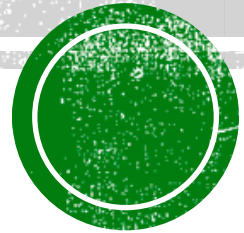
Seminar: Massive-Scale Graph Analysis,

Summer Semester 2015

Large-Scale Graph Engines

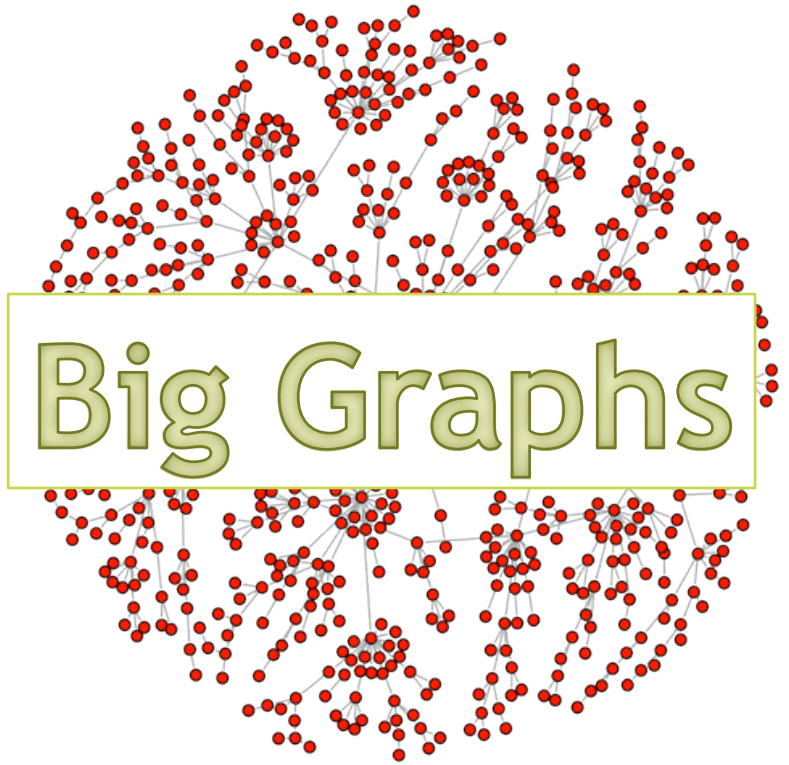


Margarita Salyaeva

s8masaly@stud.uni-saarland.de

12 June 2015

Big Data



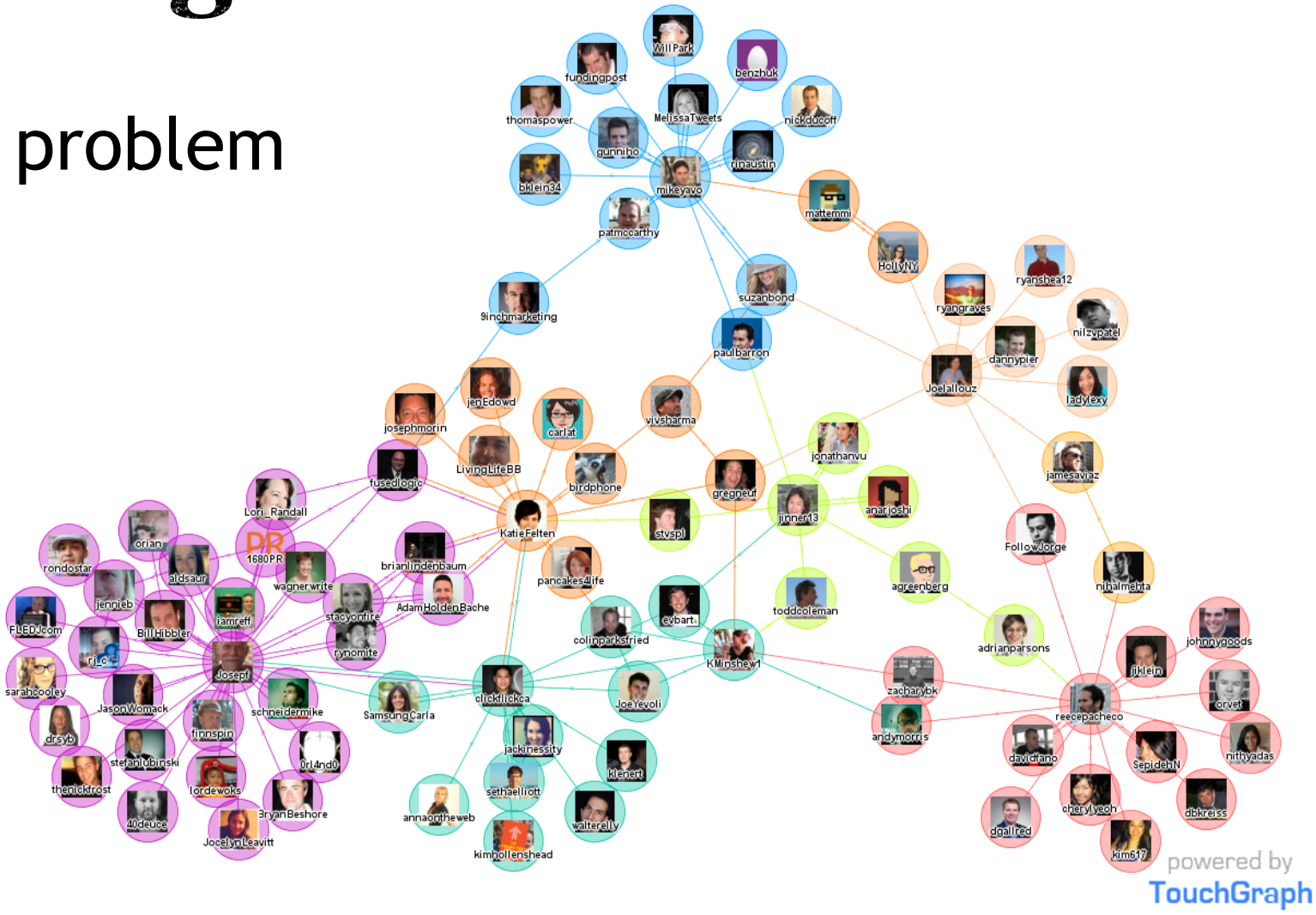
One of the systems to analyze Big Graphs

