Dryad & Consistency

Haibo Chen

M/R slides adapted from those of Jeff Dean's

Dryad slides adapted from those of Michael Isard

Outline

Programming distributed systems

- Dryad
- DryadLinq

Consistency

- Overview of consistency
- Sequential consistency
- Distributed shared virtual memory

Recap: MapReduce

- A programming model for large-scale computations
  - Process large amounts of input, produce output
  - No side-effects or persistent state (unlike file system)

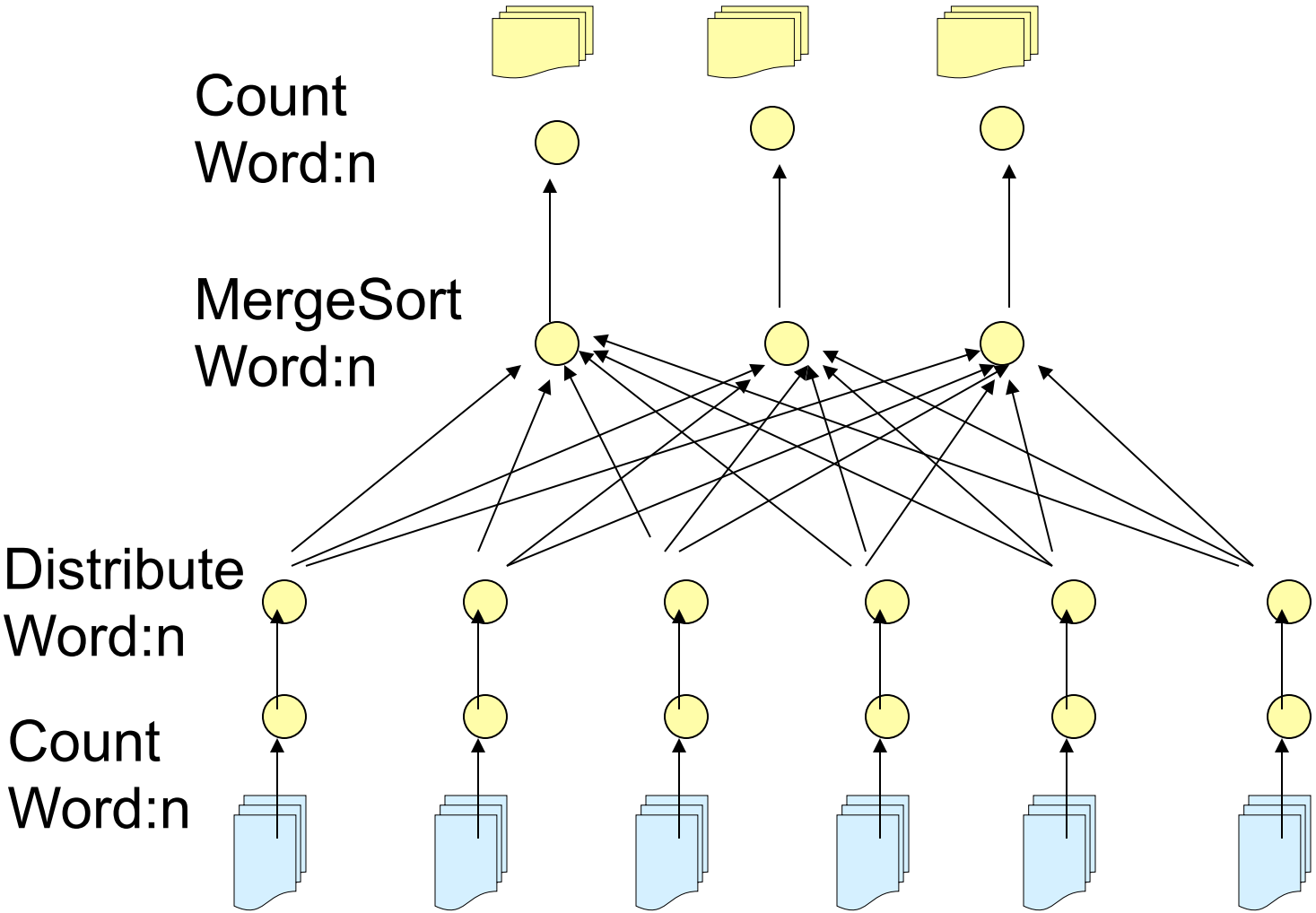
- MapReduce is implemented as a runtime library:
  - automatic parallelization
  - load balancing
  - locality optimization
  - handling of machine failures

Recap: Dryad

- Similar goals as MapReduce
  - focus on throughput, not latency
  - Automatic management of scheduling, distribution, fault tolerance

- Computations expressed as a graph
  - Vertices are computations
  - Edges are communication channels
  - Each vertex has several input and output edges

Recap: WordCount in Dryad



Advantages of DAG over MapReduce

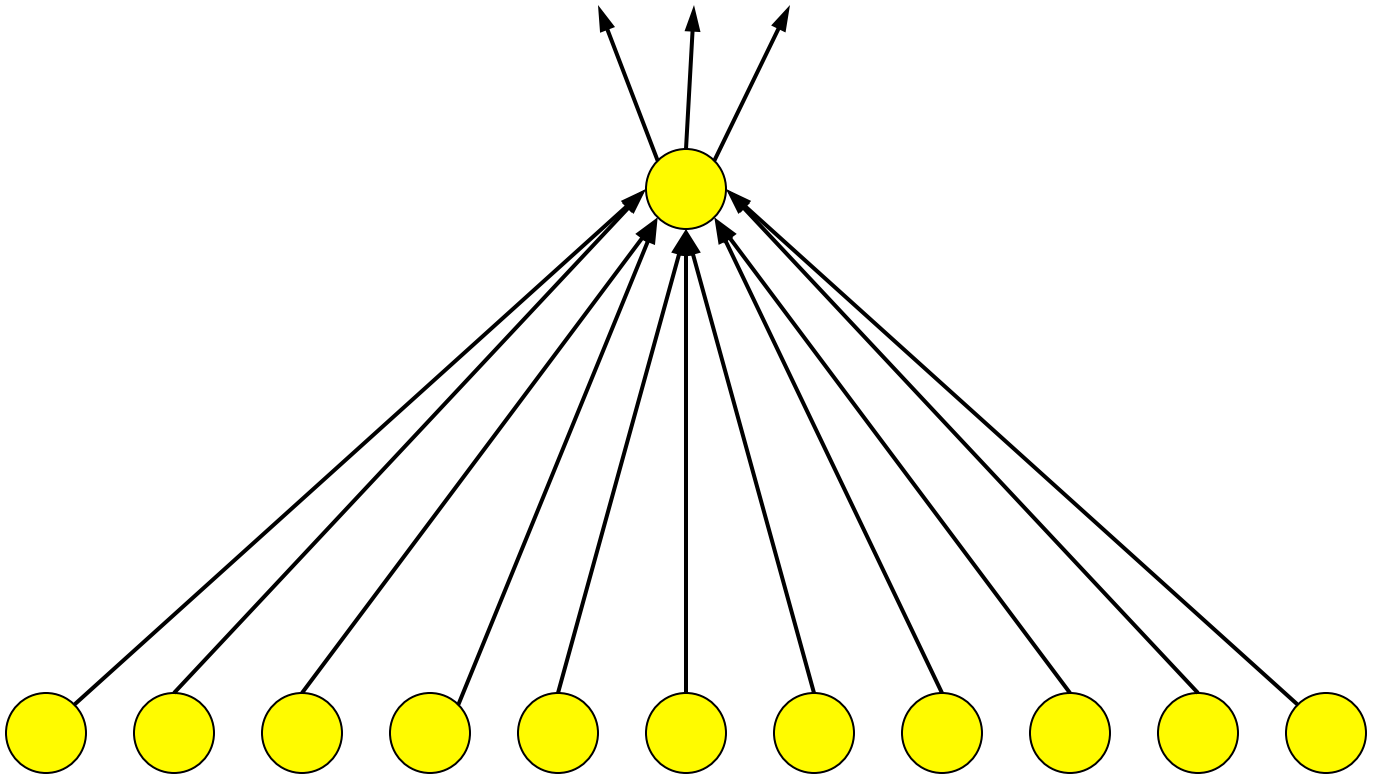
- Big jobs more efficient with Dryad
  - MapReduce: big job runs >=1 MR stages
    - reducers of each stage write to replicated storage
    - Output of reduce: 2 network copies, 3 disks
  - Dryad: each job is represented with a DAG
    - intermediate vertices write to local file

