Piccolo: Building fast distributed programs with partitioned tables

Russell Power

Jinyang Li

New York University

Motivating Example: PageRank

Repeat until convergence

```
for each node X in graph:
  for each edge X→Z:
    next[Z] += curr[X]
```

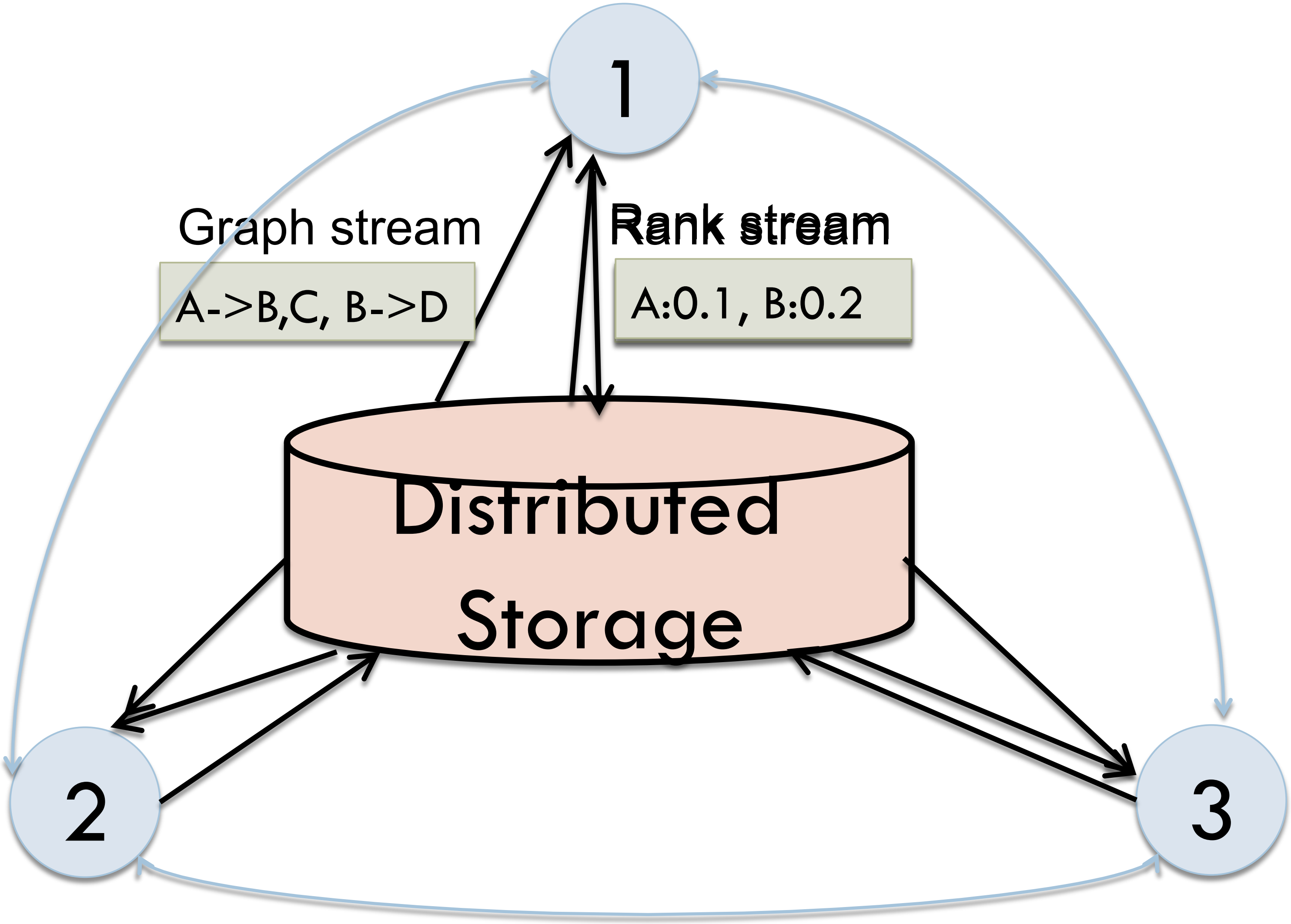
Input Graph
A→B,C,D
B→E
C→D
…

Curr	Next
A: 0.25	A: 0.25
B: 0.17	B: 0.17
C: 0.22	C: 0.22
…	…

Fits in memory!

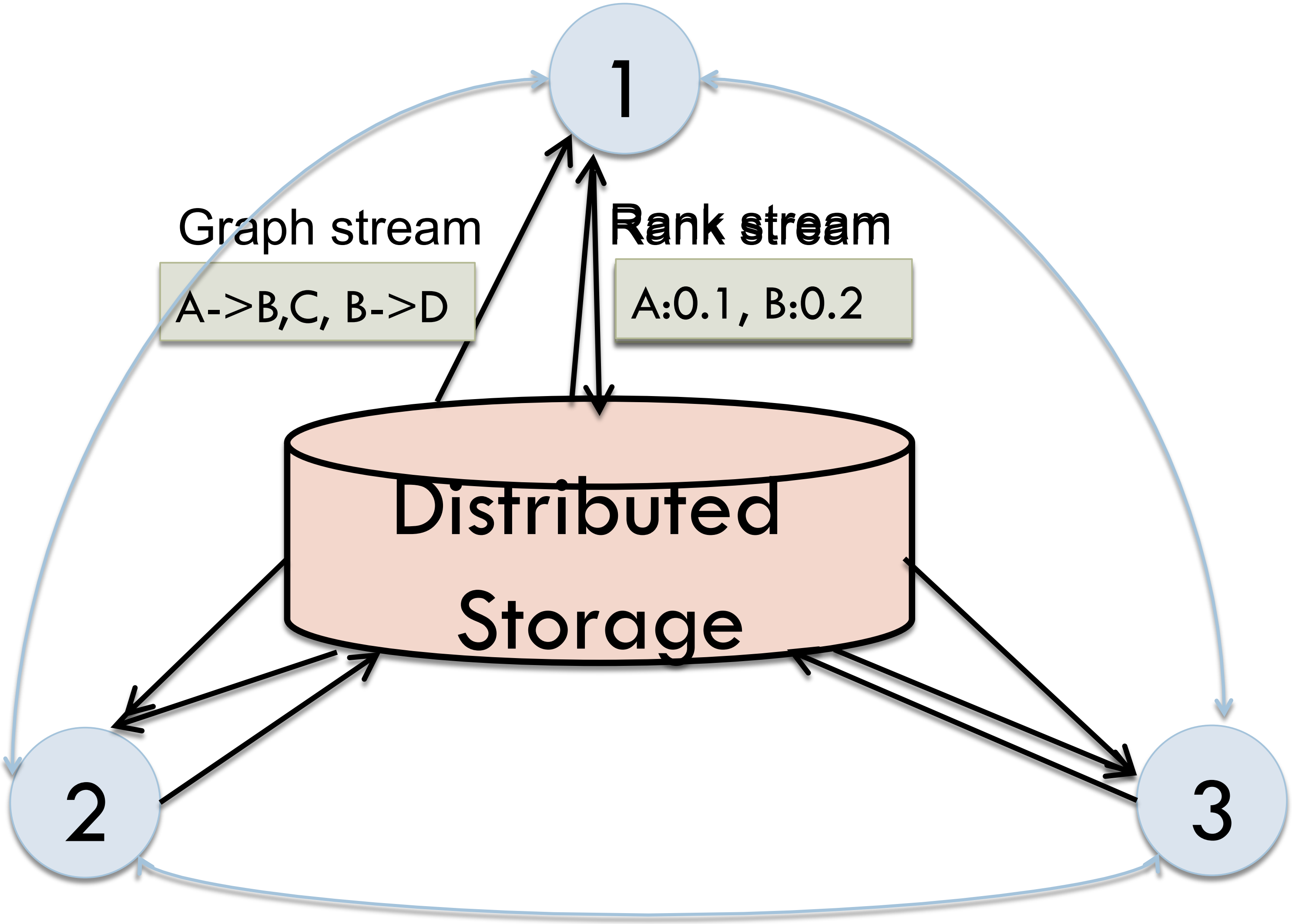
PageRank in MapReduce

- Data flow models do not expose global state.