GraphChi

Process Large-Scale Graphs on just a PC

Agenda

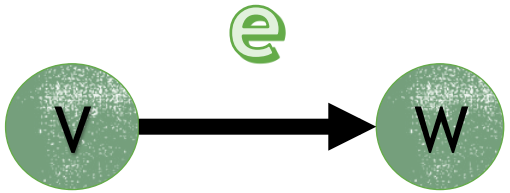
- Introduction to the problem
- Challenges
- GraphChi
- Experiments
- Limitations
- Conclusion



Computational Model

Graph G = (V, E)

V-set of vertices, E – set of edges



directed edges: e = (source, destination)
each edge and vertex **associated with a value**
Graph structure, vertex and edge **values can be modified**

Computational Model

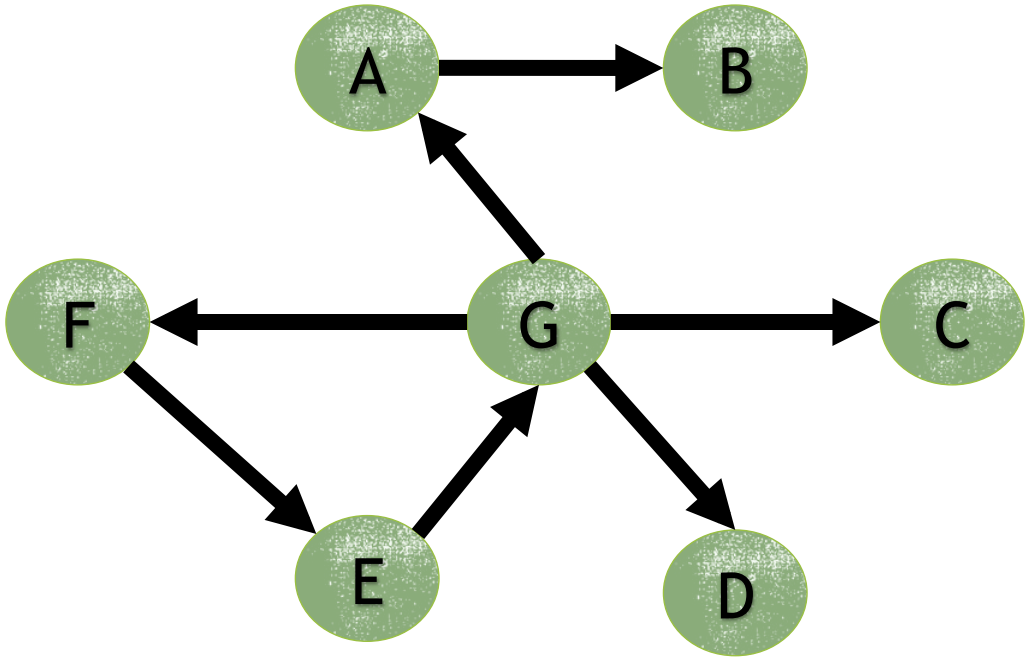
Vertex - centric Model

def myFun(vertex):