Advantages of DAG over MapReduce

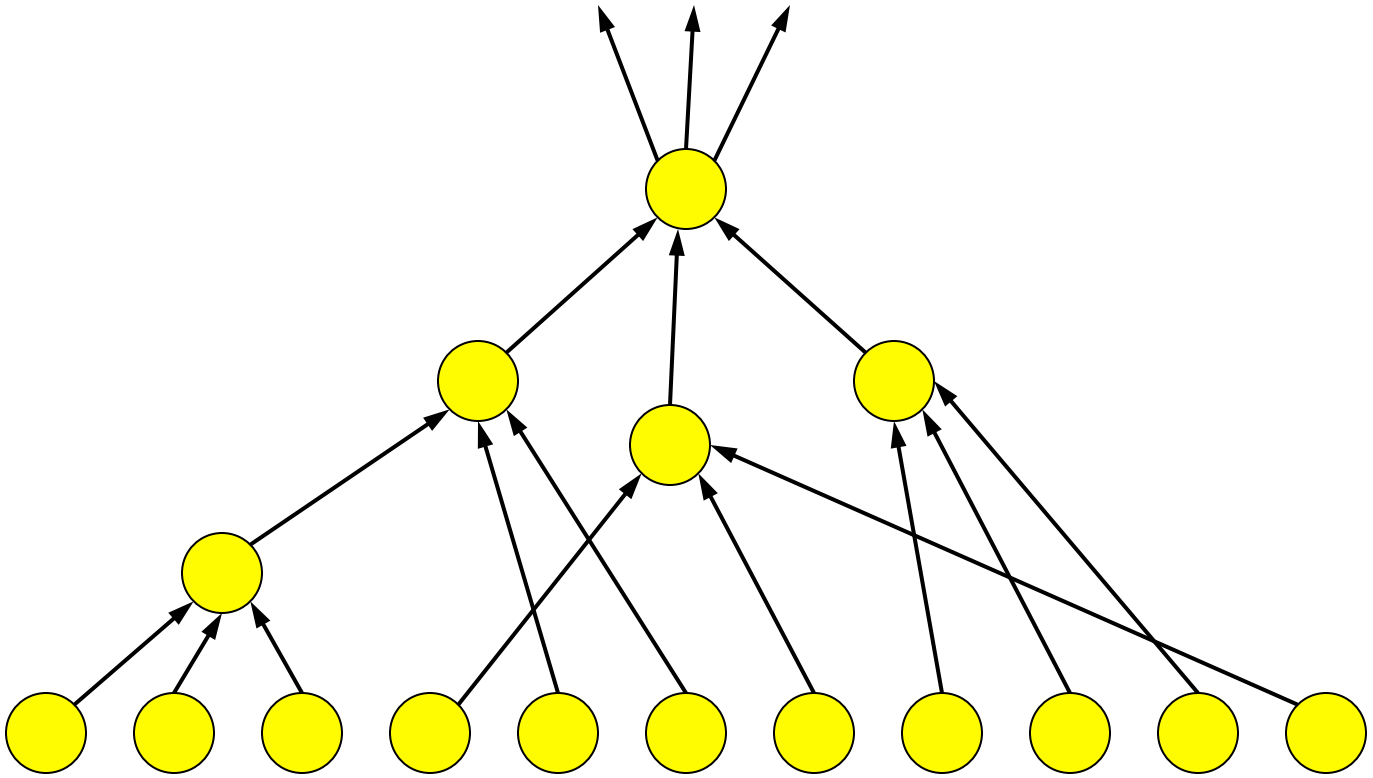
- Dryad provides explicit join
  - MapReduce: mapper (or reducer) needs to read from shared table(s) as a substitute for join
  - Dryad: explicit join combines inputs of different types

- Dryad "Split" produces outputs of different types
  - Parse a document, output text and references

DAG optimizations: merge tree



DAG optimizations: merge tree



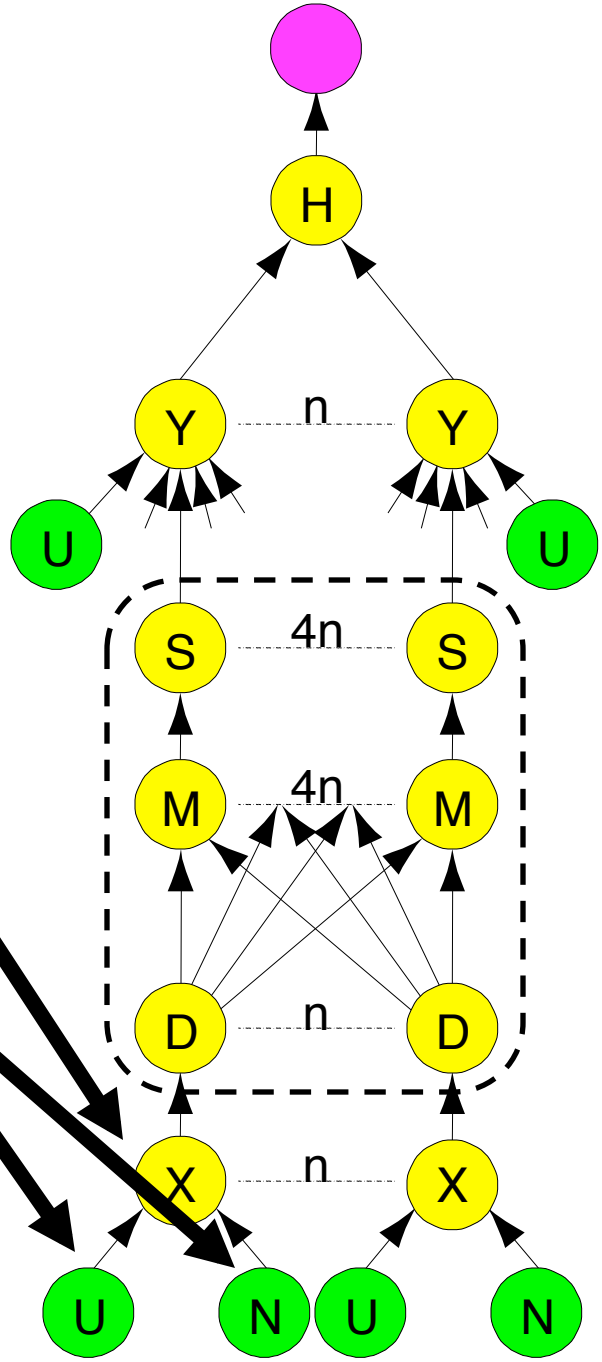
Dryad example: SkyServer Query

- 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 = N.neighborID where U.objID = N.objID, U.color
  - Join U+T to find
    U.objID where U.objID = T.neighborID
    and U.color ≈ T.color

SkyServer query

```
select
  u.color,n.neighborobjid
from u join n
where
  u.objid = n.objid
```

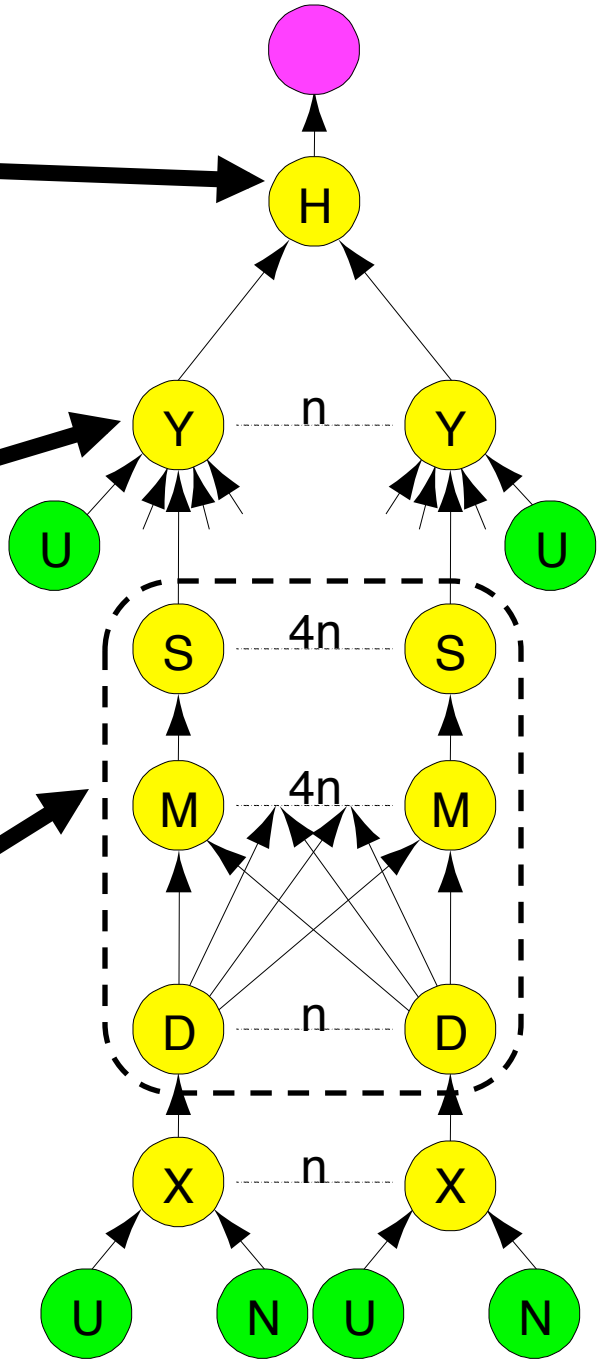
```
u: objid, color
n: objid, neighborobjid
[partition by objid]
```



```
[distinct]

[merge outputs]
```

```
(u.color,n.neighborobjid)

[re-partition by
n.neighborobjid]

[order by n.neighborobjid]
```

```
select

  u.objid

from u join <temp>

where

  u.objid = <temp>.neighborobjid
and

  |u.color - <temp>.color| < d
```