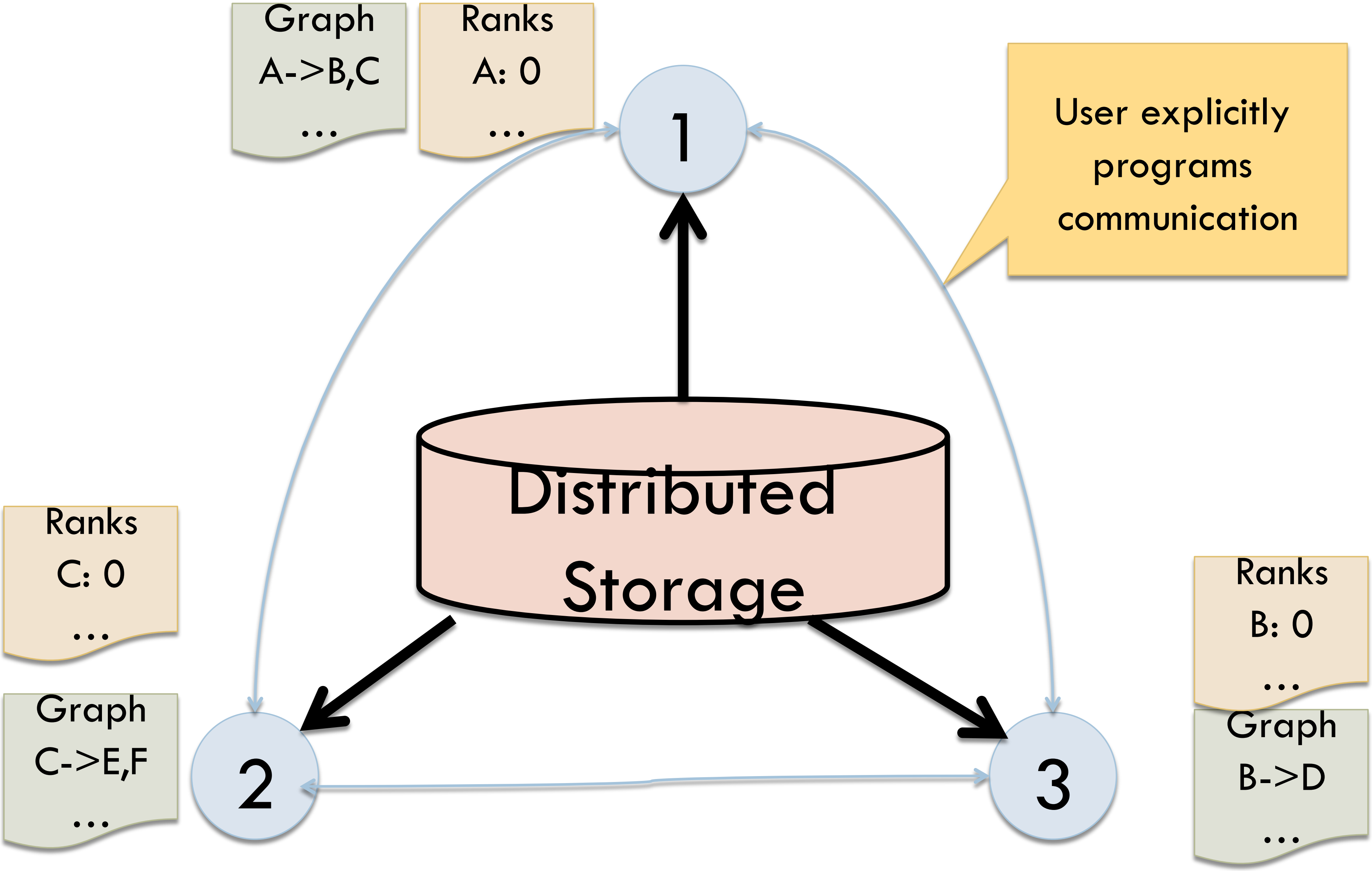


PageRank in MapReduce

- Data flow models do not expose global state.

