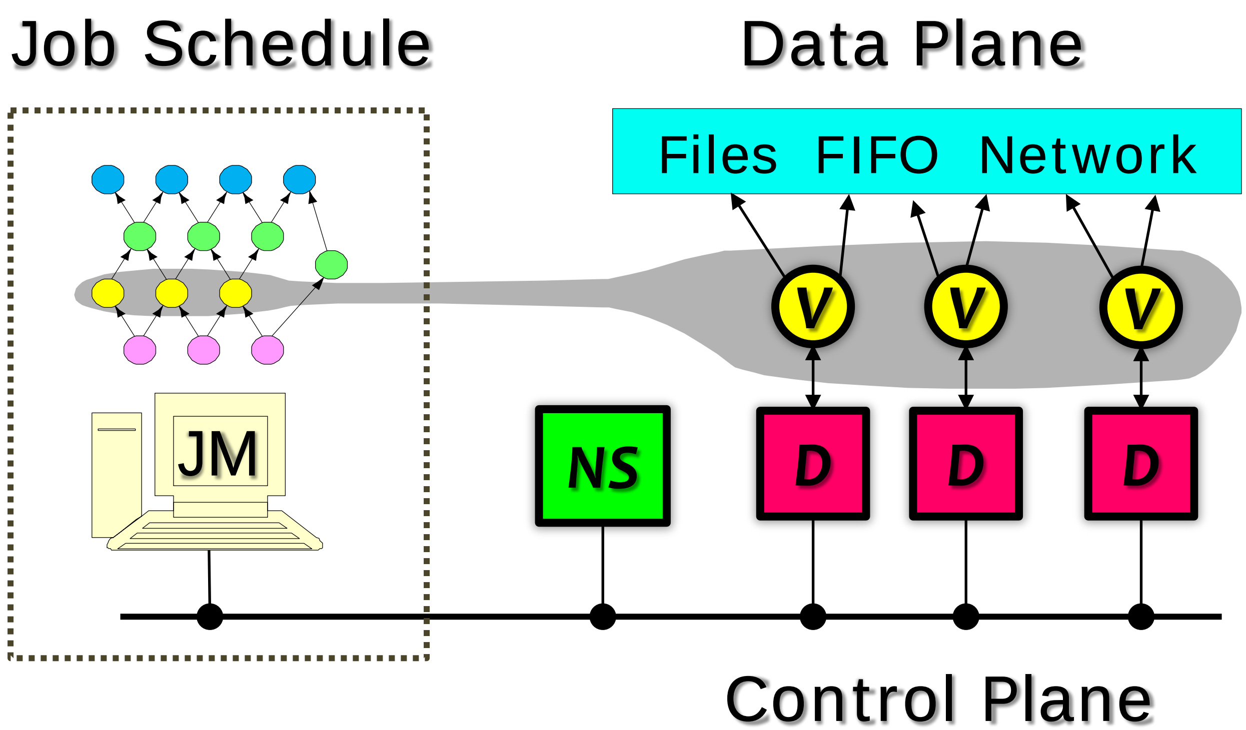
The Runtime of Dryad

Vertices (V) run arbitrary app code, **exchange** data through TCP pipes etc., and **communicate** with JM to report status