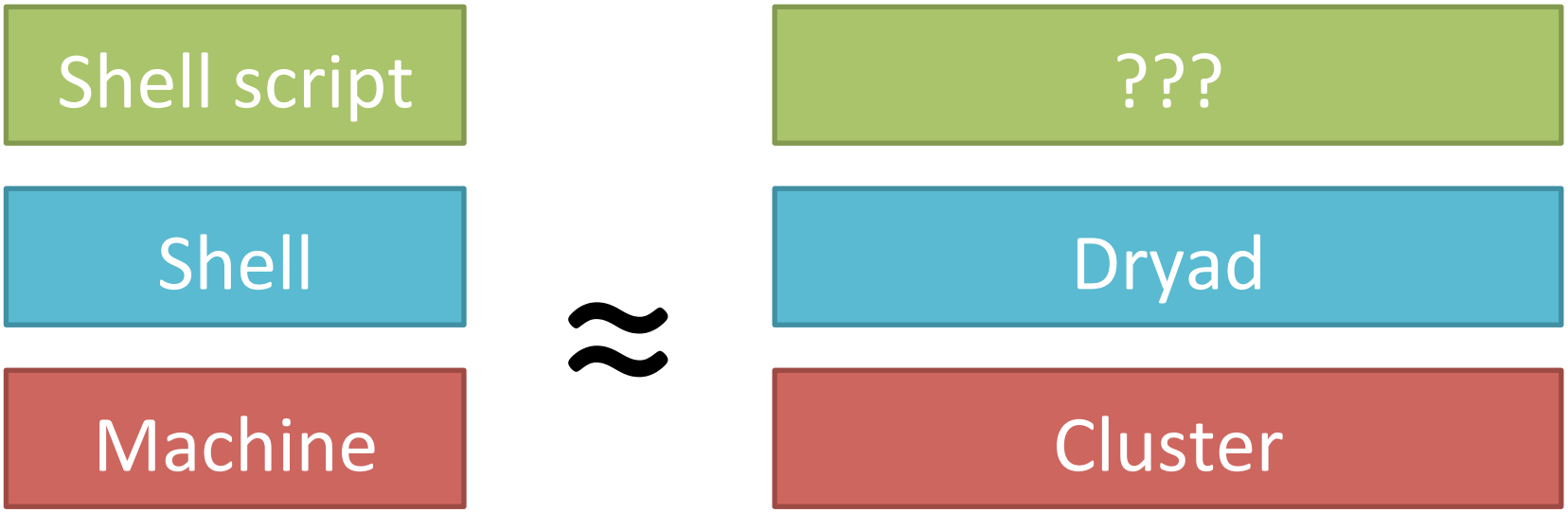


DryadLINQ

DryadLINQ

- LINQ: Relational queries integrated in C#
- Uniform data-parallel programming model
  - From SMP to clusters

Dryad = Execution Engine



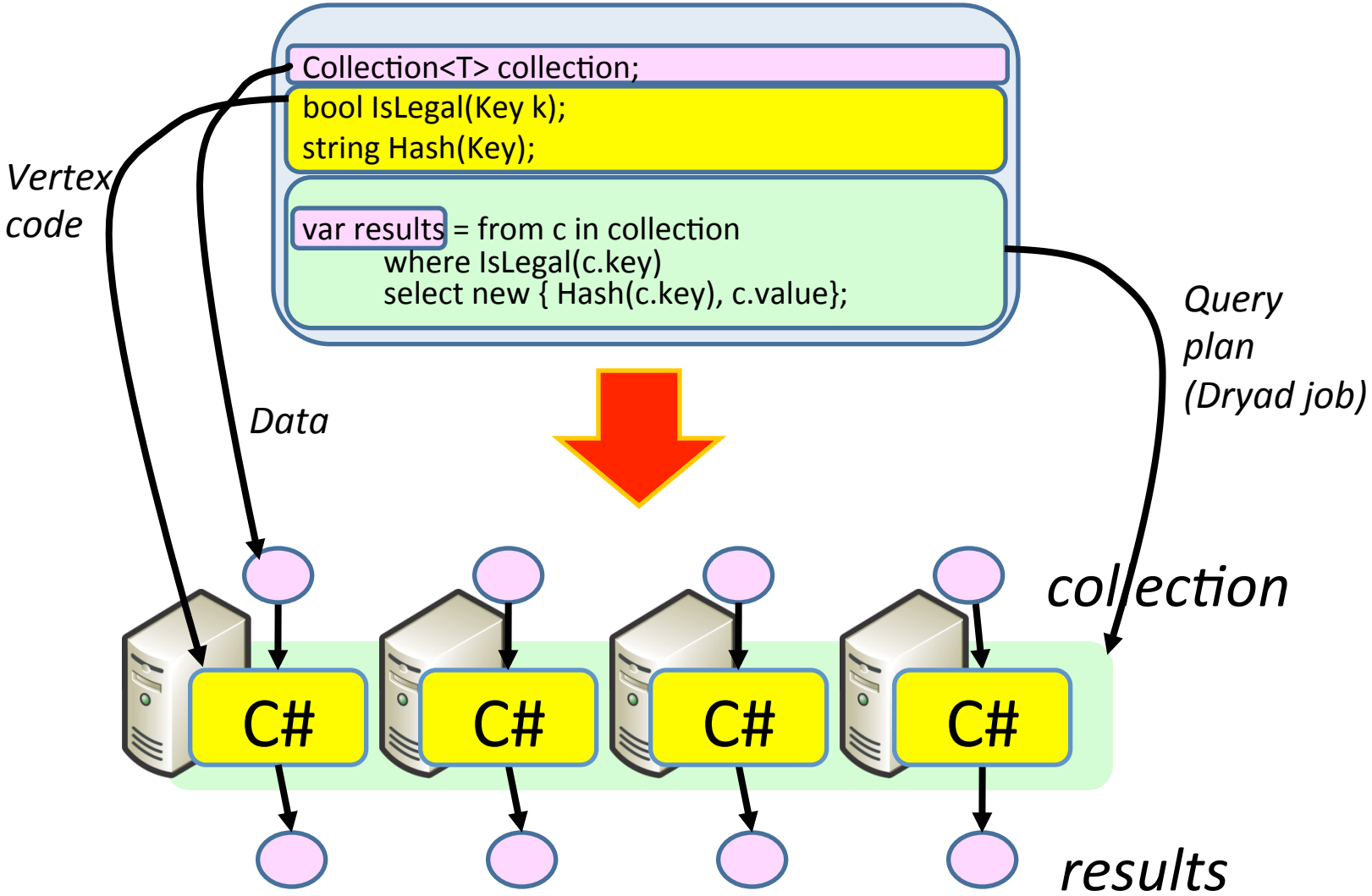
LINQ

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

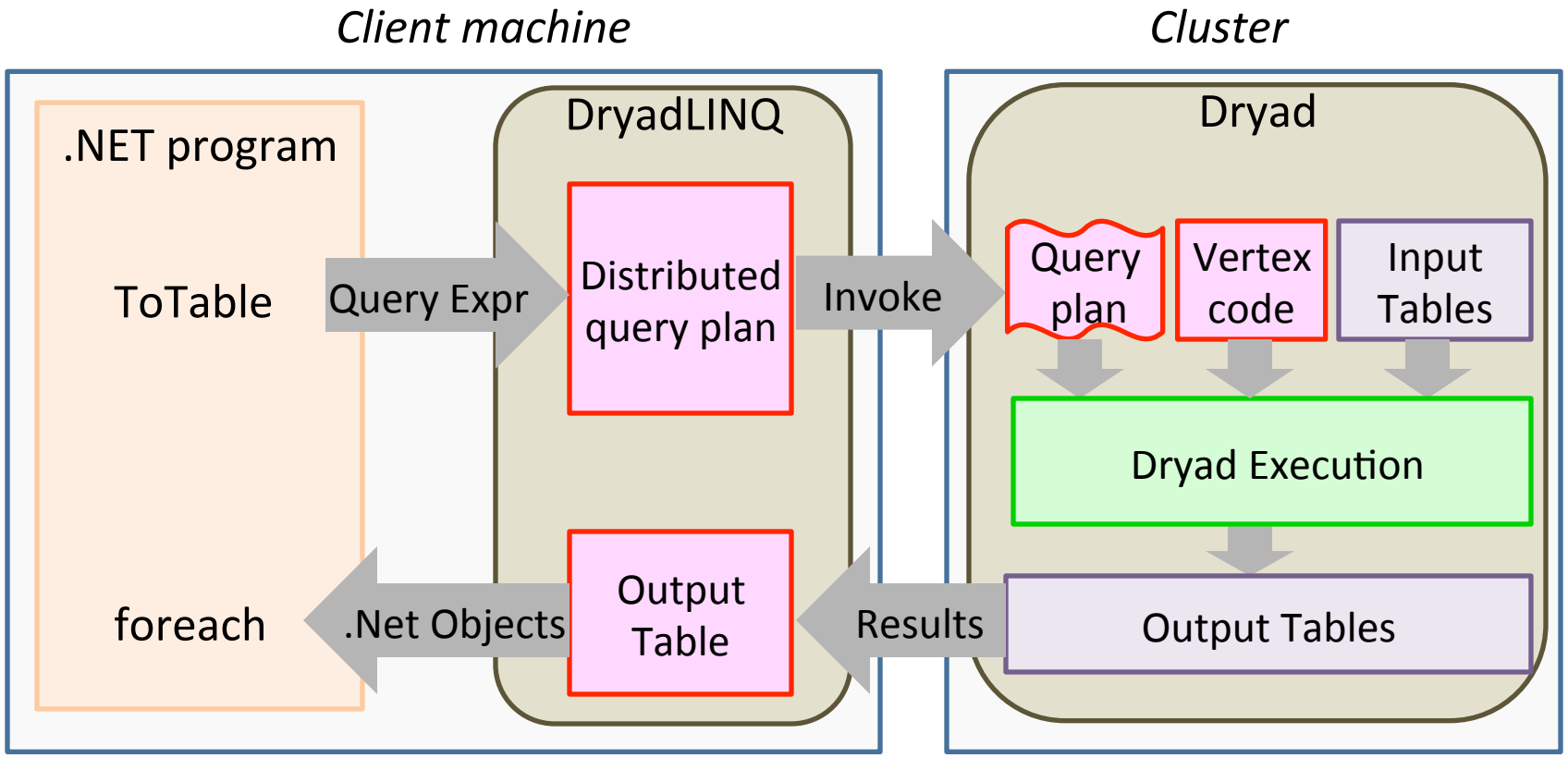
```
bool IsLegal(Key);
string Hash(Key);
```