***

modify vertex value

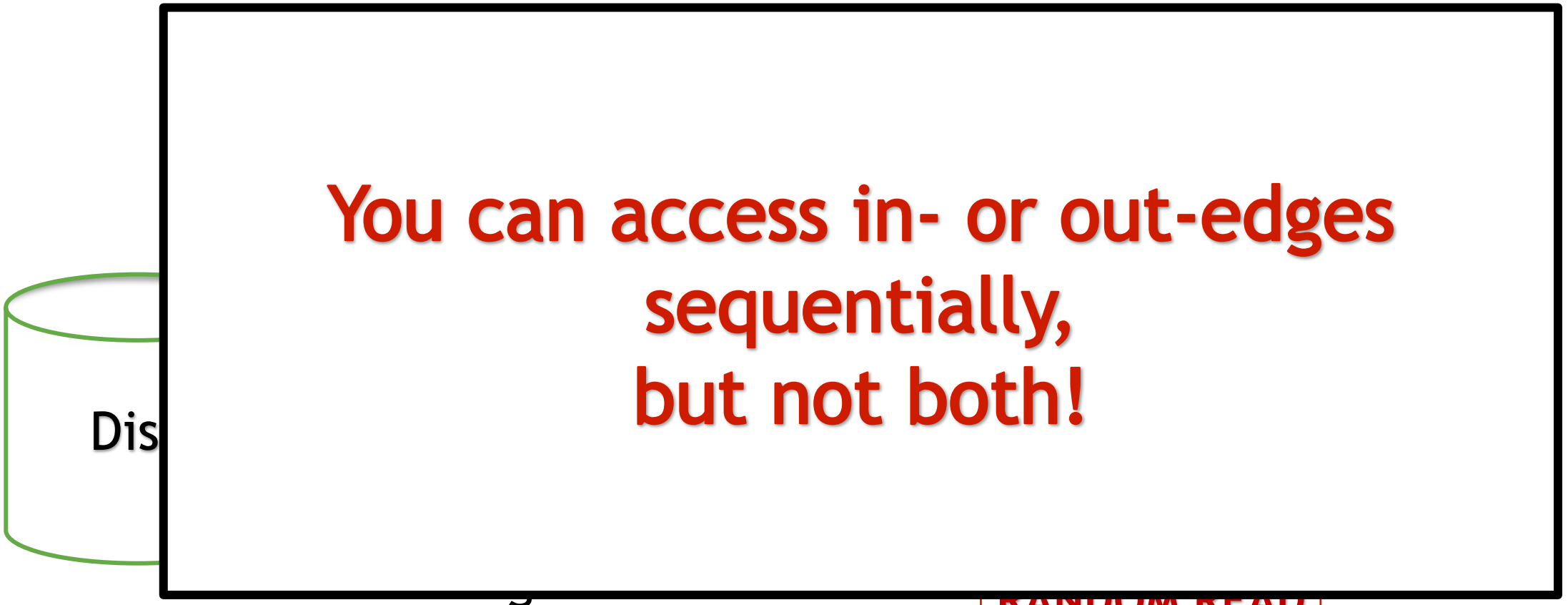
modify values of incident edges



Challenges

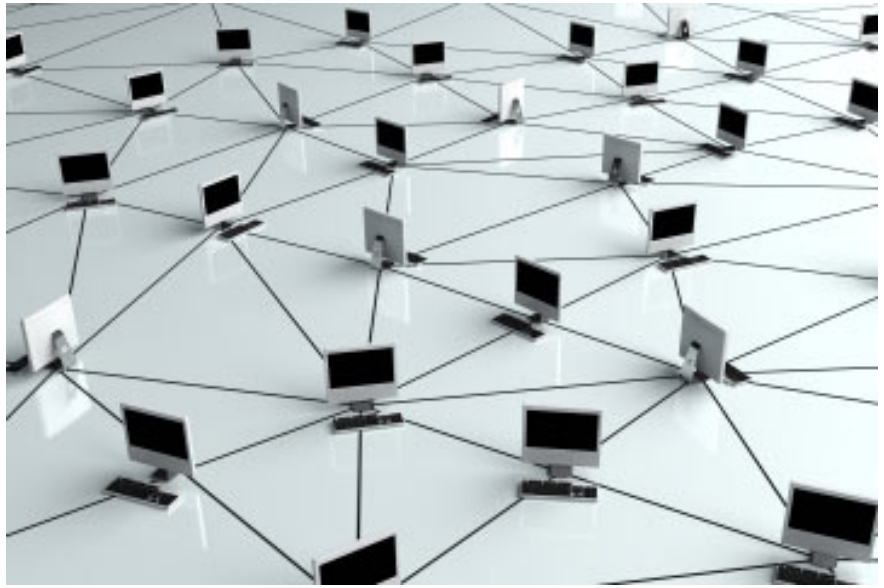
- Large amount of memory
  - Billions of edges and vertices
- Large-scale computation
- Processing evolving graphs
  - How to support addition and removal of edges and vertices?
- Random data access

Random Access Problem



Distributed systems. Overheads

- Partitioning graphs
  - Balanced graph cuts
  - Minimize communication between nodes
- Managing clusters
- Fault tolerance
- Unpredictable performance
- Debugging is hard
- Optimizing algorithms is hard
- Infrastructure costs



Is it possible on Just a PC?