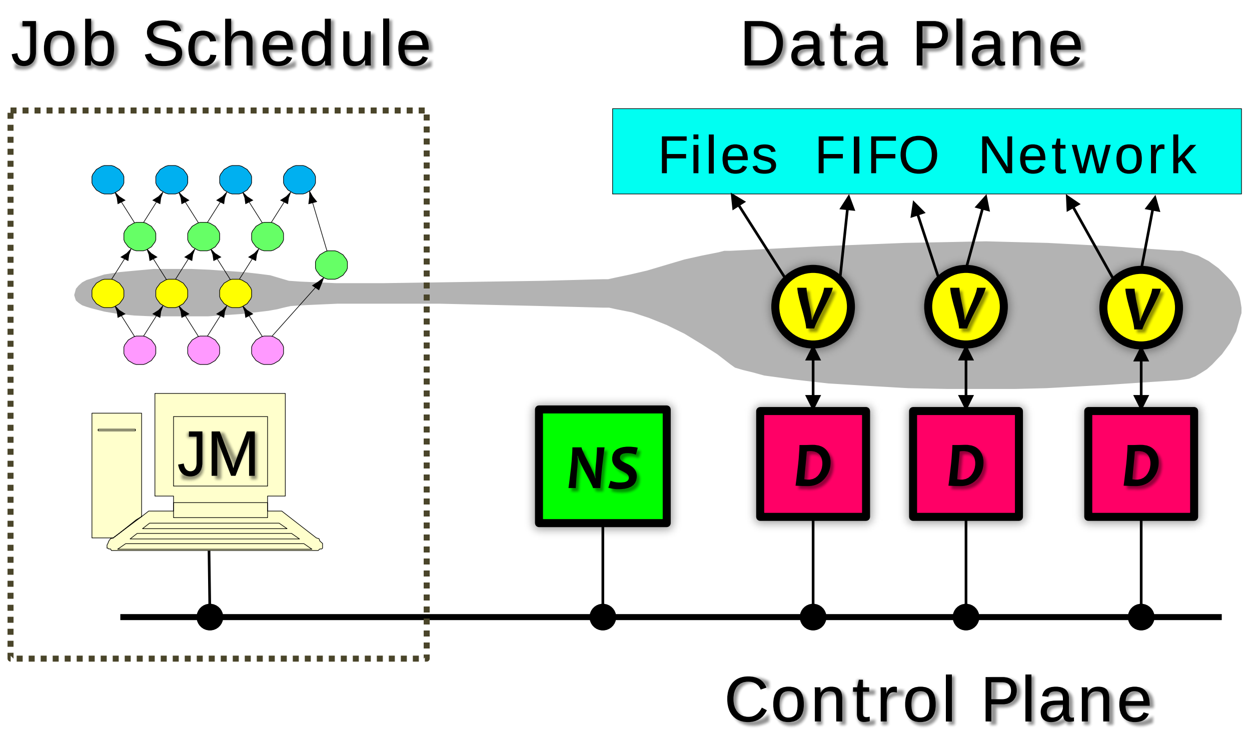
The Runtime of Dryad

Job Manager (JM) consults name server (NS) to **discover** available nodes, and **maintains** job graph and **schedules** vertices



The Runtime of Dryad

Daemon process (D) executes vertices



Scheduling in JM

General scheduling rules

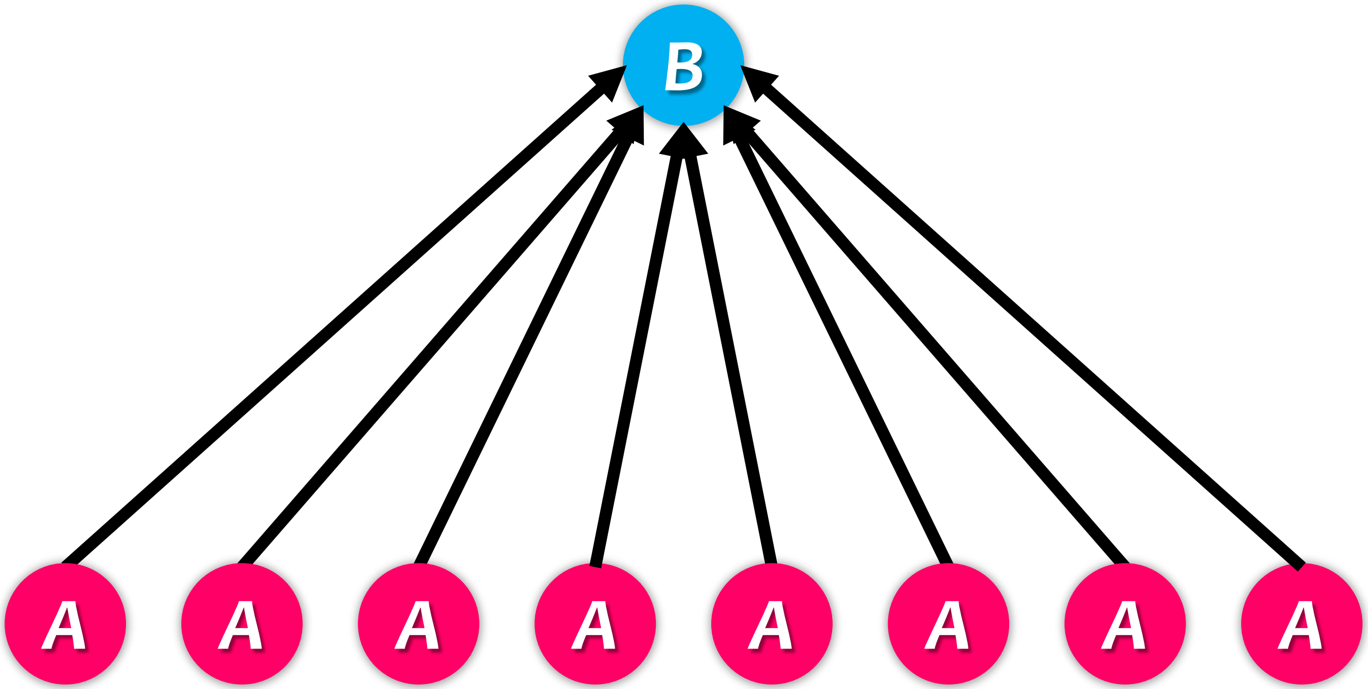
- Vertex can run anywhere once all its inputs are ready
  - Prefer executing a vertex near its inputs (locality)
- Fault tolerance
  - Vertex fails → run it again
  - Vertex's inputs are gone → run upstream vertices again (recursively)
  - Vertex is slow → run another copy elsewhere and use output from whichever finishes first