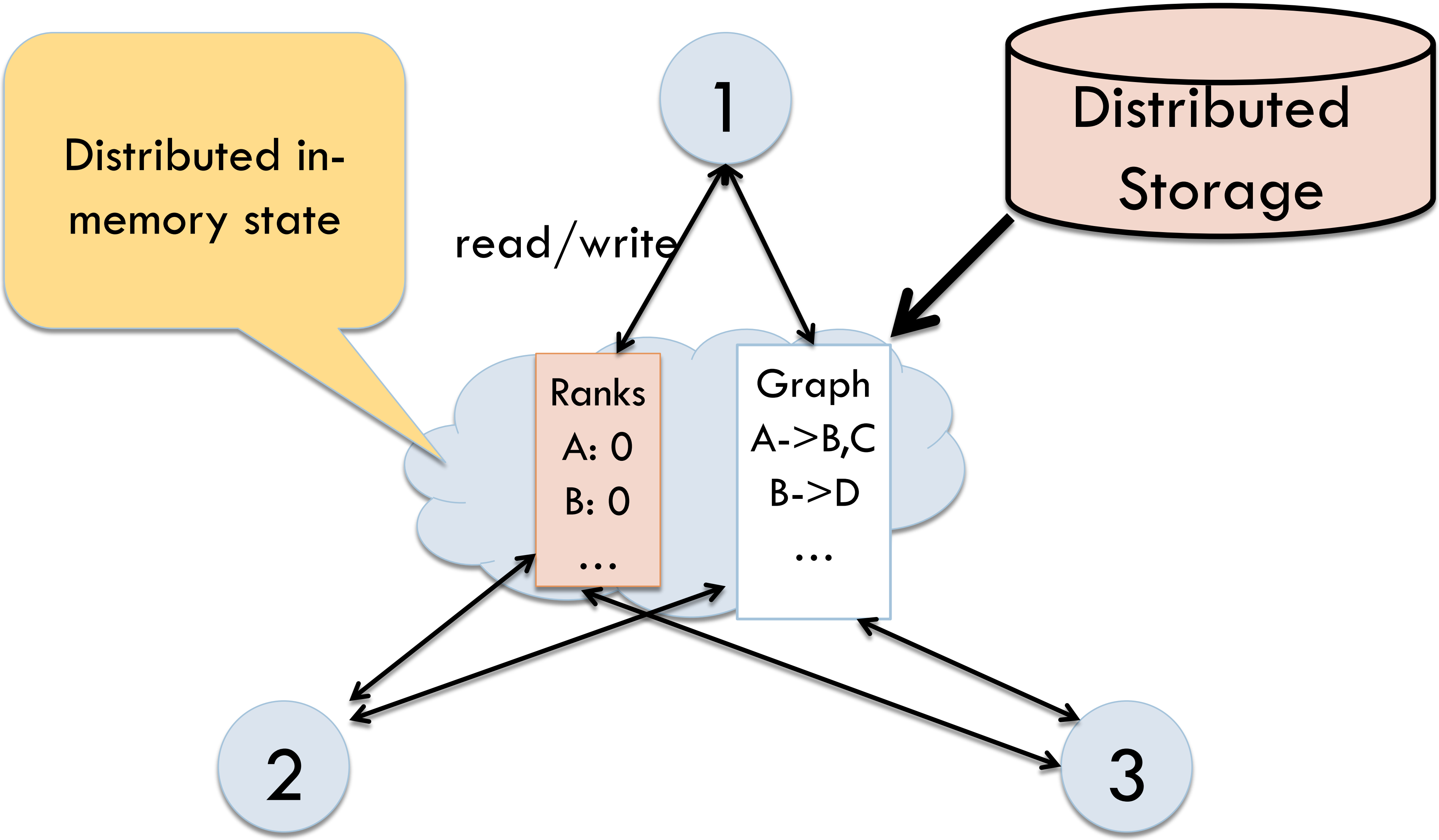


PageRank With MPI/RPC

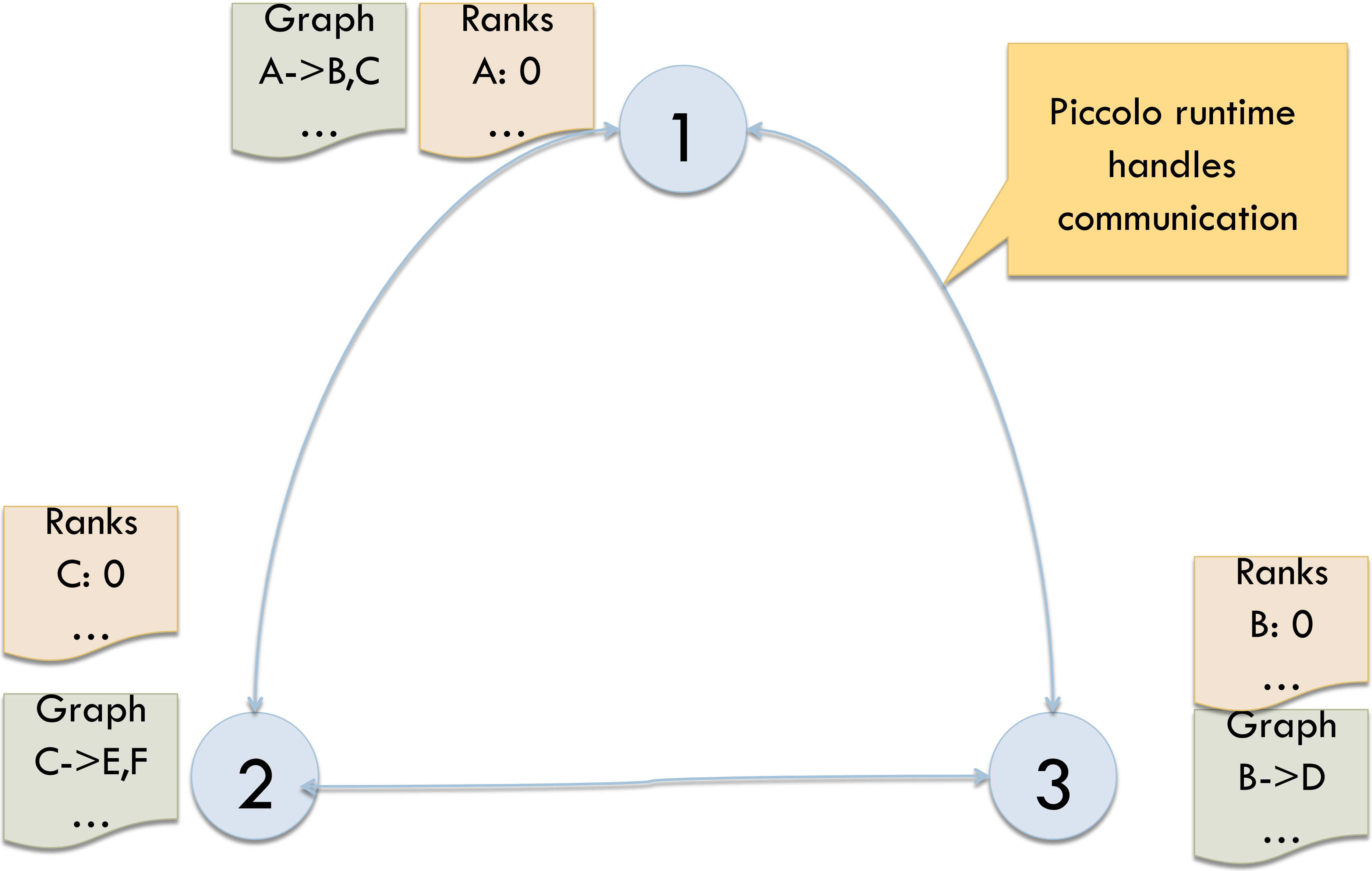


User explicitly programs communication

Piccolo's Goal: Distributed Shared State



Piccolo's Goal: Distributed Shared State



Graph
A->B,C
…

Ranks
A: 0
…

Ranks
C: 0
…

Graph
C->E,F
…

Ranks
B: 0
…

Graph
B->D
…

Piccolo runtime handles communication

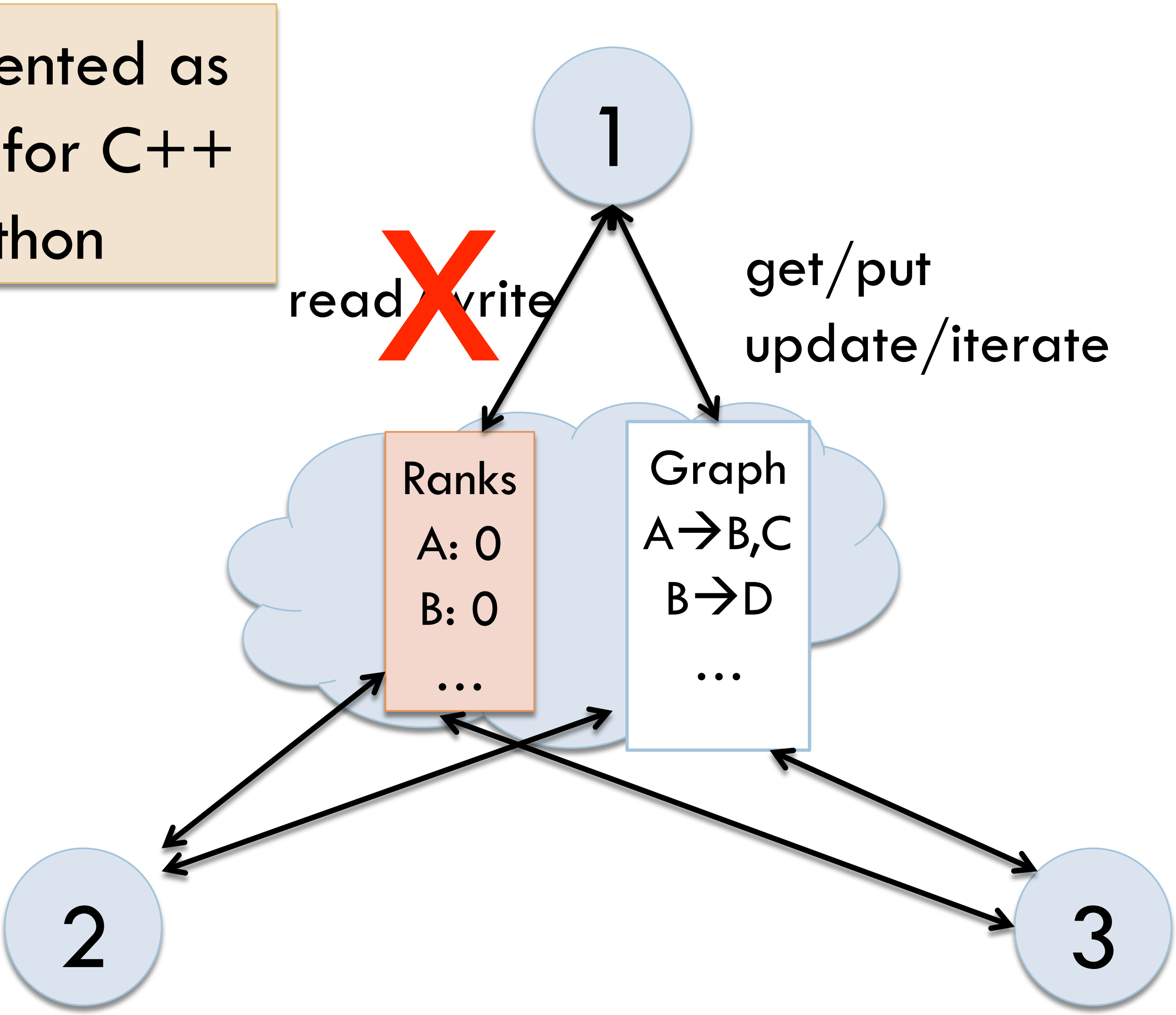
Ease of use
Performance

Talk outline

- Motivation
- Piccolo's Programming Model
- Runtime Scheduling
- Evaluation

Programming Model

Implemented as library for C++ and Python



Naïve PageRank with Piccolo

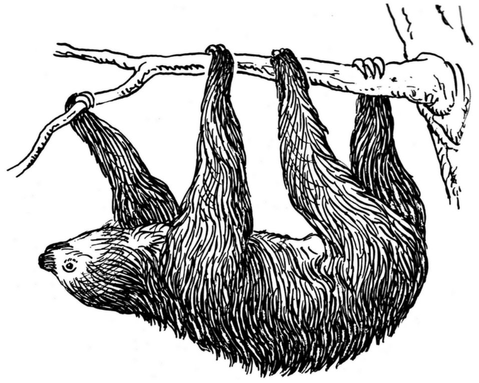
```
curr = Table(key=PageID, value=double)
next = Table(key=PageID, value=double)

def pr_kernel(graph, curr, next):
   i = my_instance
   n = len(graph)/NUM_MACHINES
   for s in graph[(i-1)*n:i*n]
      for t in s.out:
         next[t] += curr[s.id] / len(s.out)