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

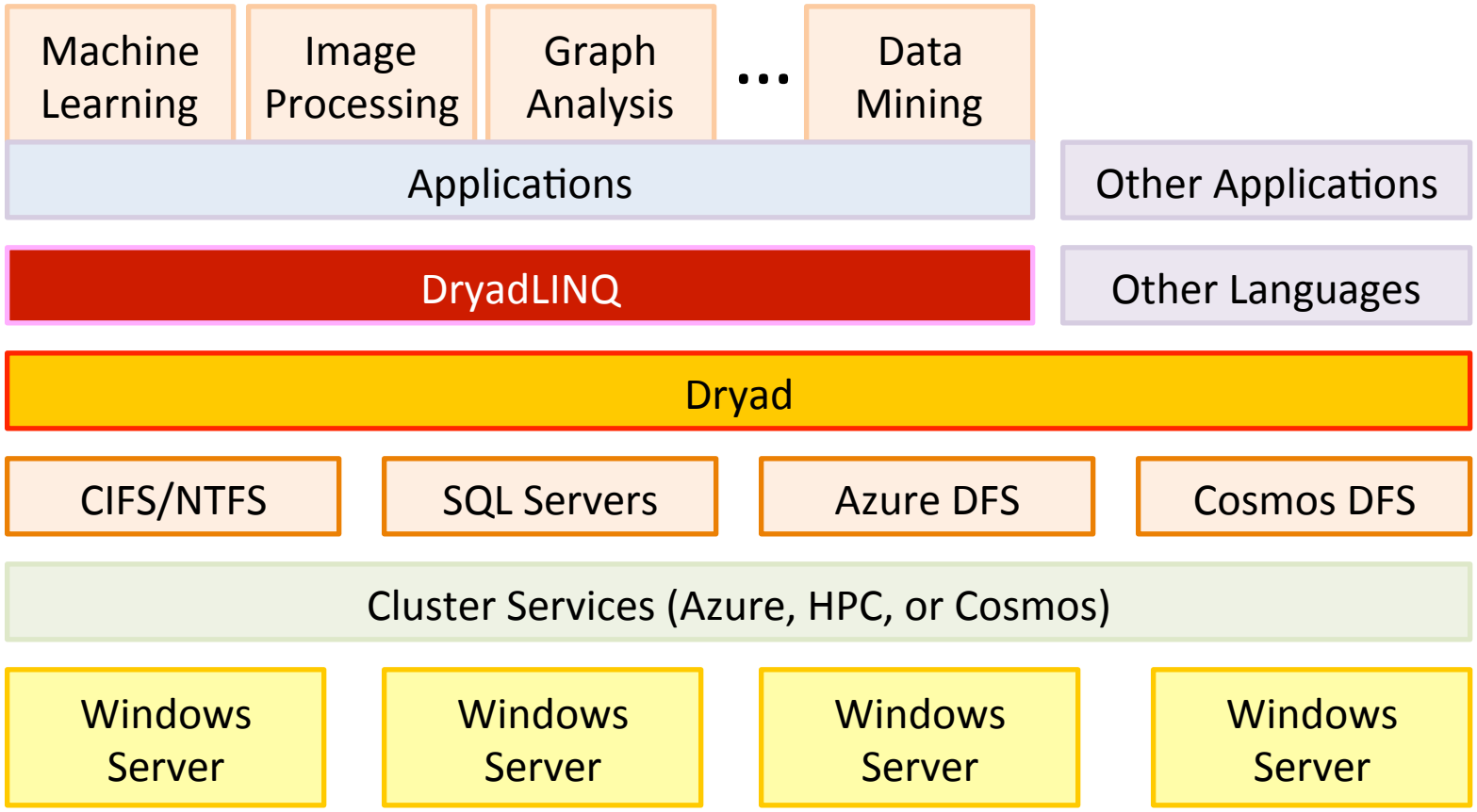
DryadLINQ = LINQ + Dryad



DryadLINQ System Architecture



Software Stack



A Simple LINQ Example: Word Count

Count word frequency in a set of documents:

```
var documents = GetDocuments();
var words = documents.SelectMany
          (document => document.Words);

var groups = words.GroupBy(word=>word);

var counts = groups.Select
             (group =>
              new WordCount
         (group.Key, group.Count()));
```

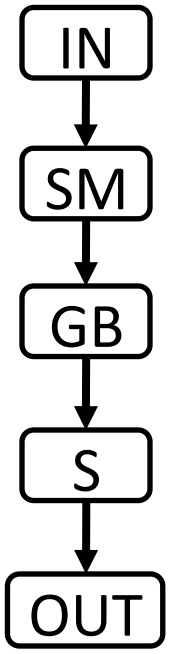
Word Count in DryadLINQ

Count word frequency in a set of documents:

```
var documents = DryadLinq.GetTable<Document>
        ("file://docs.txt");
var words = documents.SelectMany
          (document => document.Words);
var groups = words.GroupBy(word=>word);
var counts = groups.Select
           (group =>
            new WordCount
        (group.Key, group.Count()));
```

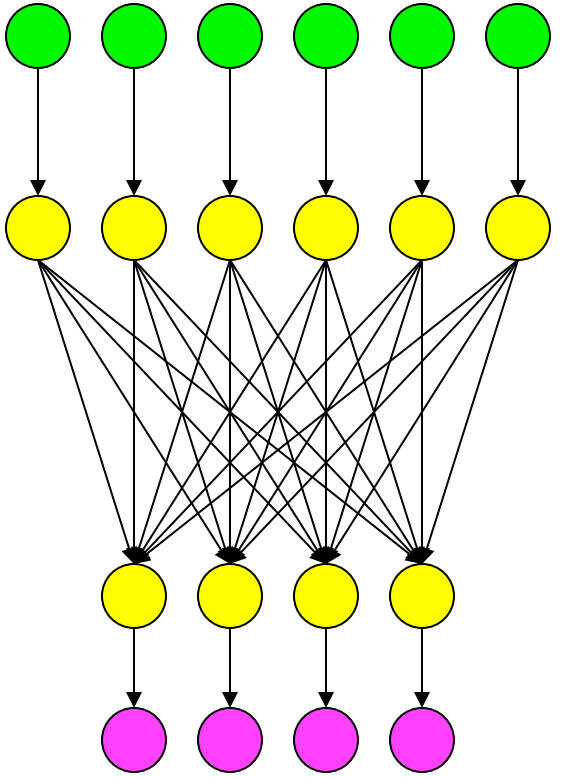
Distributed Execution of Word Count

LINQ expression



DryadLINQ

Dryad execution



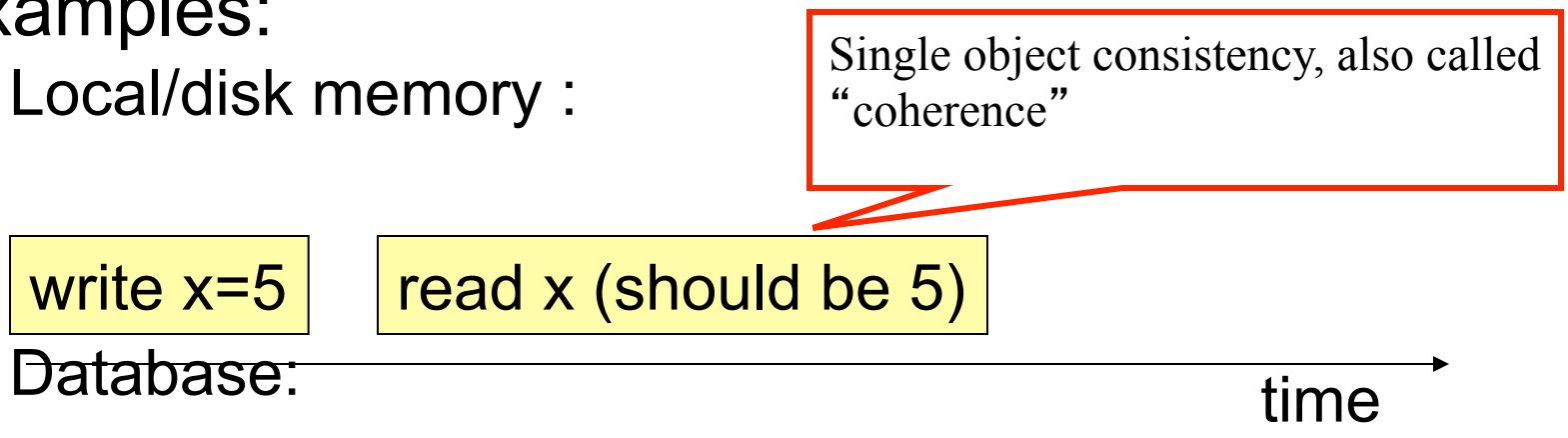
Lessons of Dryad/DryadLINQ

- Acyclic dataflow graph is a powerful computation model
- Language integration increases programmer productivity
- Decoupling of Dryad and DryadLINQ
  - Dryad: execution engine (given DAG, do scheduling and fault tolerance)
  - DryadLINQ: programming model (given query, generate DAG)

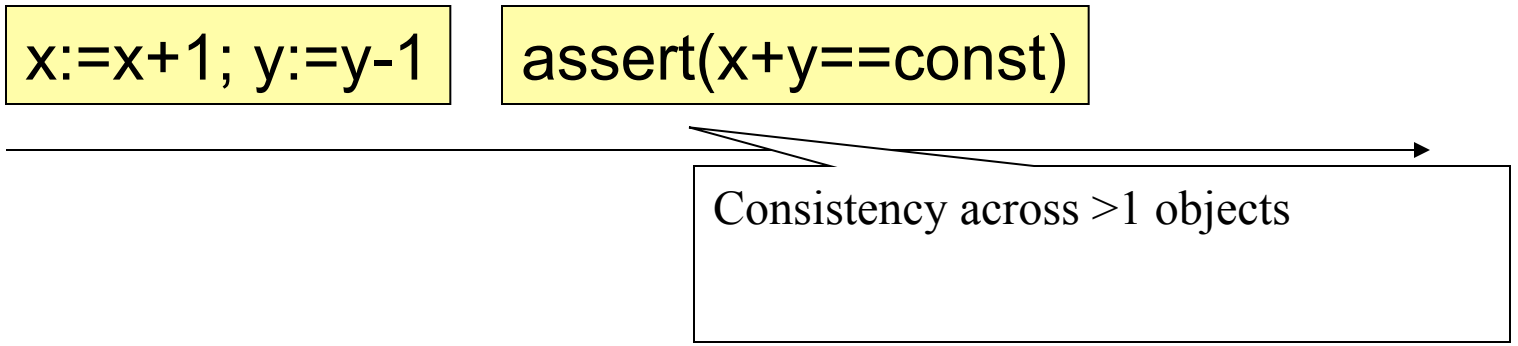
Consistency

What is consistency?