Advantages of Dryad

- Big jobs more efficient with Dryad
  - MapReduce: big job runs multiple (>=1) stages
    - Reducers of each stage wrt. to dist-storage (GFS)
    - Output of reduce → 2 network copies + 3 disks
  - Dryad: each job is represented with a DAG
    - Intermediate vertices write to local file
- Dryad provides more flexible operations
  - Join operations

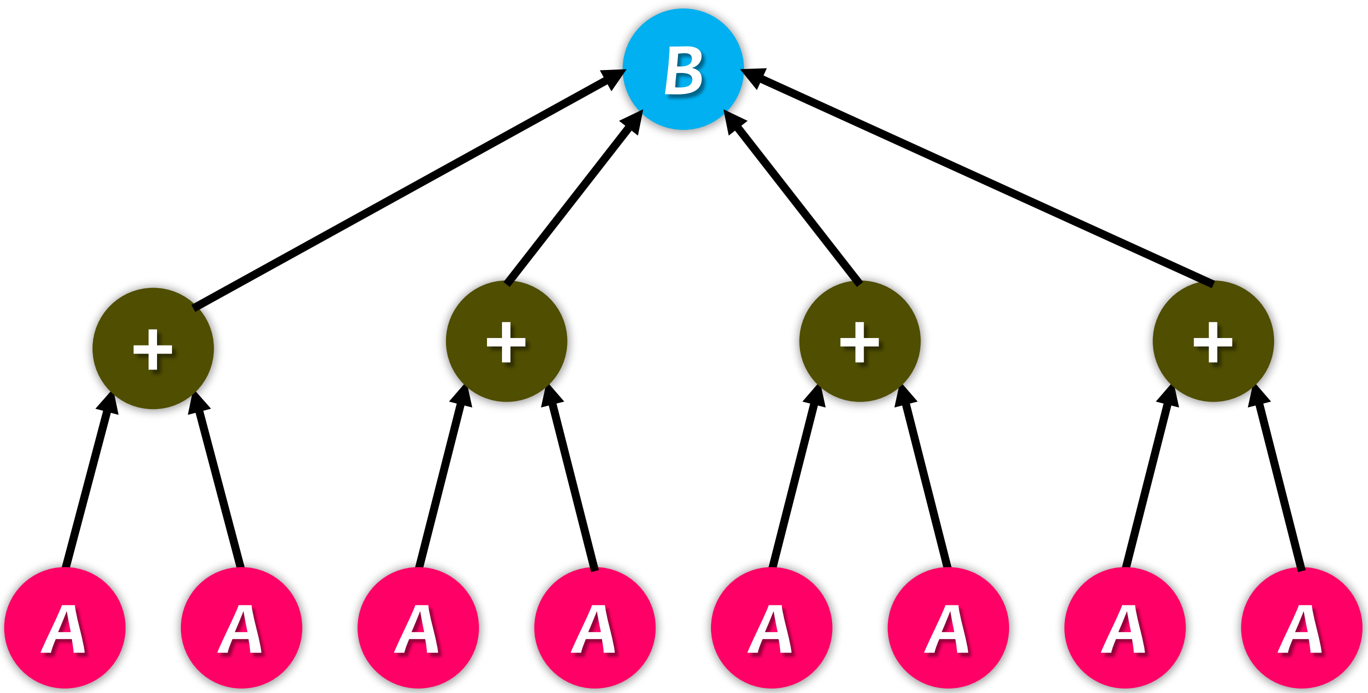