def main():
   for i in range(50):
      launch_jobs(NUM_MACHINES, pr_kernel,
                  graph, curr, next)
      swap(curr, next)
      next.clear()
```

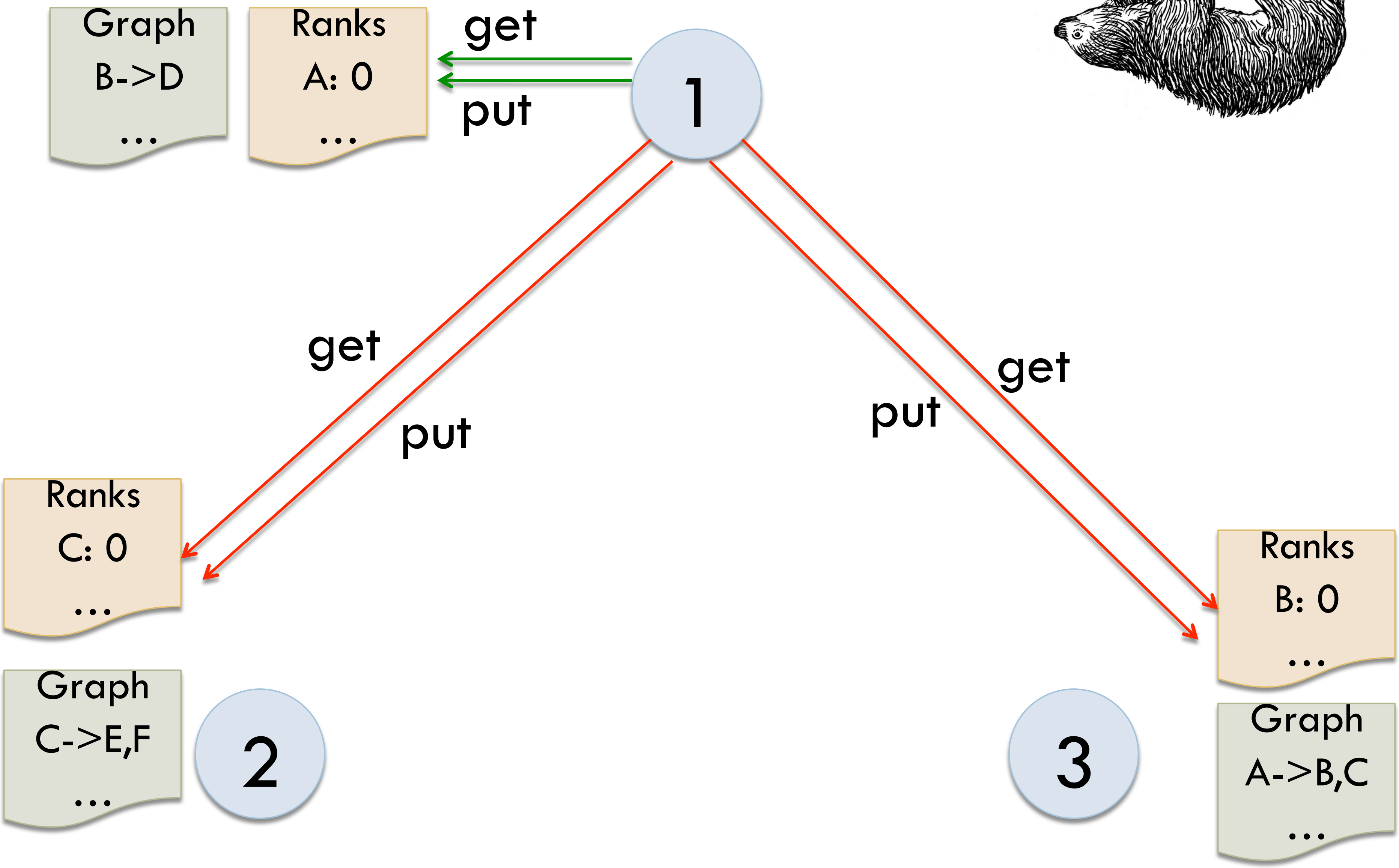
Jobs run by many machines

Controller launches jobs in parallel

Run by a single controller

Naïve PageRank is Slow





PageRank: Exploiting Locality

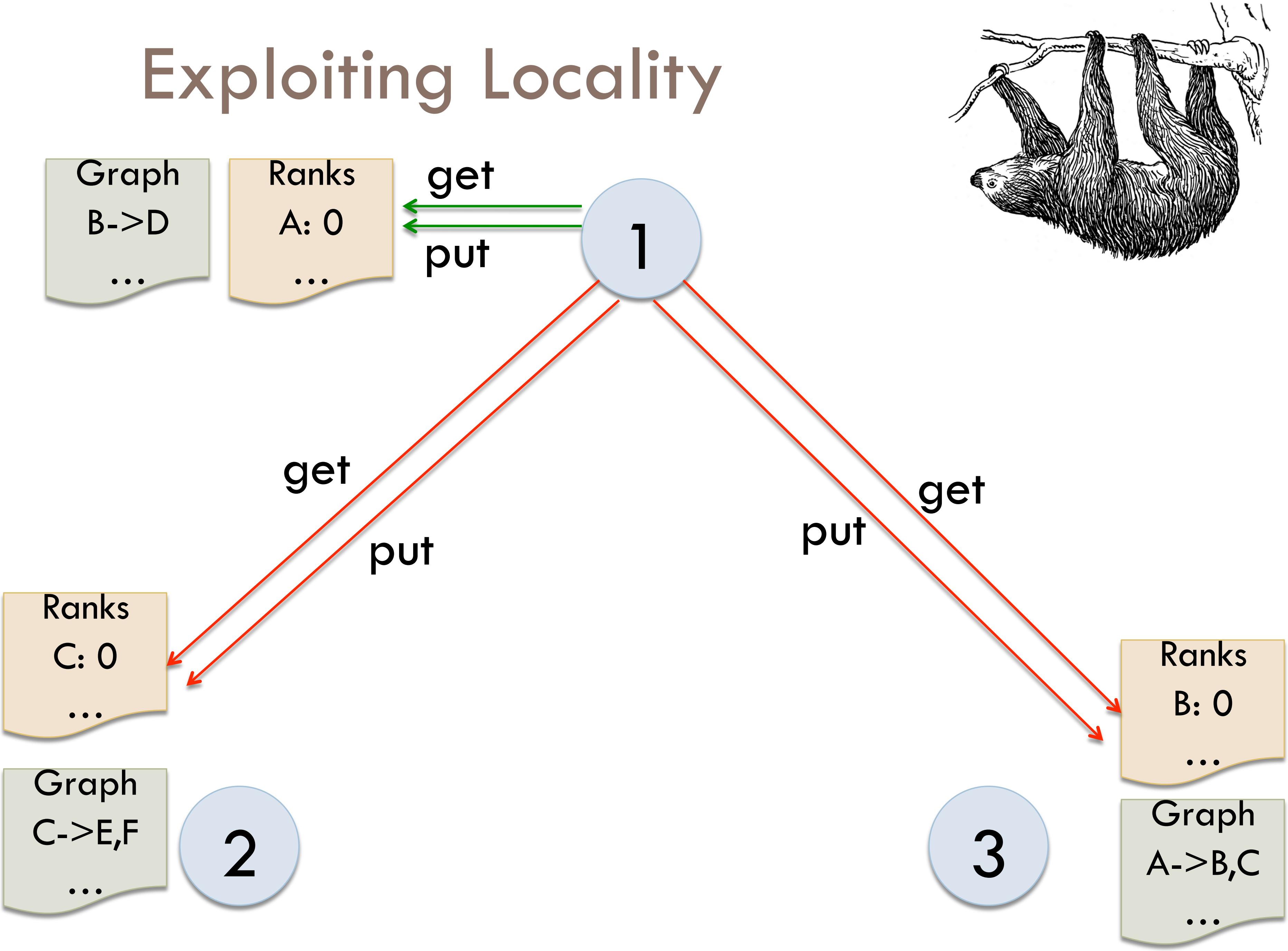
```
curr = Table(…,partitions=100,partition_by=site)
next = Table(…,partitions=100,partition_by=site)
group_tables(curr,next,graph)

def pr_kernel(graph, curr, next):
   for s in graph.get_iterator(my_instance)
      for t in s.out:
         next[t] += curr[s.id] / len(s.out)

def main():
   for i in range(50):
      launch_jobs(curr.num_partitions,
                    pr_kernel,
                    graph, curr, next,
                    locality=curr)
      swap(curr, next)
      next.clear()
```