- Graph structure, edge values and vertex values don’t fit into memory
- Edges and their values of any single vertex fit into memory

Advantages of single-machine systems

- Programmer productivity
  - Global state
  - Debuggers
- Inexpensive to install, administer, less power
- Scalability
  - Use cluster of single-machine systems to solve many tasks in parallel

Possible Solutions

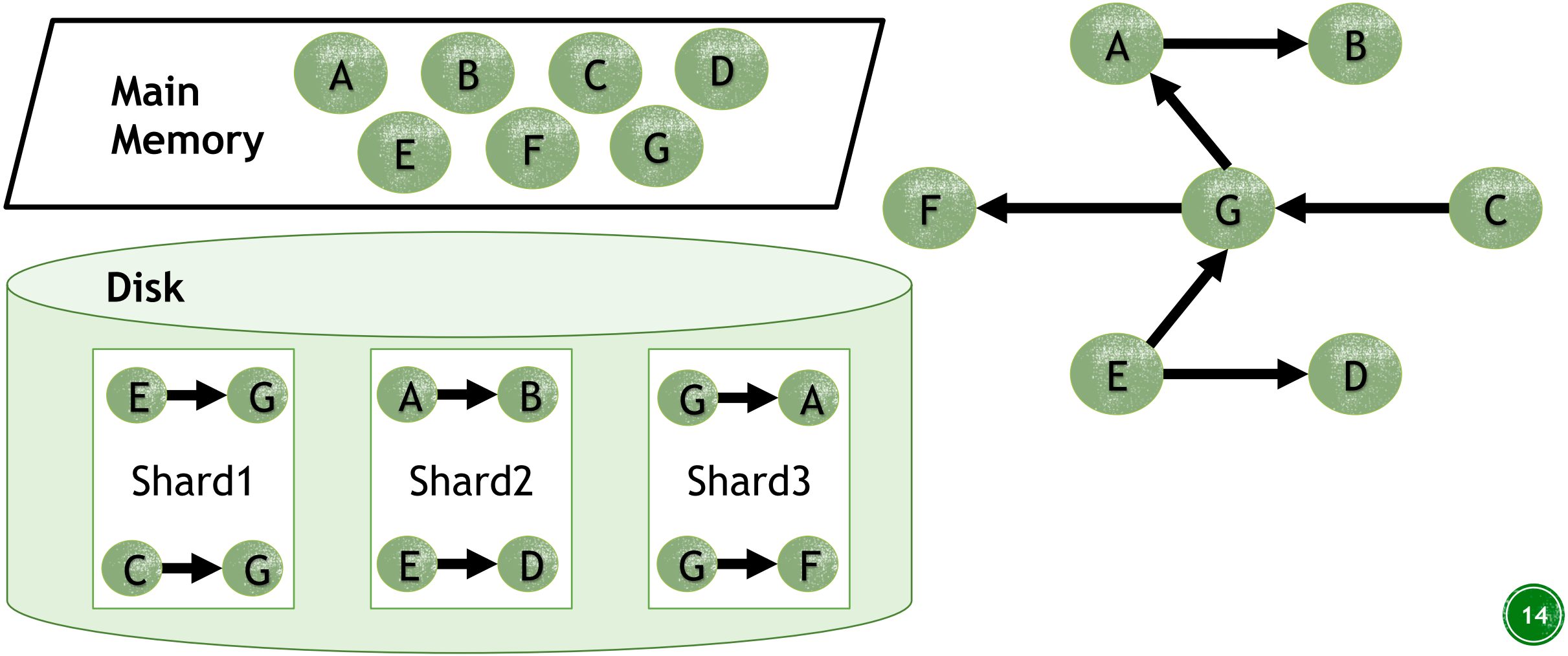
SSD as a memory-extension	Exploit Locality
Too many small objects	Locality is limited Optimizing is costly

Caching	Bulk-Synchronous Processing	Graph Compression
Unpredictable performance Cache policy	High costs of synchronization step Stores 2 versions of all values	Associated values do not compress well, and are mutated

GraphChi

Challenges	GraphChi's Solutions
Large amount of memory	Separate graph structure from edge values
	Compress graph structure
	Vertices are in Main Memory
Large-scale computation	Single-machine
Random Data Access	Organize edges to facilitate sequential scan
	Divide edges into shards
	Each shard can fit into Main Memory
	Parallel Sliding Windows technique
Processing evolving graphs	Parallel Sliding Windows technique