Run-time Graph Refinement

Simple Example: aggregation tree



Run-time Graph Refinement

Simple Example: aggregation tree

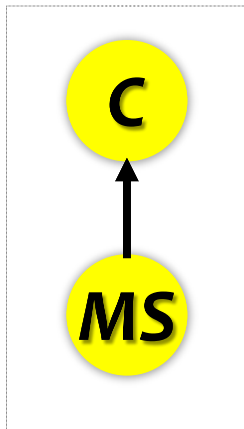


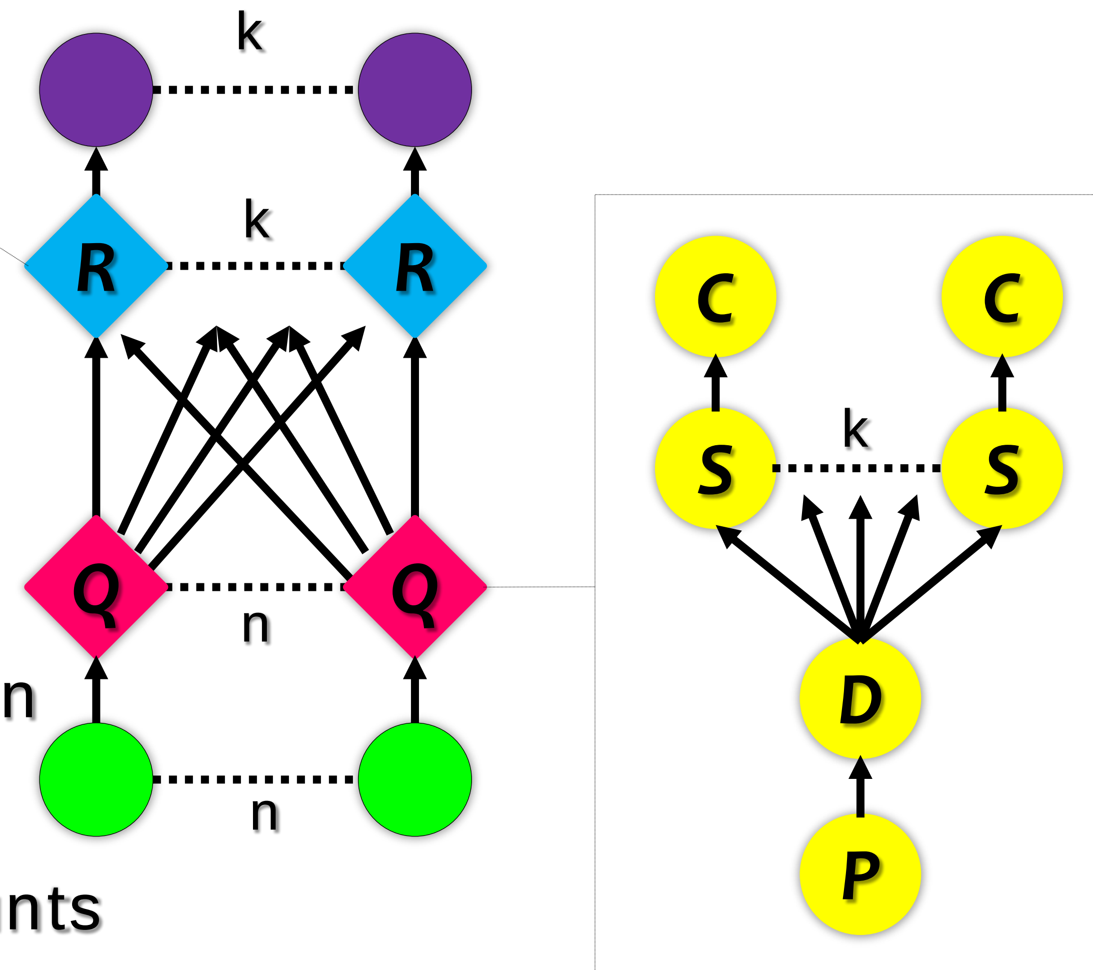
Example: Query Histogram

Computation

- **Input**: log files (n partitions)
- **Extract** queries from log partitions
- **Re-partition** by hash of query (k buckets)
- **Compute** histogram within each bucket