Control table partitioning

Co-locate tables

Co-locate execution with table

Exploiting Locality



Graph
B->D
...

Ranks
A: 0
...

get

put

1

get

put

get

put

Ranks
C: 0
...

Graph
C->E,F
...

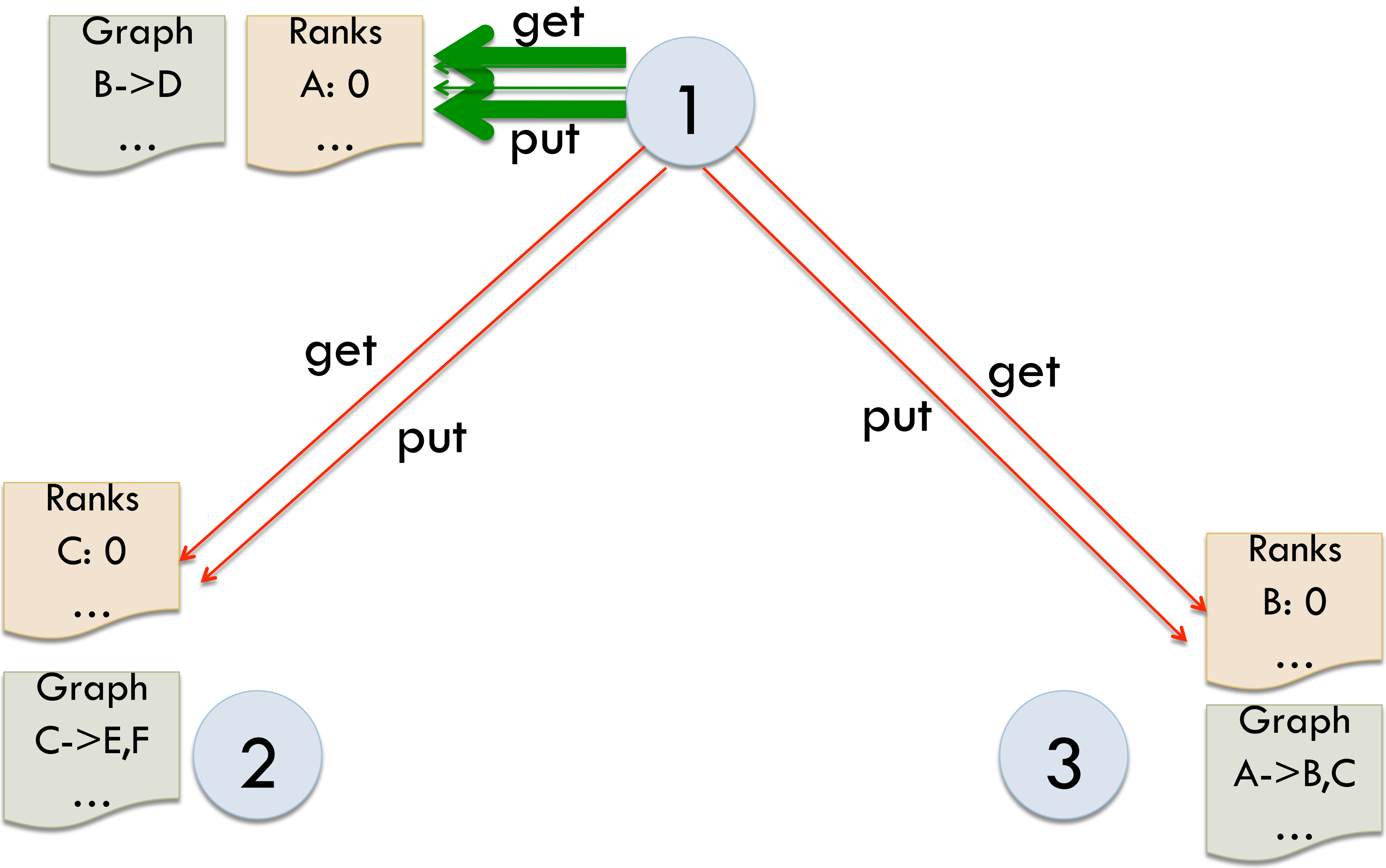
2

3

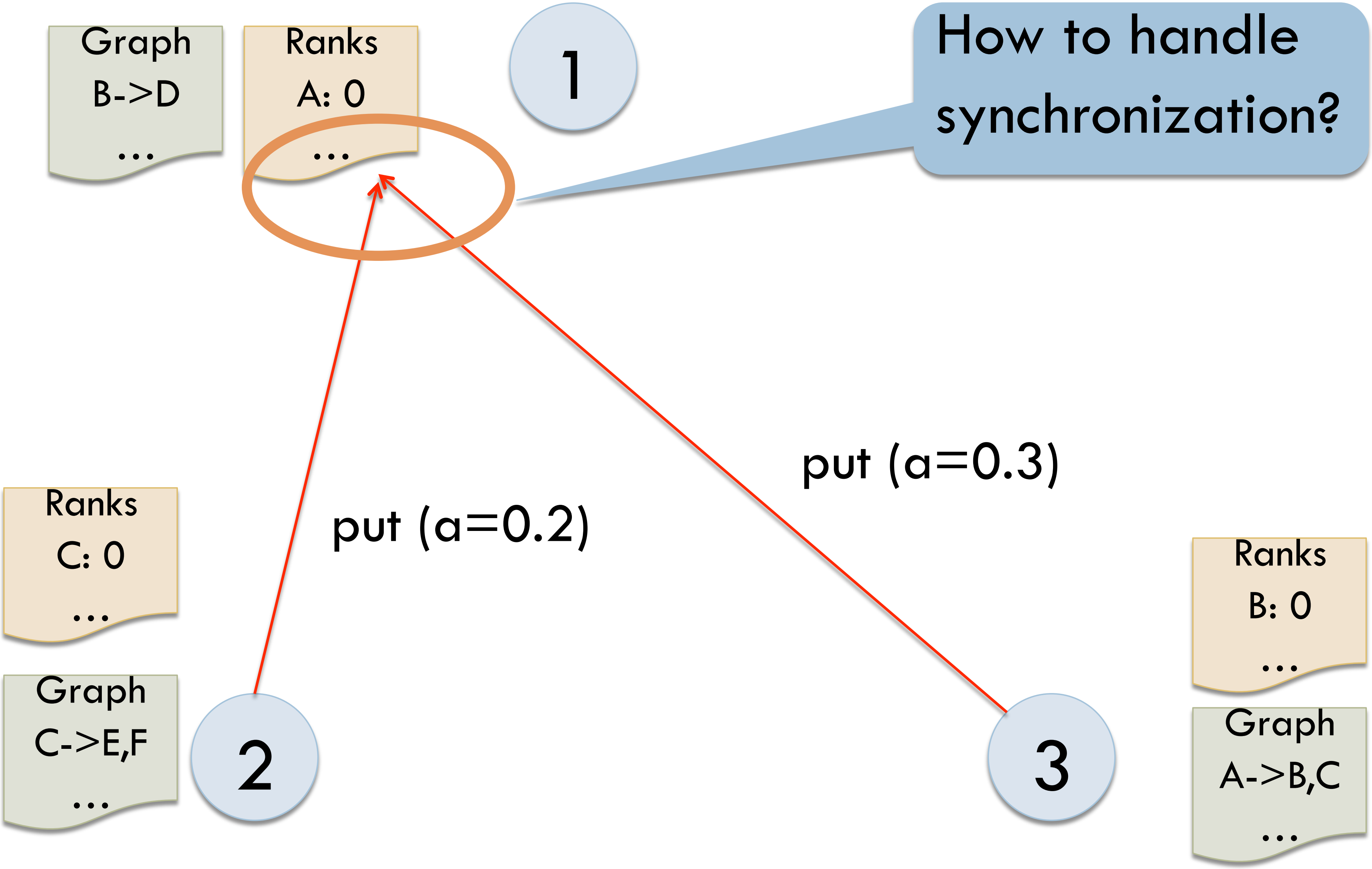
Ranks
B: 0
...