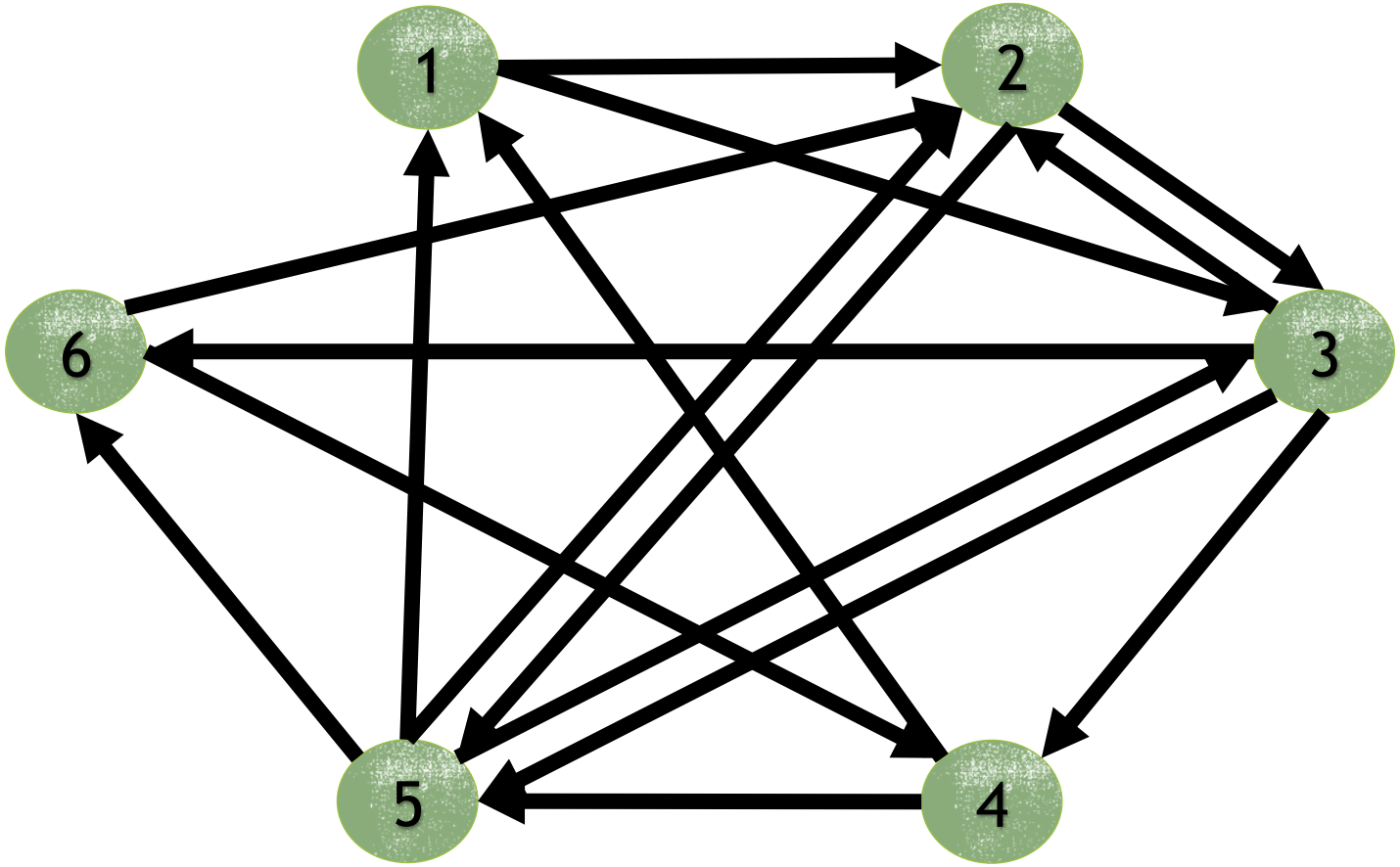
GraphChi. Basic idea



Parallel Sliding Windows. 3 Steps