- Consistency model:
  - A constraint on the system state observable by applications
- Examples:
  - Local/disk memory :



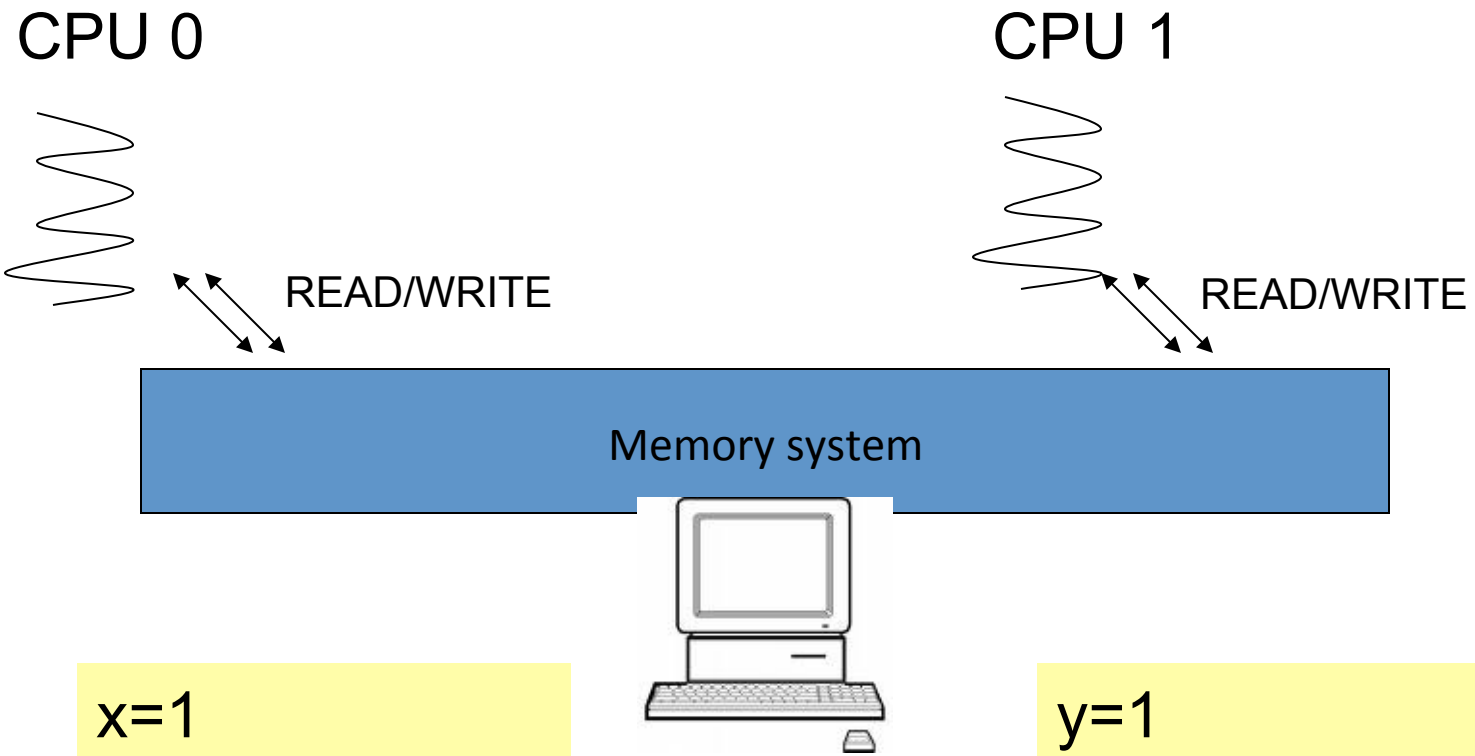
  - Database:



Consistency challenges

- No right or wrong consistency models
  - Tradeoff between ease of programmability and efficiency
  - E.g. what's the consistency model for web pages?
- Consistency is hard in (distributed) systems:
  - Data replication (caching)
  - Concurrency
  - Failures

Example application program




```
x=1
If y==0
   critical section
```

```
y=1
If x==0
   critical section
```

- Is this program correct?

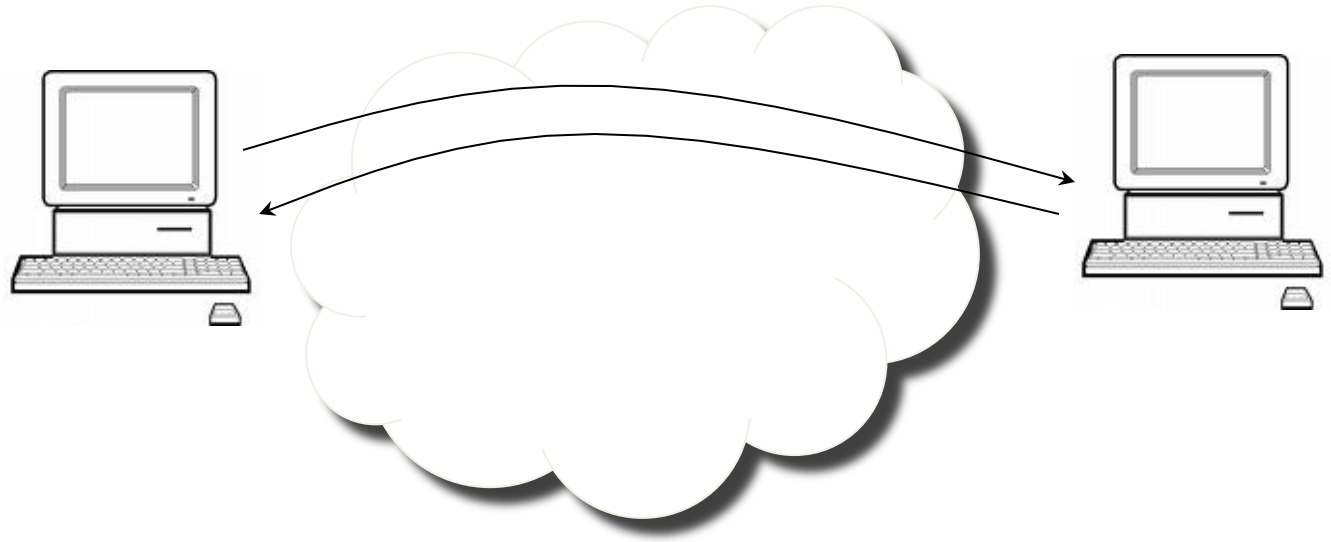
Example

```
x=1
If y==0
   critical section
```

```
y=1
If x==0
   critical section
```

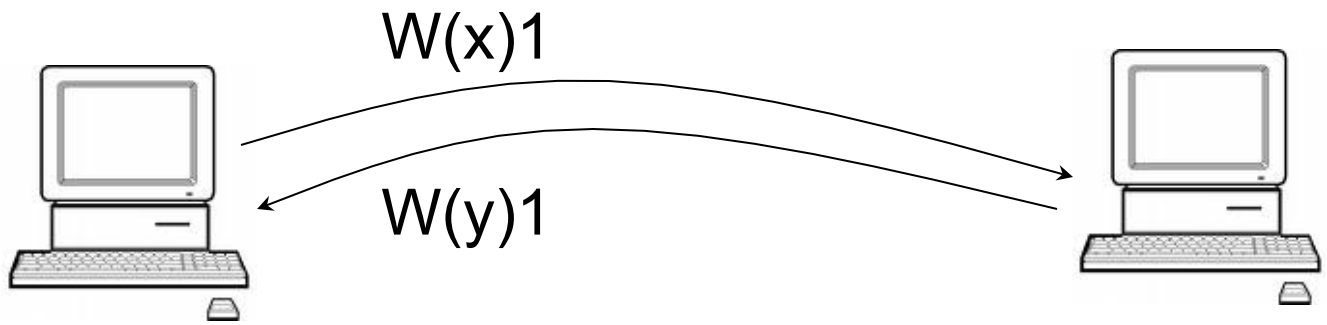
- CPU0's instruction stream: W(x) R(y)
- CPU1's instruction stream: W(y) R(x)
- Enumerate all possible inter-leavings:
  - W(x)1 R(y)0 W(y)1 R(x)1
  - W(x)1 W(y)1 R(y)1 R(x)1
  - W(x)1 W(y)1 R(x)1 R(y)1
  - ….
- None violates mutual exclusion

An example distributed shared memory



- Each node has a local copy of state
- Read from local state
- Send writes to the other node, but do not wait