Graph
A->B,C
...

Exploiting Locality



Synchronization



How to handle synchronization?

Synchronization Primitives

- Avoid write conflicts with accumulation functions
  - `NewValue = Accum(OldValue, Update)`
    - *sum, product, min, max*

→Global barriers are sufficient

- Tables provide release consistency

PageRank: Efficient Synchronization

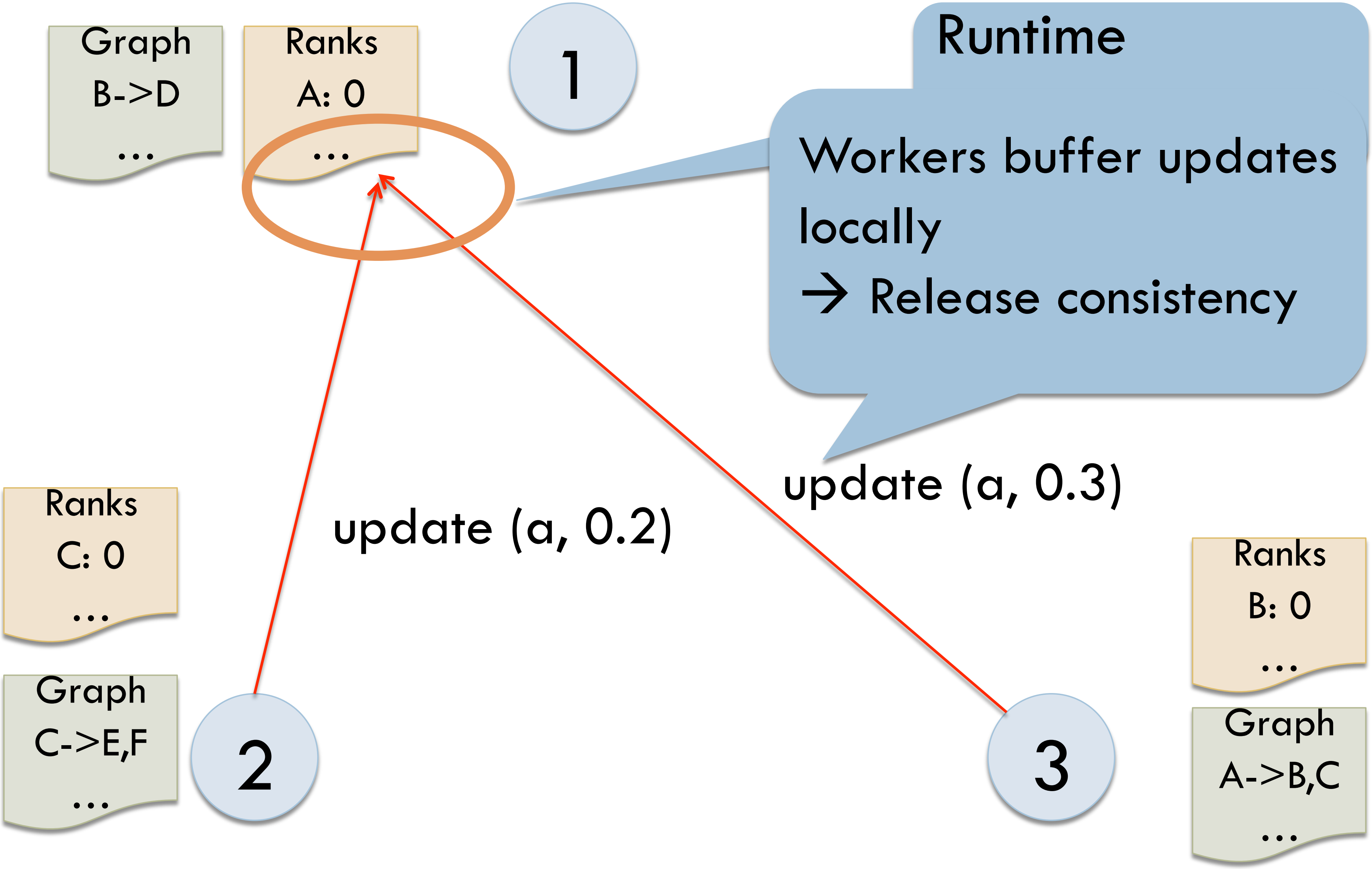
```
curr = Table(…,partition_by=site,accumulate=sum)
next = Table(…,partition_by=site,accumulate=sum)
group_tables(curr,next,graph)

def pr_kernel(graph, curr, next):
   for s in graph.get_iterator(my_instance)
      for t in s.out:
         next.update(t, curr.get(s.id)/len(s.out))

def main():
   for i in range(50):
      handle = launch_jobs(curr.num_partitions,
                             pr_kernel,
                             graph, curr, next,
                             locality=curr)
      barrier(handle)
      swap(curr, next)
      next.clear()
```