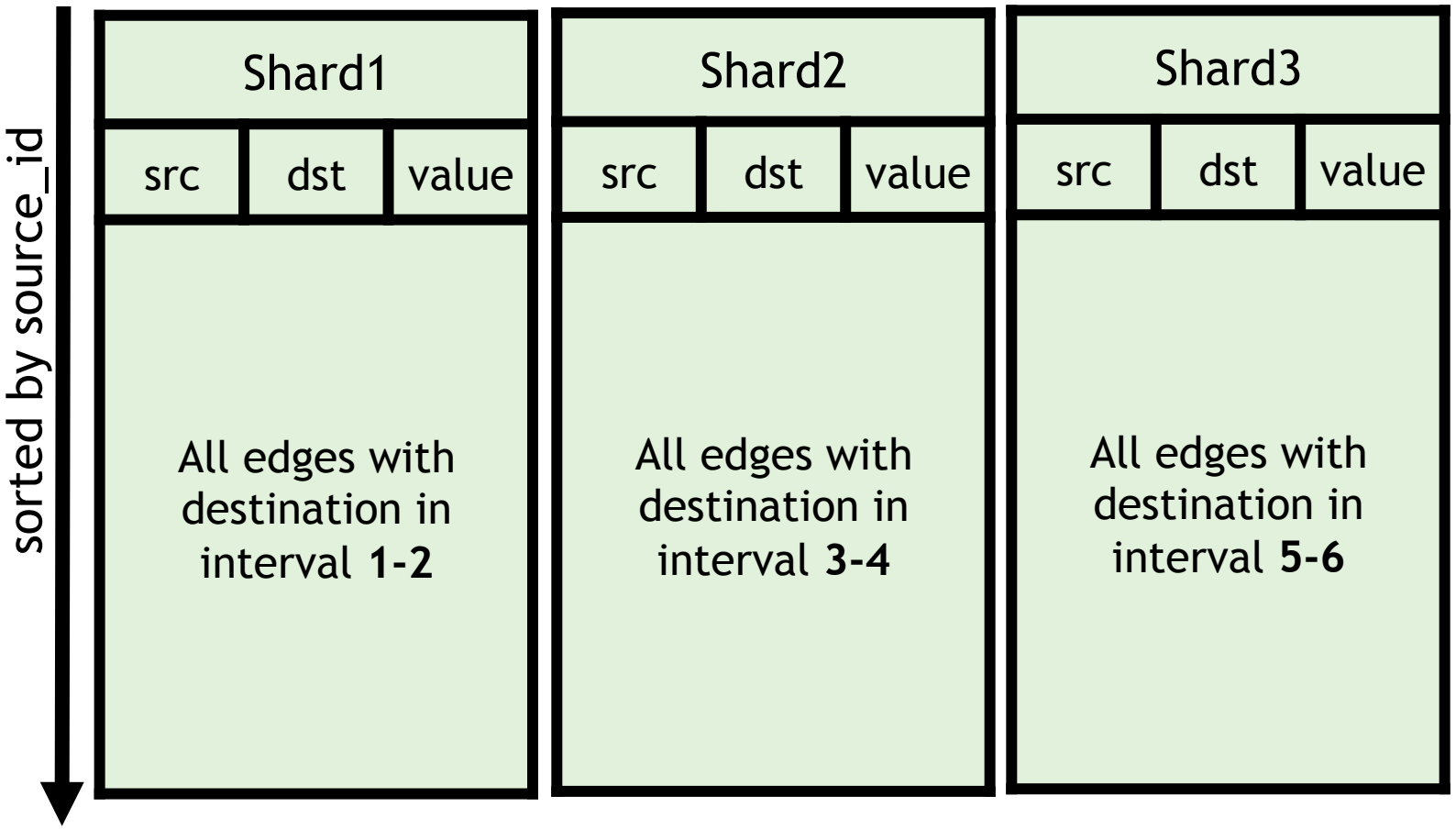
- Loading the Graph
- Parallel Updates
- Updating Graph to Disk