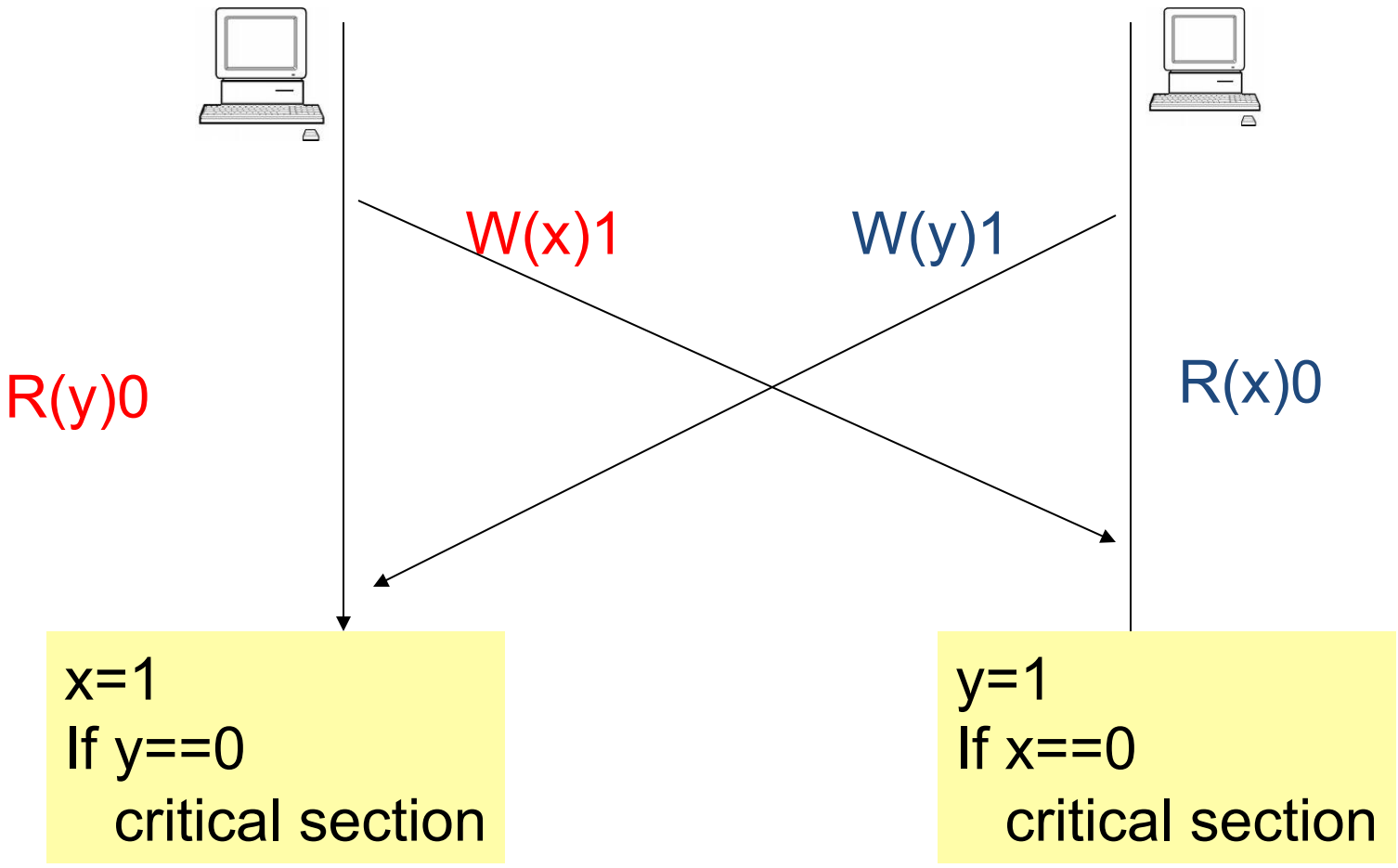
Consistency challenges: example



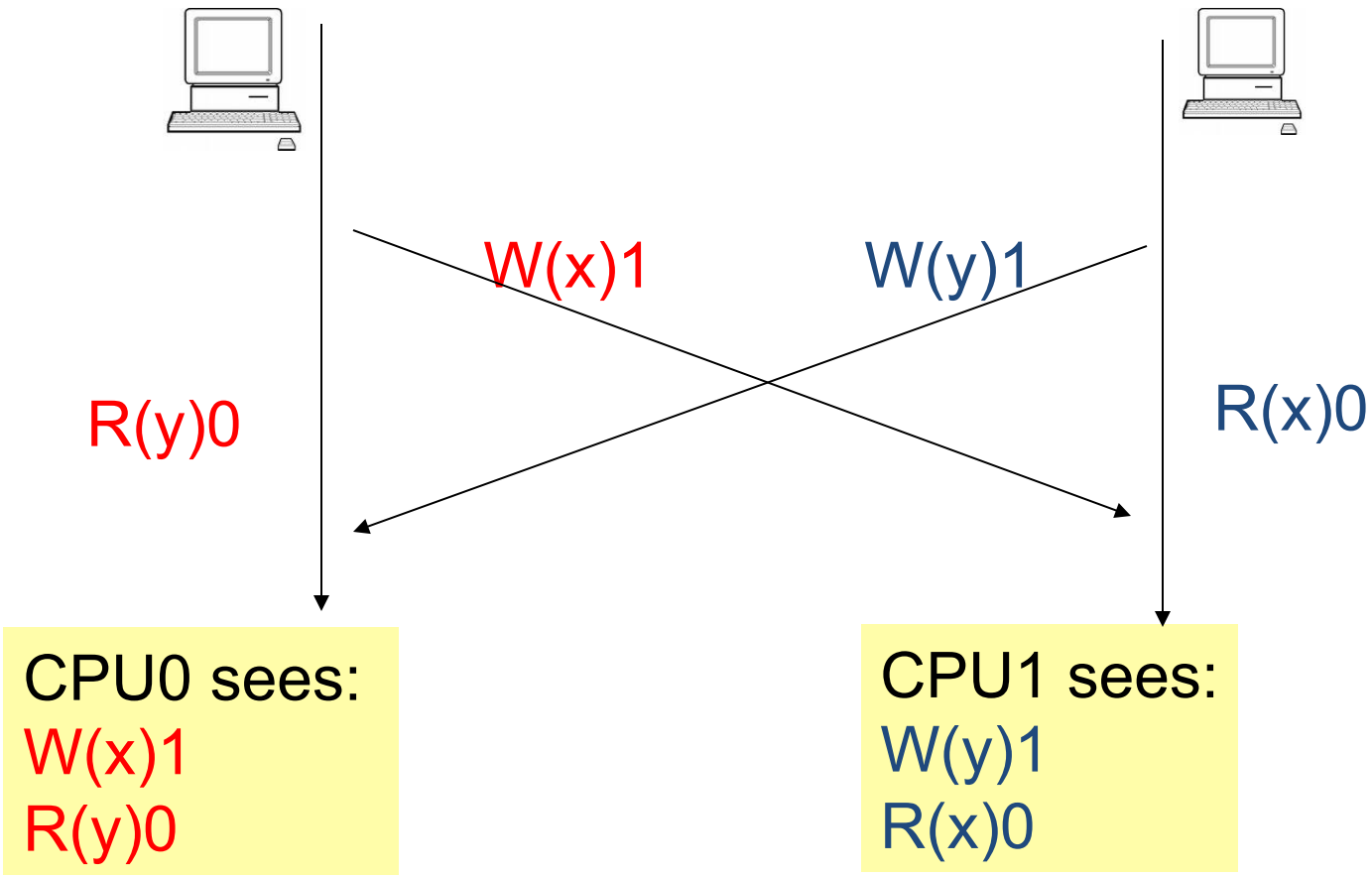

```
x=1
If y==0
   critical section
```

```
y=1
If x==0
   critical section
```

Does this work?



What went wrong?

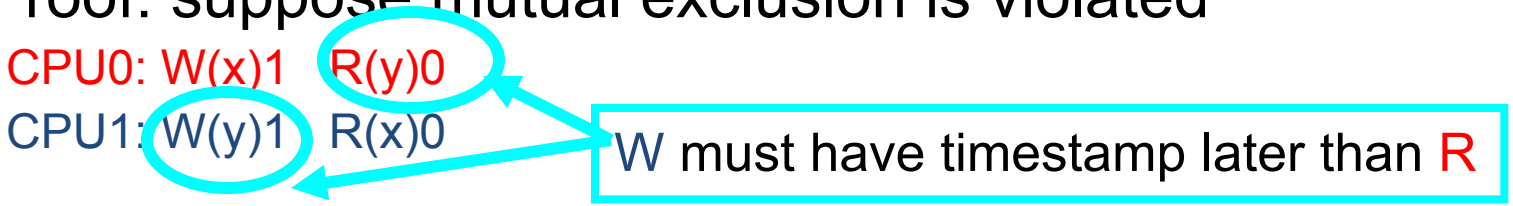


Strict consistency

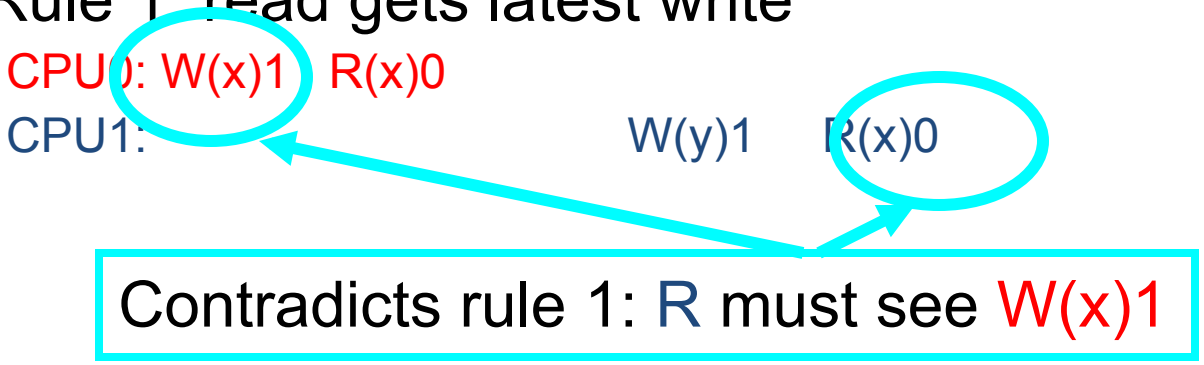
- Each operation is stamped with a global wall-clock time
- Rules:
  1. Each read gets the latest write value
  2. All operations at one CPU have time-stamps in execution order