Example: Query Histogram





P → Parse lines

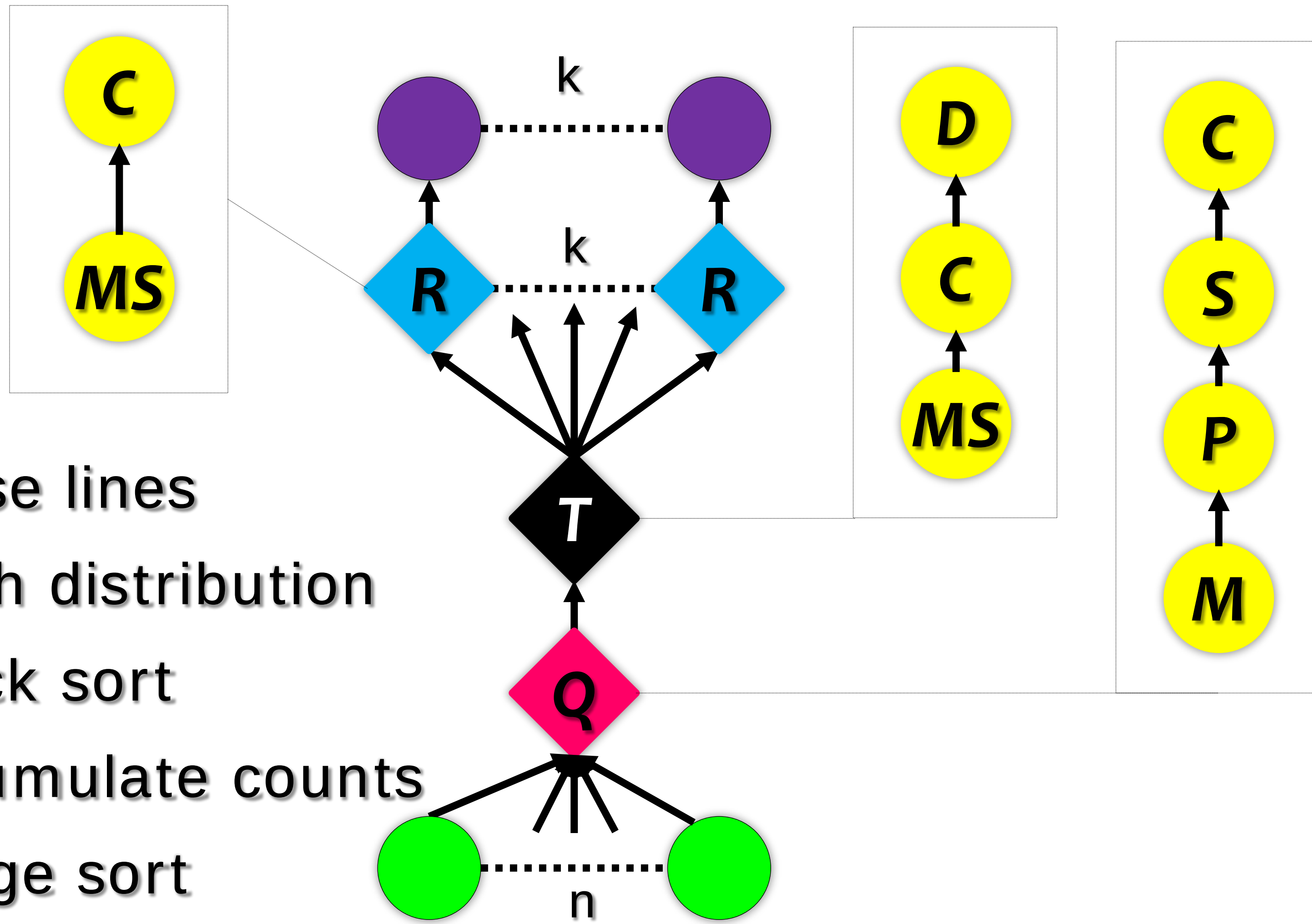
D → Hash distribution

S → Quick sort

C → Accumulate Counts

MS → Merge sort

Example: Query Histogram



P → Parse lines

D → Hash distribution

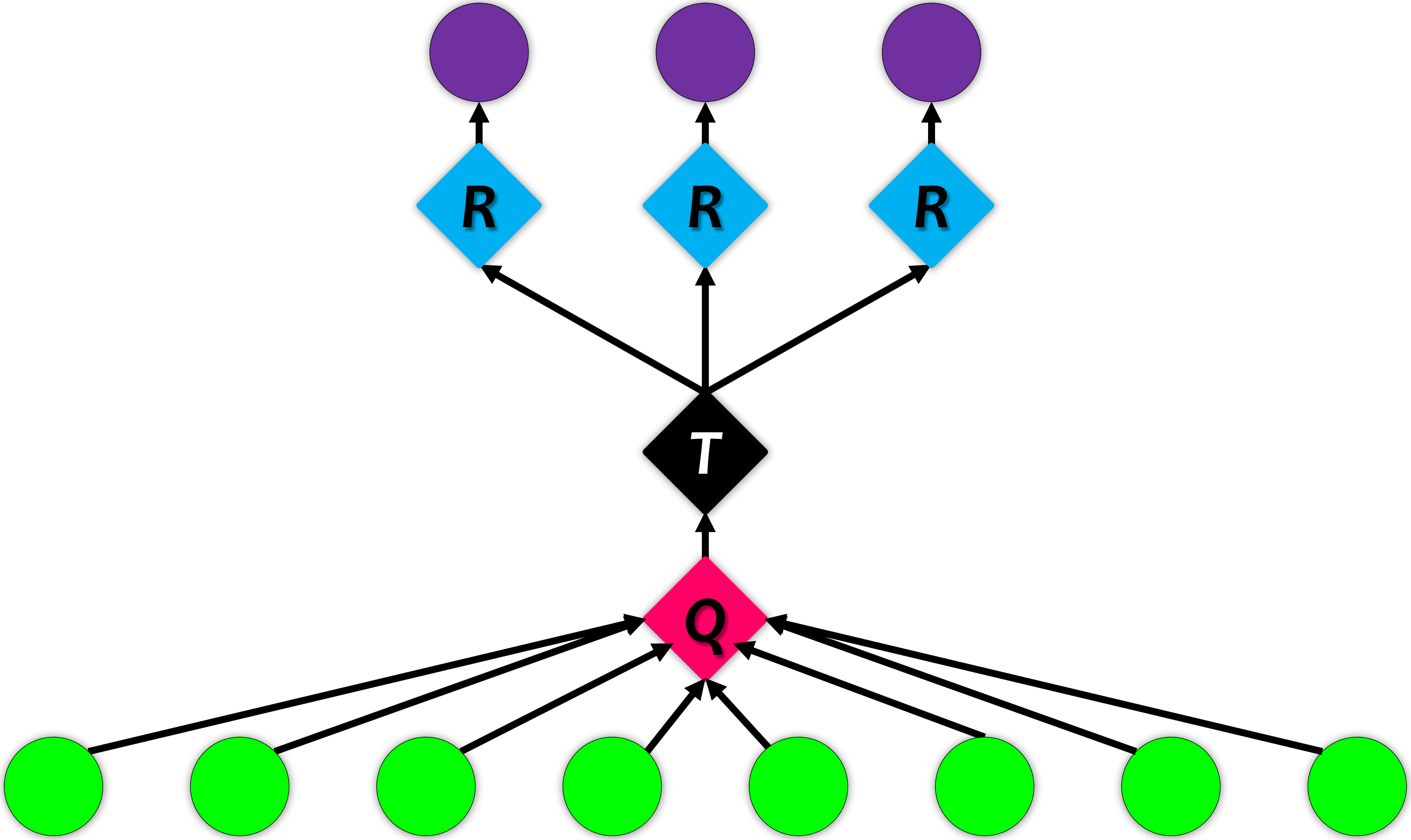
S → Quick sort