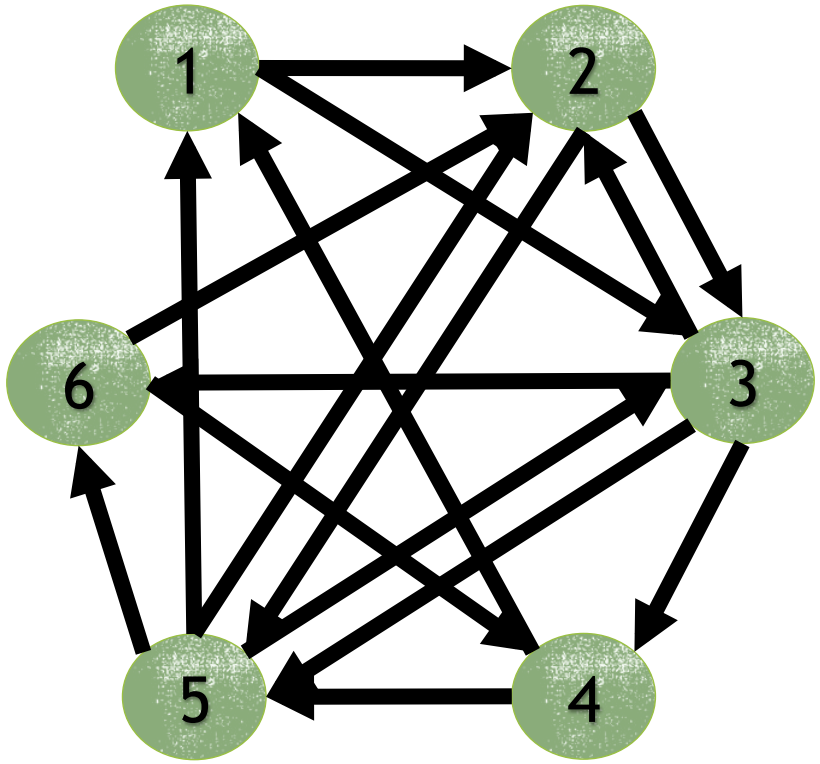
Parallel Sliding Windows.
Toy Example



PSW. Loading

- 3 intervals: 1-2, 3-4, 5-6



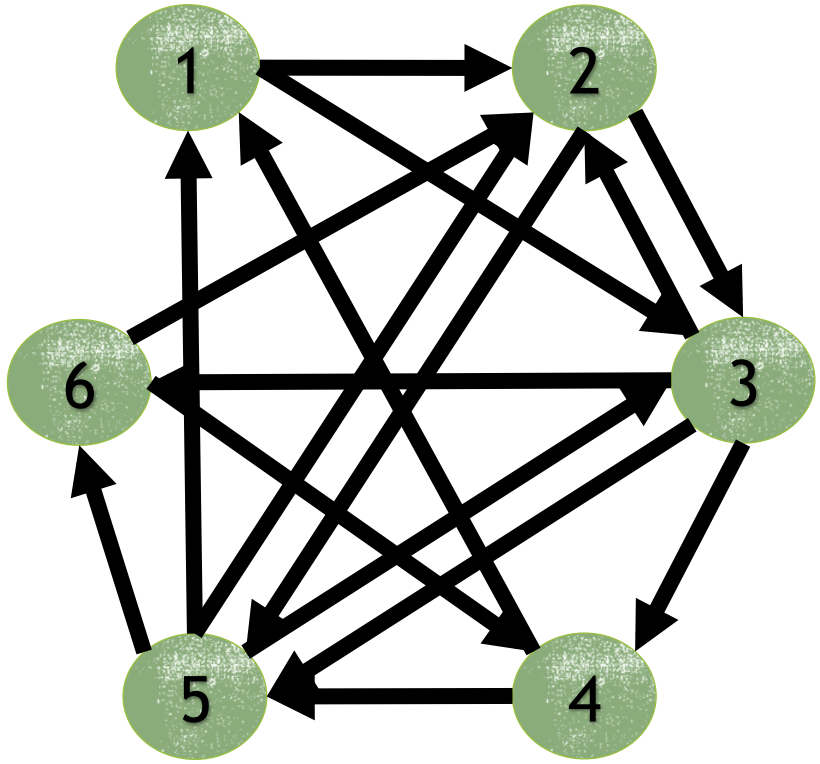


PSW. Loading

Shard 1		
src	dst	value
1	2	0,3
3	2	0,2
4	1	1,40
5	1	0,5
5	2	0,6
6	2	0,8

Shard 2		
src	dst	value
1	3	0,4
2	3	0,3
3	4	0,8
5	3	0,2
6	4	1,9

Shard 3		
src	dst	value
2	5	0,6
3	5	0,9
3	6	1,2
4	5	0,3
5	6	1,1



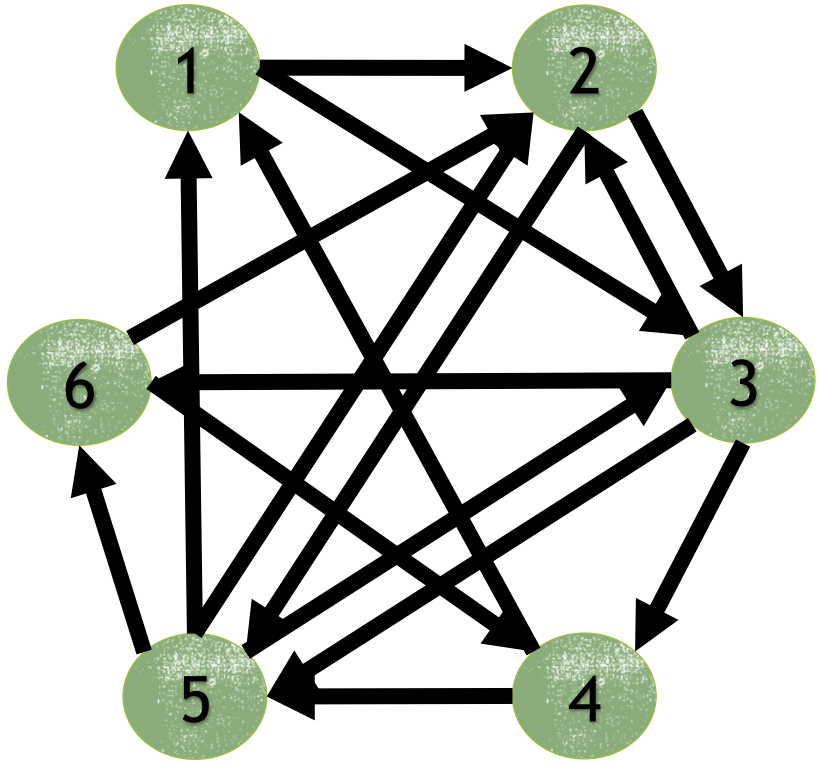
PSW. Load and Update

- Interval 1-2

Shard 1			Shard 2			Shard 3		
src	dst	value	src	dst	value	src	dst	value
1	2	0,3	1	3	0,4	2	5	0,6
3	2	0,2	2	3	0,3	3	5	0,9
4	1	1,40	3	4	0,8	3	6	1,2
5	1	0,5	5	3	0,2	4	5	0,3
5	2	0,6	6	4	1,9	5	6	1,1
6	2	0,8						