Strict consistency gives "intuitive" results

- No two CPUs in the critical section
- Proof: suppose mutual exclusion is violated



- Rule 1: read gets latest write



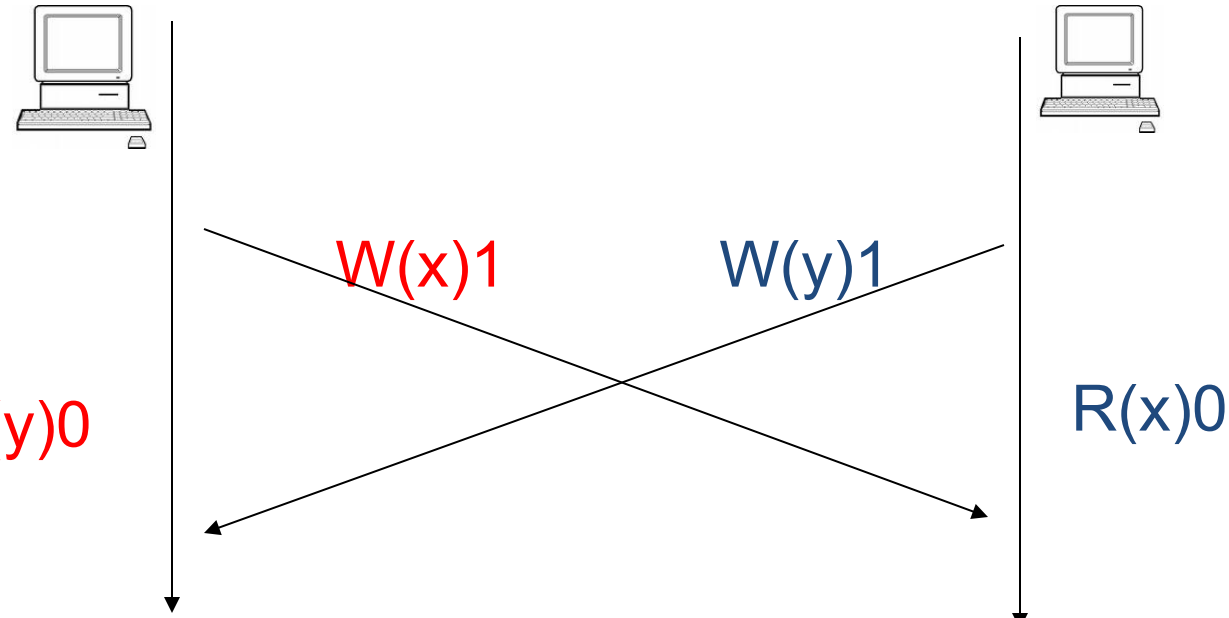
Sequential consistency

- Strict consistency is not practical
  - No global wall-clock available
- Sequential consistency is the closest
- Rules: There is a total order of ops
  - Each CPUs' ops appear in order
  - All CPUs see results according to total order (i.e. reads see most recent writes)

Sequential consistency is also intuitive

- Recall our proof for correctness
- Enumerate all possible total orders:
  - Each CPUs' ops appear in order
  - All CPUs see results according to total order (i.e. reads see most recent writes)
- Show no total order violates mutual excl

Naive DSM example gives no sequential consistency



CPU0 sees:
W(x)1
R(y)0
W(y)1

CPU1 sees:
W(y)1
R(x)0
W(x)1

No total order can explain this execution

Sequential consistency is easier to implement

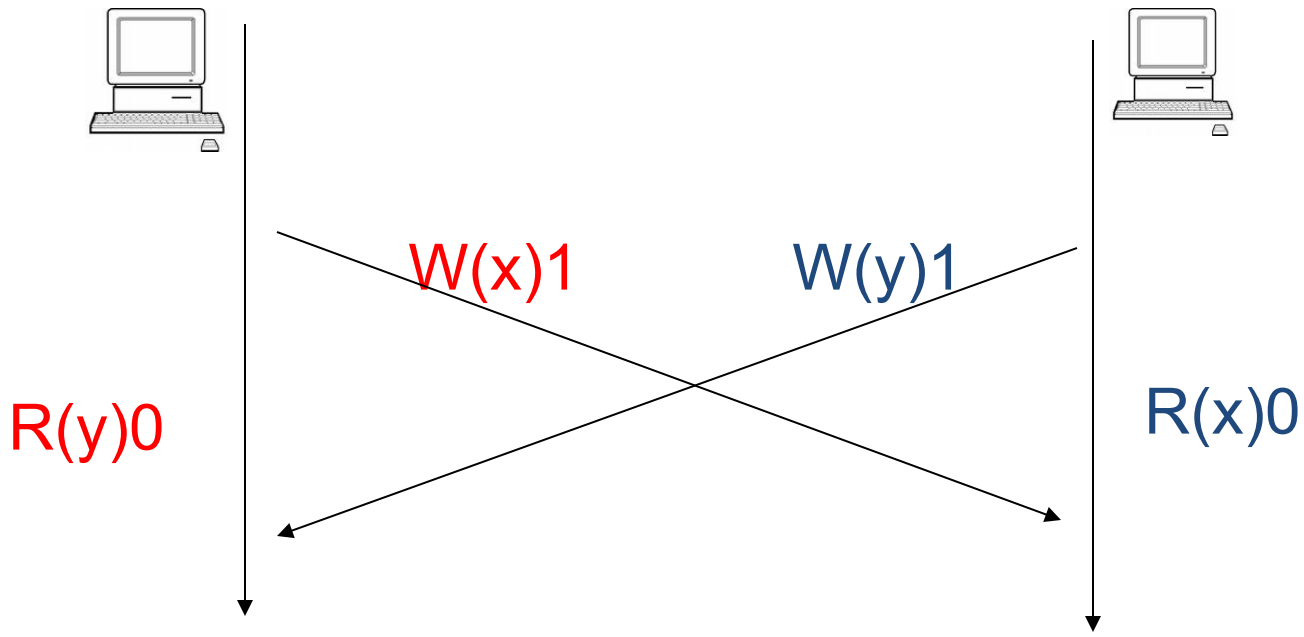
- There's no notion of real time
- System is free to order concurrent events
- However, when the system finishes a write or reveals a read, it commits to certain partial orders

Requirement for sequential consistency

1. Each processor issues requests in the order specified by the program
   - Do not issue the next request unless the previous one has finished

2. Requests to an individual memory location (storage object) are served from a single FIFO queue.
   - Writes occur in a single order
   - Once a read observes the effect of a write, it's ordered behind that write

Naive DSM violates R1,R2



- Read from local state
- Send writes to the other node, but do not wait

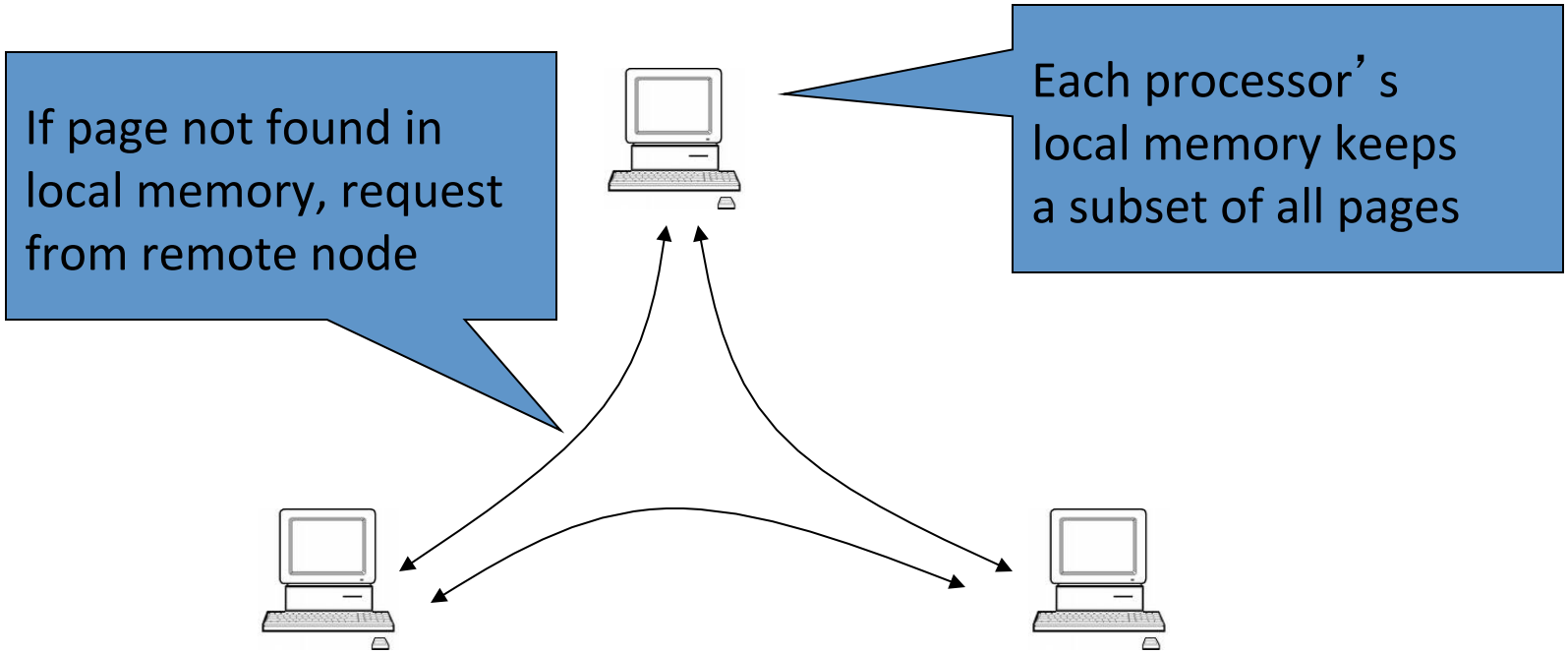
~~R1~~: a processor issues read before waiting for write to complete