Accumulation via sum

Update invokes accumulation function

Explicitly wait between iterations

Efficient Synchronization



Runtime

Workers buffer updates locally

→ Release consistency

Oh Noez!
Teh Servr Crashd!
ICANHASCHEEZBURGER.COM

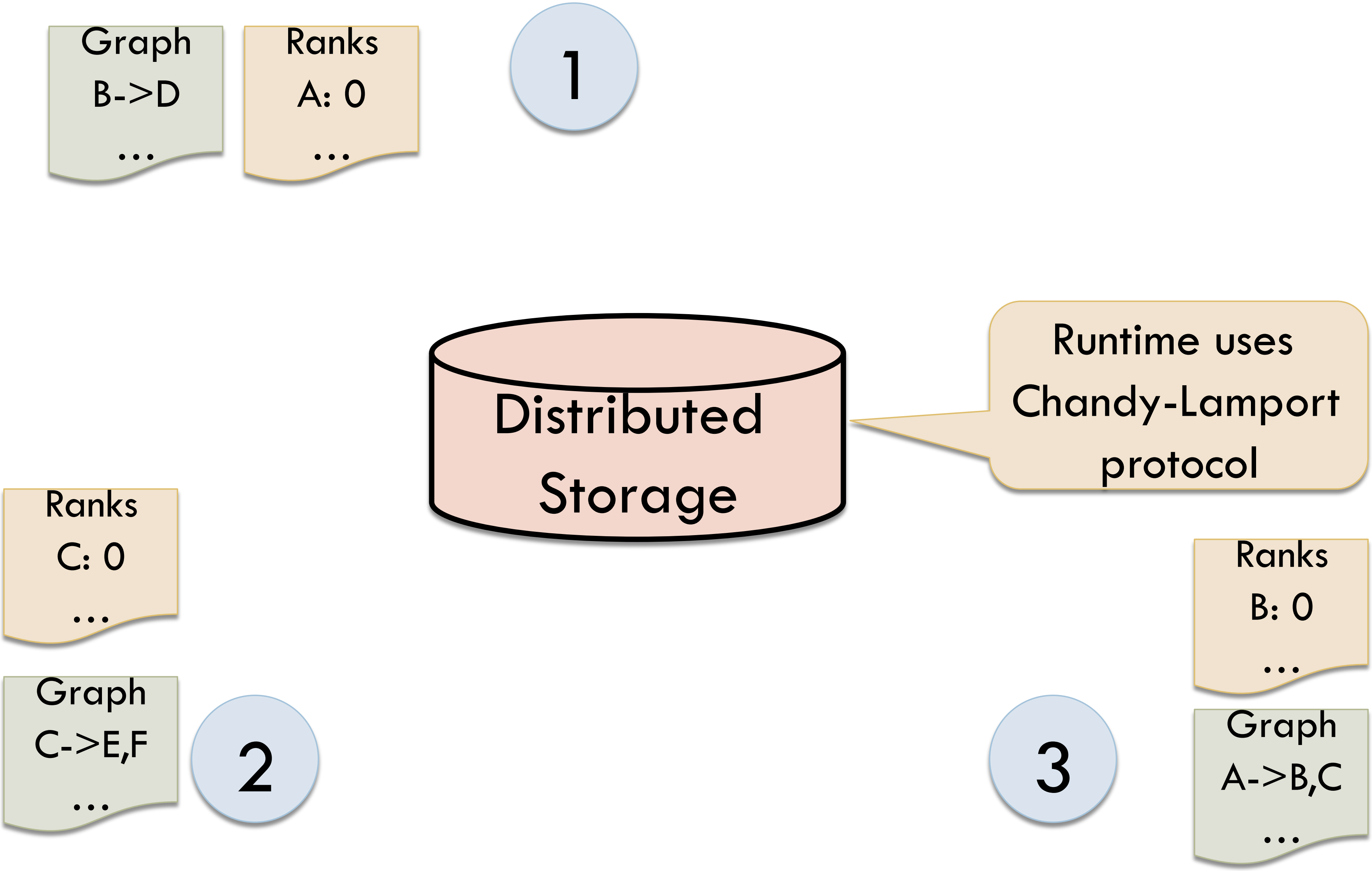
PageRank with Checkpointing

```
curr = Table(…,partition_by=site,accumulate=sum)
next = Table(…,partition_by=site,accumulate=sum)
group_tables(curr,next)
def pr_kernel(graph, curr, next):
   for node in graph.get_iterator(my_instance)
      for t in s.out:
         next.update(t,curr.get(s.id)/len(s.out))

def main():
   curr, userdata = restore()
   last = userdata.get(‘iter’, 0)
   for i in range(last,50):
      handle = launch_jobs(curr.num_partitions, pr_kernel,
                           graph, curr, next,
                           locality=curr)
      cp_barrier(handle, tables=(next), userdata={‘iter’:i})
      swap(curr, next)
      next.clear()
```

Restore previous computation

User decides which tables to checkpoint and when

Recovery via Checkpointing

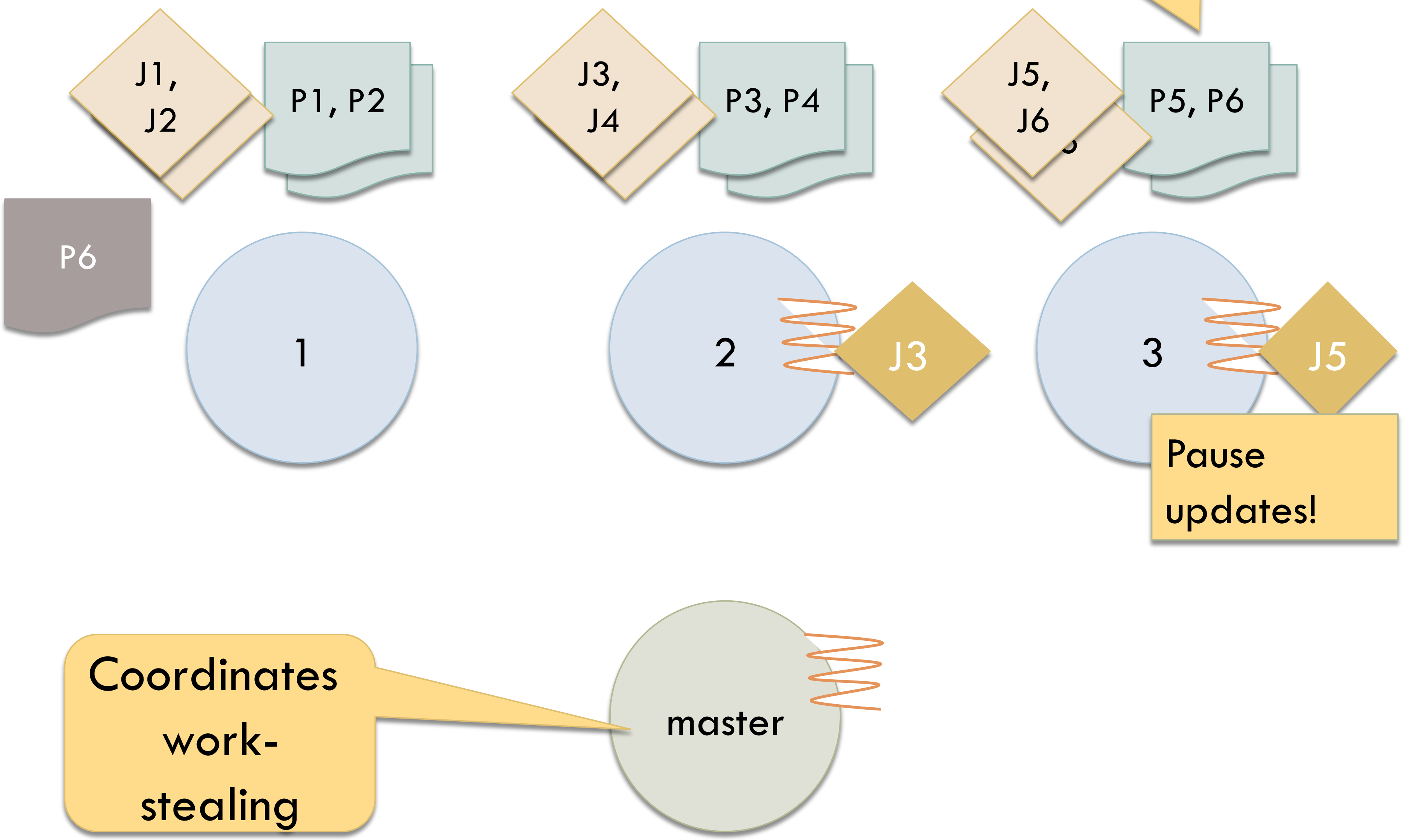


Runtime uses Chandy-Lamport protocol

Talk Outline

- □ Motivation
- □ Piccolo's Programming Model
- □ Runtime Scheduling
- □ Evaluation

Load Balancing



Other workers are updating P6!