C → Accumulate counts

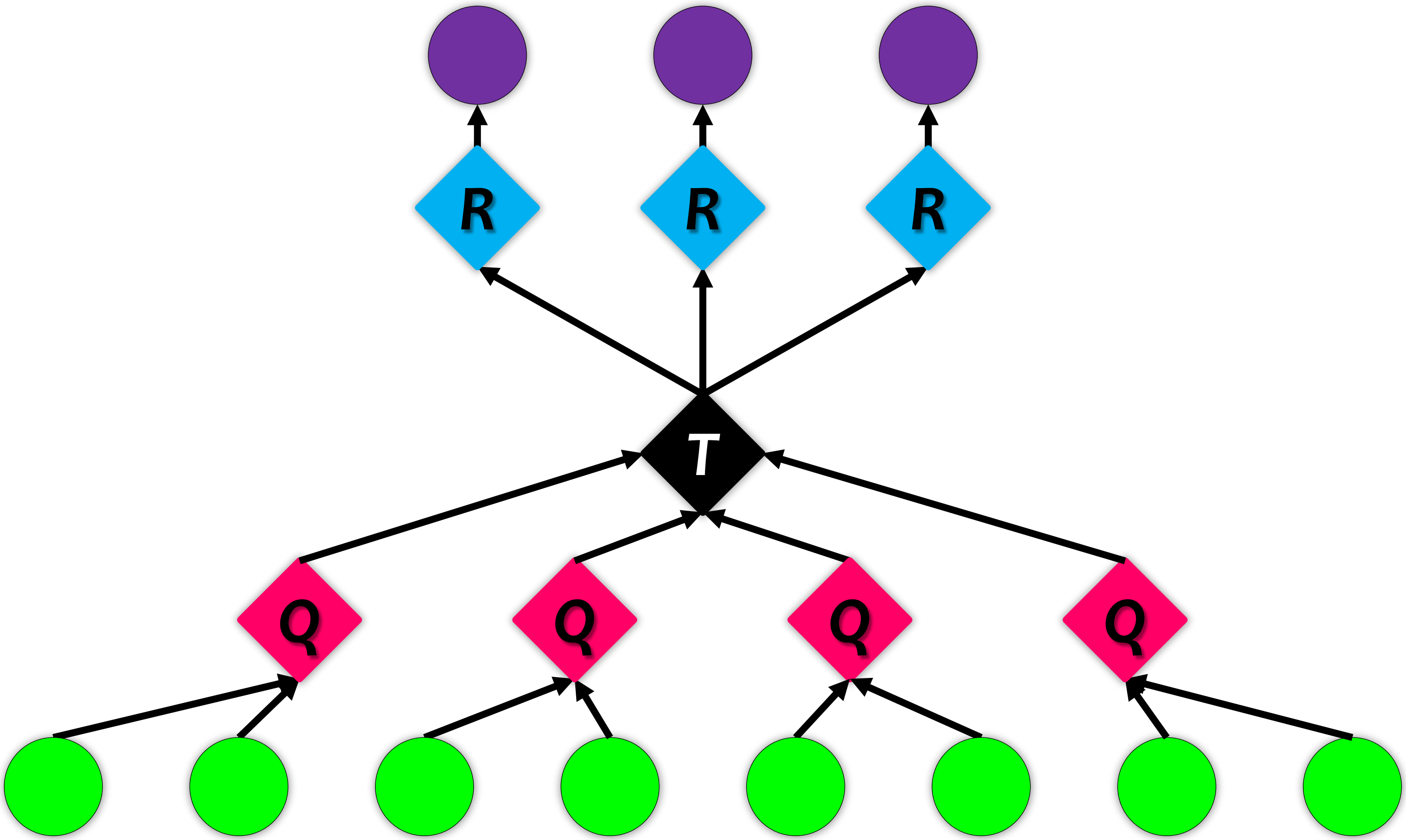
MS → Merge sort

M → Non-deterministic merge

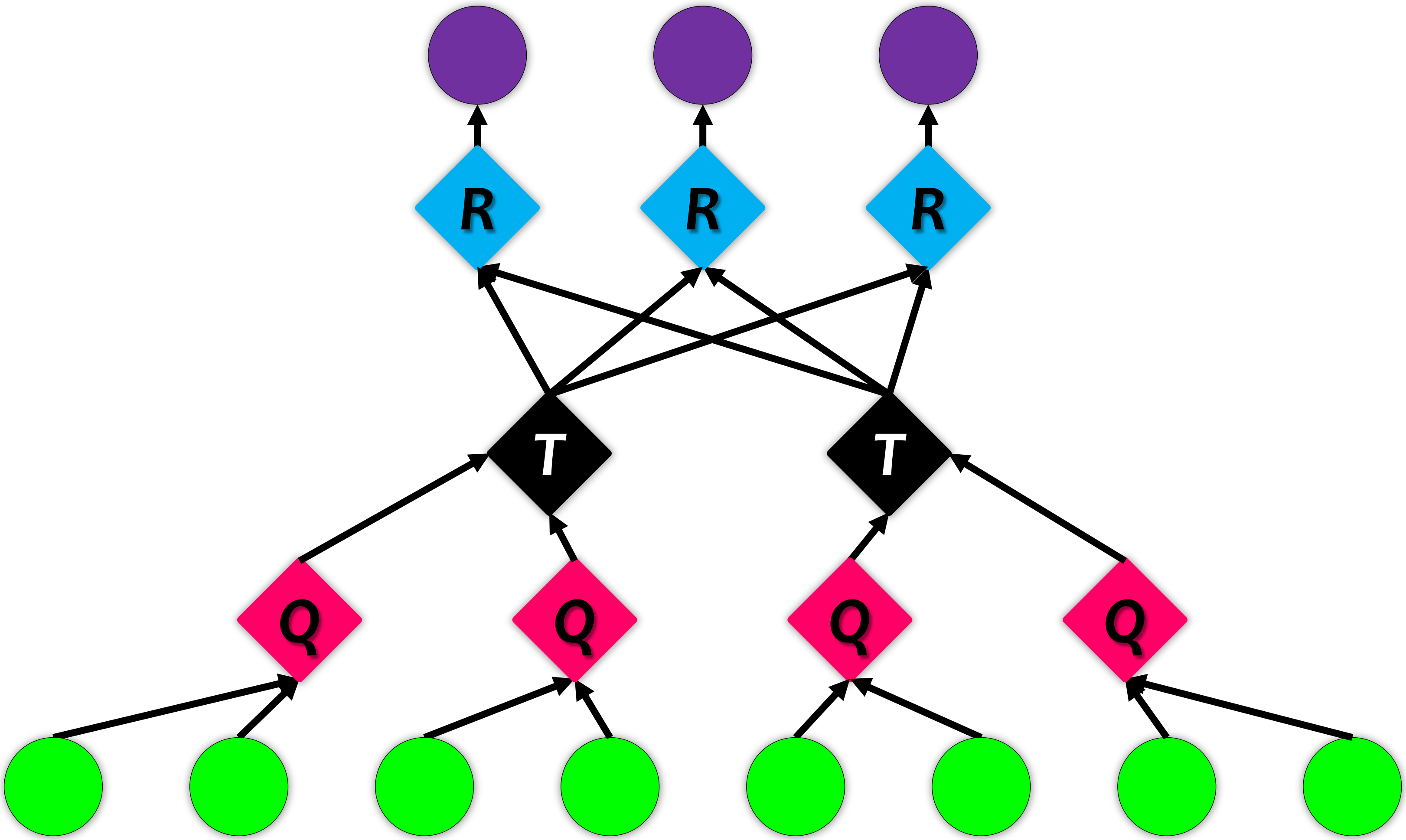
Example: Query Histogram



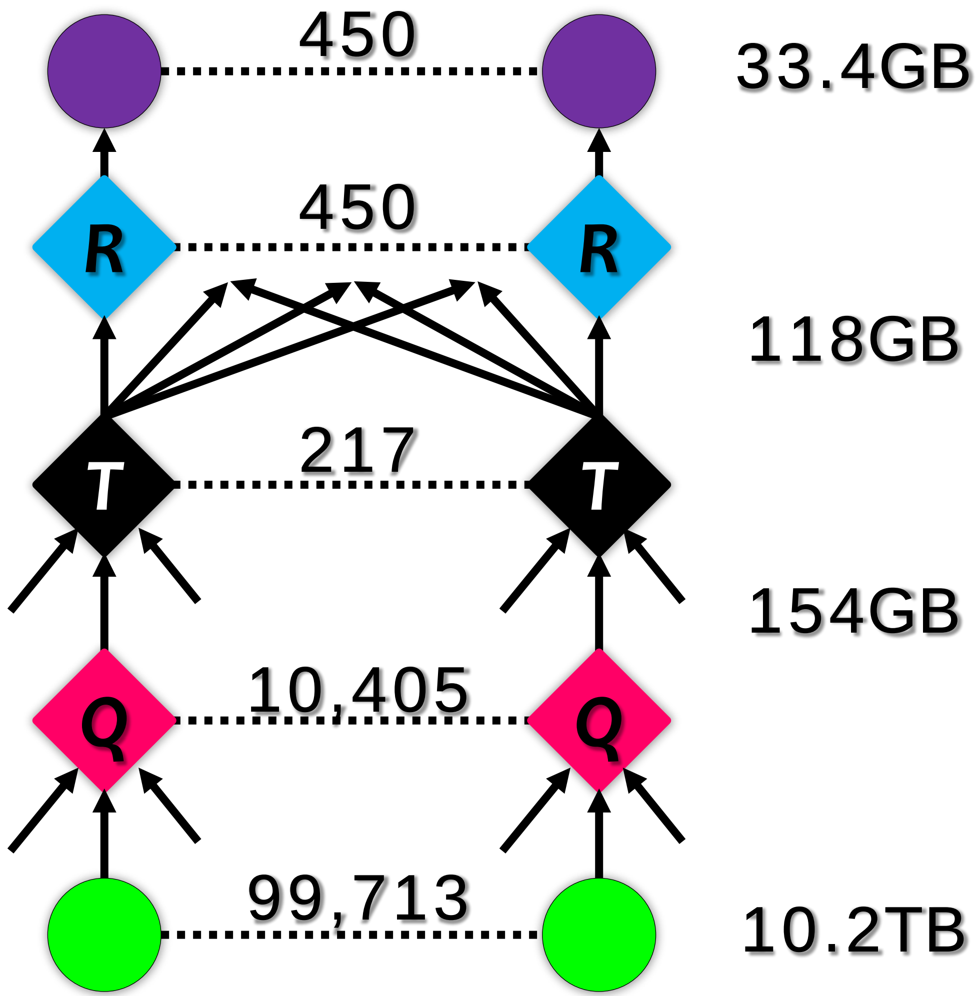
Example: Query Histogram



Example: Query Histogram



Example: Query Histogram



Summary

Dryad lets developers easily create large-scale distributed apps without requiring them to master any concurrency techniques beyond being able to draw a graph of the data dependencies of their algorithms. *-- Michael Isard*

Sacrifice some architectural simplicity compared with MapReduce system design

Provide more flexible abstract to developers expressing their code as a DAG

Next Lecture

Topic: Sequential/Release Consistency

<u>Paper</u>

Kai Li and Paul Hudak, ***"Memory Coherence in Shared Virtual Memory Systems"***. ACM symposium on Principles of distributed computing (PODC), **1986**

THANKS