~~R2~~: 2 processors issue writes concurrently, no single order

Ivy distributed shared memory

- What does Ivy do?
  - Provide a shared memory system across a group of workstations
- Why shared memory?
  - Easier to write parallel programs with than using message passing
  - We'll come back to this choice of interface in later lectures

Ivy architecture



- Each node caches read pages
  - Why?
- Can a node directly write cached pages?

Ivy aims to provide sequential consistency

- How?
  - Always read a fresh copy of data
    - Must invalidate all cached pages before writing a page.
    - This simulates the FIFO queue for a page because once a page is written, all future reads must see the latest value
  - Only one processor (owner) can write a page at a time

Ivy implementation

- The ownership of a page moves across nodes
  - Latest writer becomes the owner
  - Why?
- Challenge:
  - how to find the owner of a page?
  - how to ensure one owner per page?
  - How to ensure all cached pages are invalidated?

Ivy: centralized manager

Page#, copy_set, owner
p1, {..}, {A}

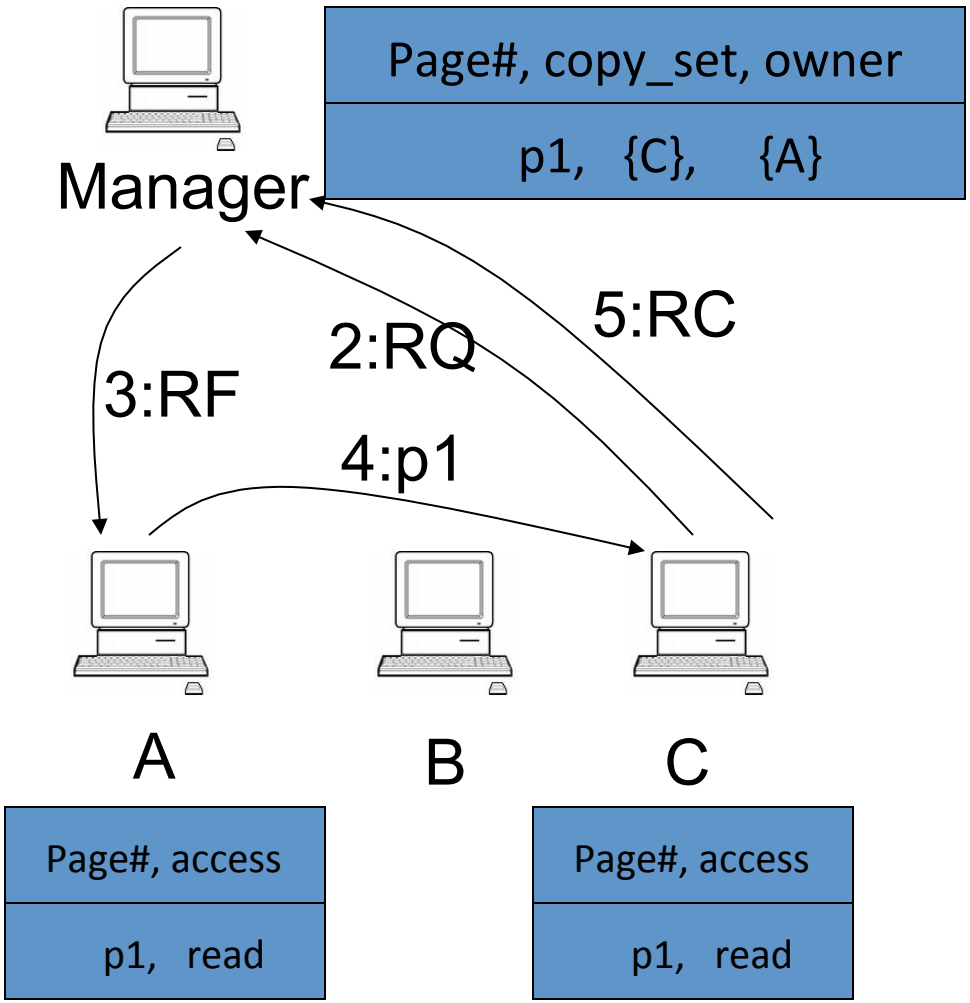
Page#, access
p1, read


A


B

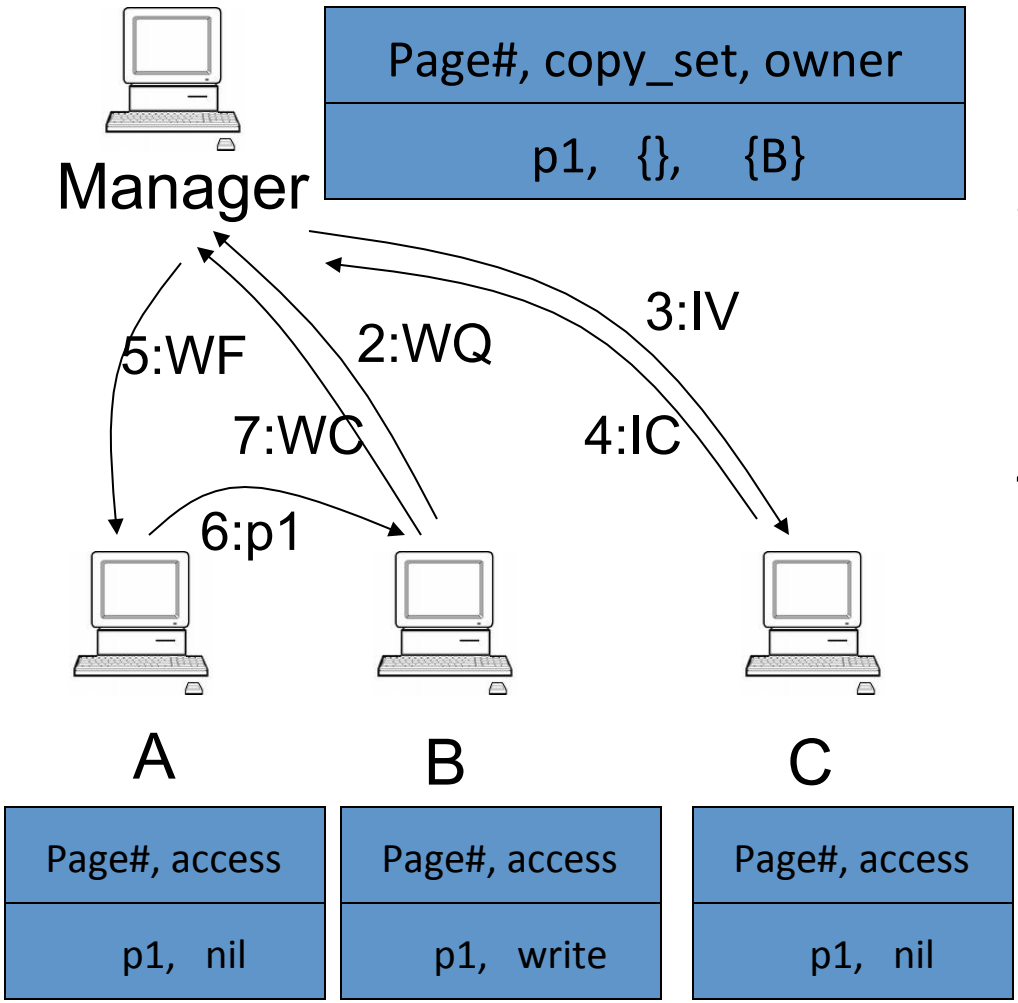

C

Ivy: read



1. Page fault for p1 on C
2. C sends RQ(read request) to M
3. M sends RF(read forward) to A, M adds C to copy_set
4. A sends p1 to C, C marks p1 as read-only
5. C sends RC(read confirmation) to M

Ivy: write



1. Page fault for p1 on B
2. B sends WQ(write request) to M
3. M sends IV(invalidate) to copy_set = {C}
4. C sends IC(invalidate confirm) to M
5. M clears copy_set, sends WF(write forward) to A
6. A sends p1 to B, clears access
7. B sends WC(write confirmation) to M

Ivy properties

```
x=1
If y==0
   critical section
```

```
y=1
If x==0
   critical section
```

- Does Ivy work with our example app?
- Why does it need RC and WC messages?

Ivy invariants?

- Every page has exactly one current owner
- Current owner has a copy of the page
- If mapped r/w by owner, no other copies
- If mapped r/o by owner, identical to other copies
- Manager knows about all copies

Next Class

- Consistency: Relaxed consistency
  Preparation: Read TreadMarks

Backup Slides