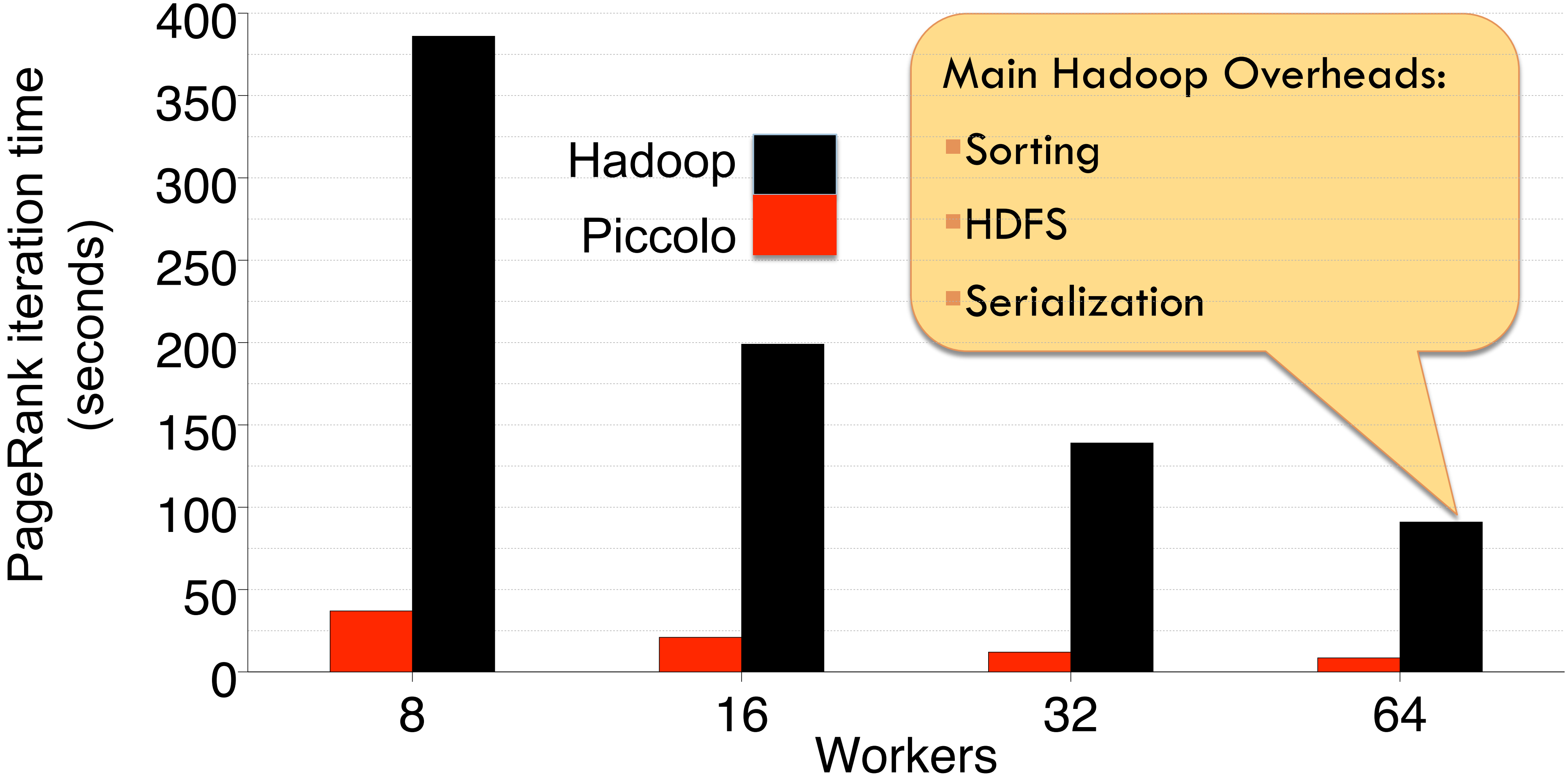
Pause updates!

Coordinates work-stealing

Talk Outline

- □ Motivation
- □ Piccolo's Programming Model
- □ System Design
- □ Evaluation

Piccolo is Fast



Main Hadoop Overheads:

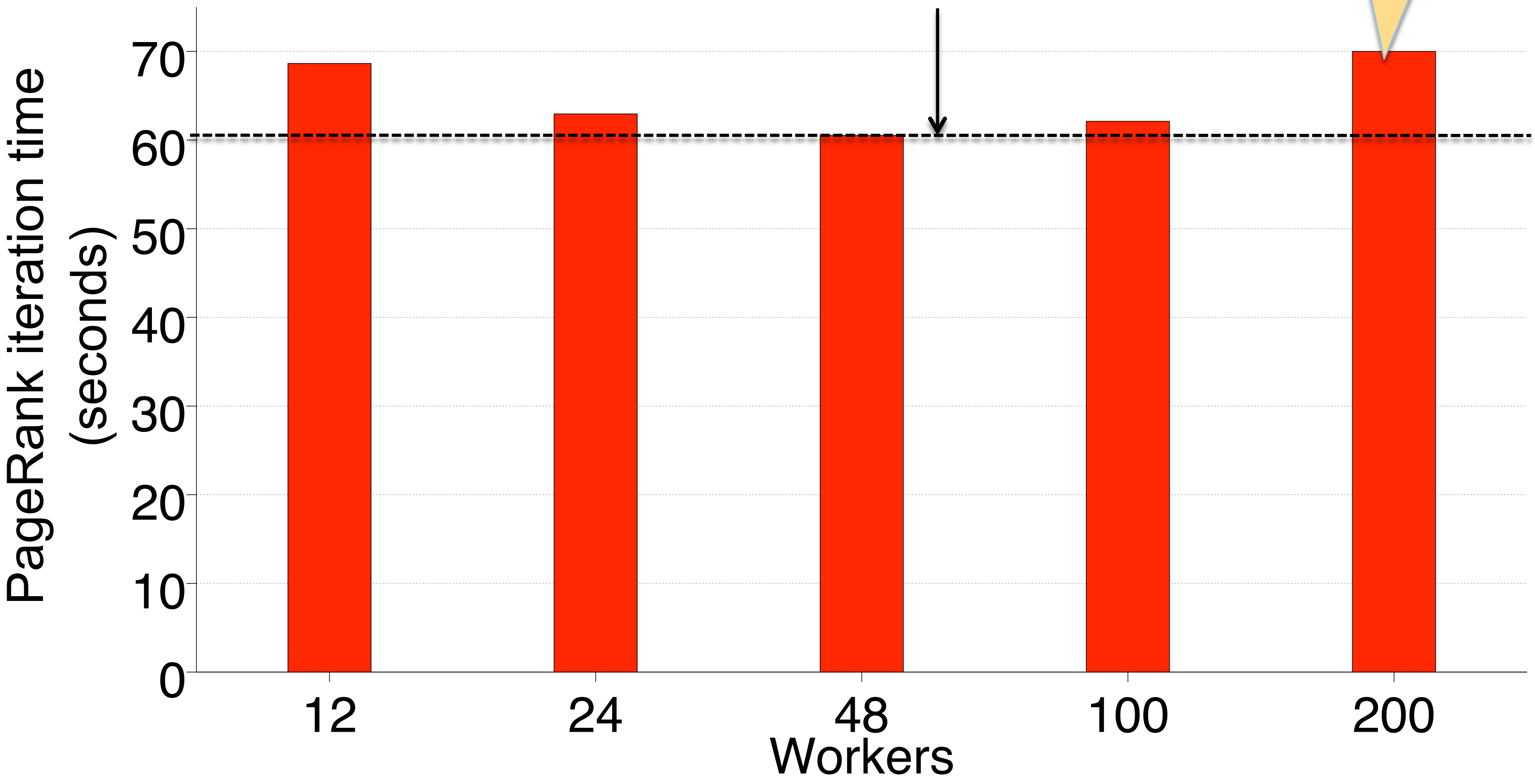
- Sorting
- HDFS
- Serialization

- NYU cluster, 12 nodes, 64 cores
- 100M-page graph

Piccolo Scales Well

1 billion page graph

ideal



- EC2 Cluster - linearly scaled input graph

Other applications

- Iterative Applications
  - N-Body Simulation
  - Matrix Multiply
- Asynchronous Applications
  - Distributed web crawler

No straightforward Hadoop implementation

Related Work

- Data flow
  - MapReduce, Dryad
- Tuple Spaces
  - Linda, JavaSpaces
- Distributed Shared Memory
  - CRL, TreadMarks, Munin, Ivy
  - UPC, Titanium

Conclusion

- Distributed shared table model
- User-specified policies provide for
  - Effective use of locality
  - Efficient synchronization
  - Robust failure recovery

Gratuitous Cat Picture

I can haz kwestions?

Try it out:

piccolo.news.cs.nyu.edu