Sliding shards

Memory shard



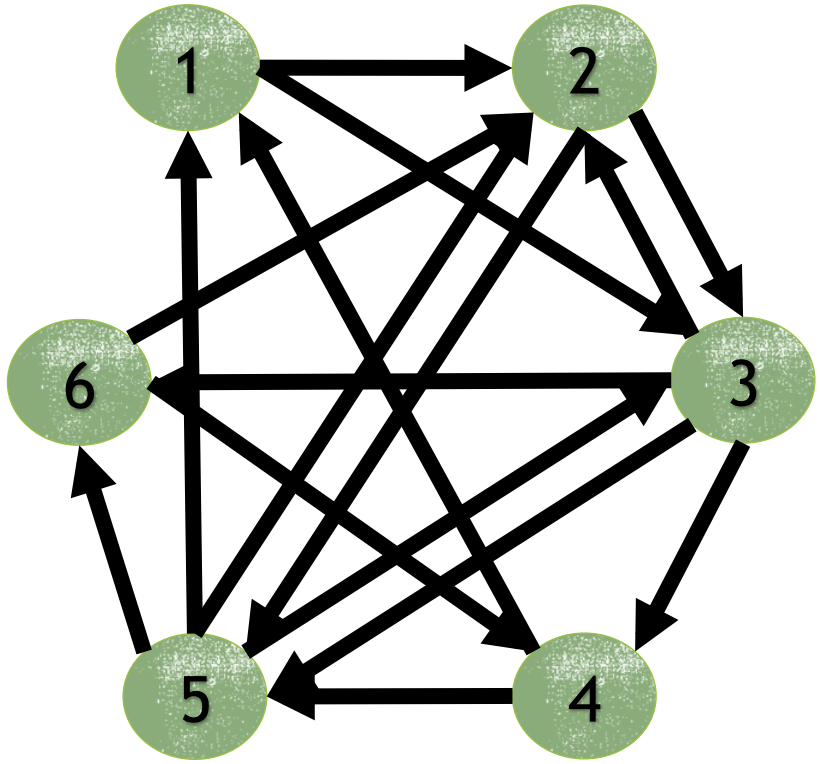
PSW. Load and Update

- Interval 1-2

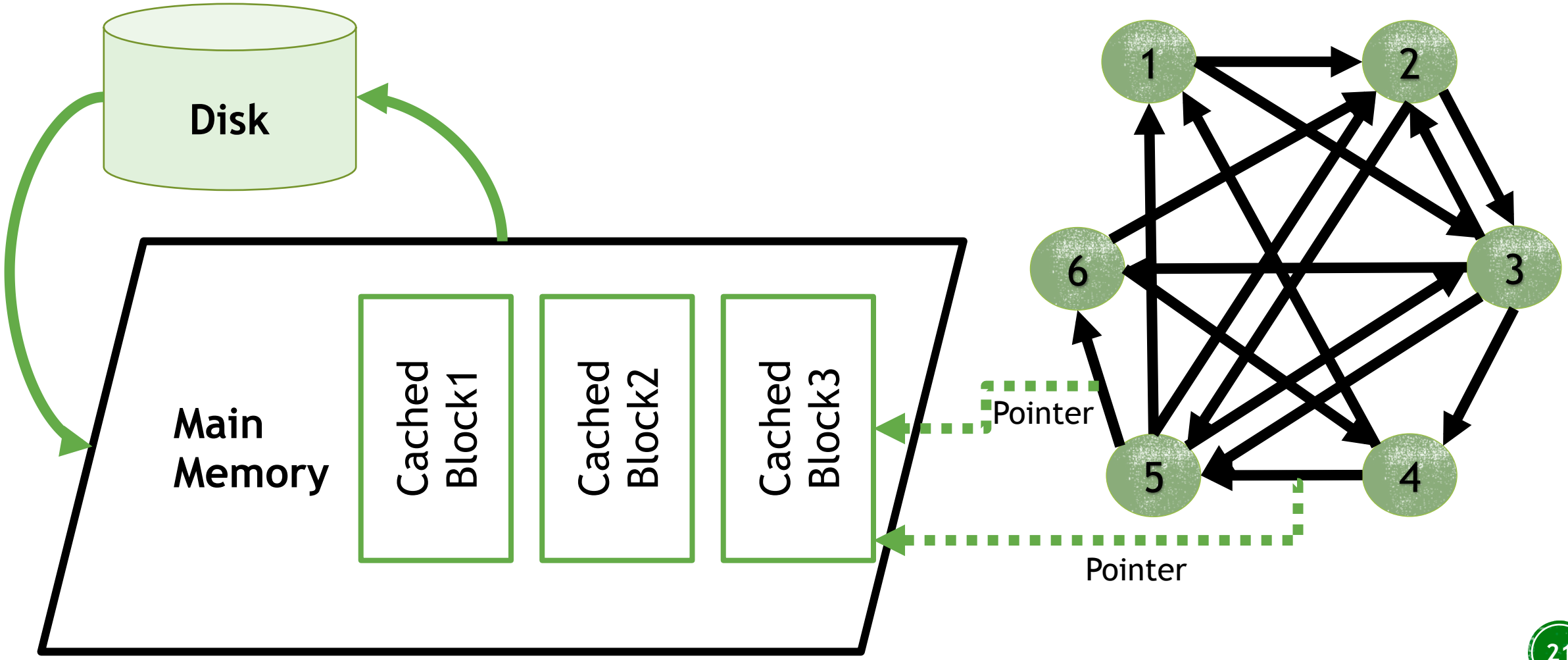
Shard 1			Shard 2			Shard 3		
src	dst	value	src	dst	value	src	dst	value
1	2	0,273	1	3	0,364	2	5	0,55
3	2	0,22	2	3	0,273	3	5	0,9
4	1	1,54	3	4	0,8	3	6	1,2
5	1	0,55	5	3	0,2	4	5	0,3
5	2	0,66	6	4	1,9	5	6	1,1
6	2	0,88						

Memory shard

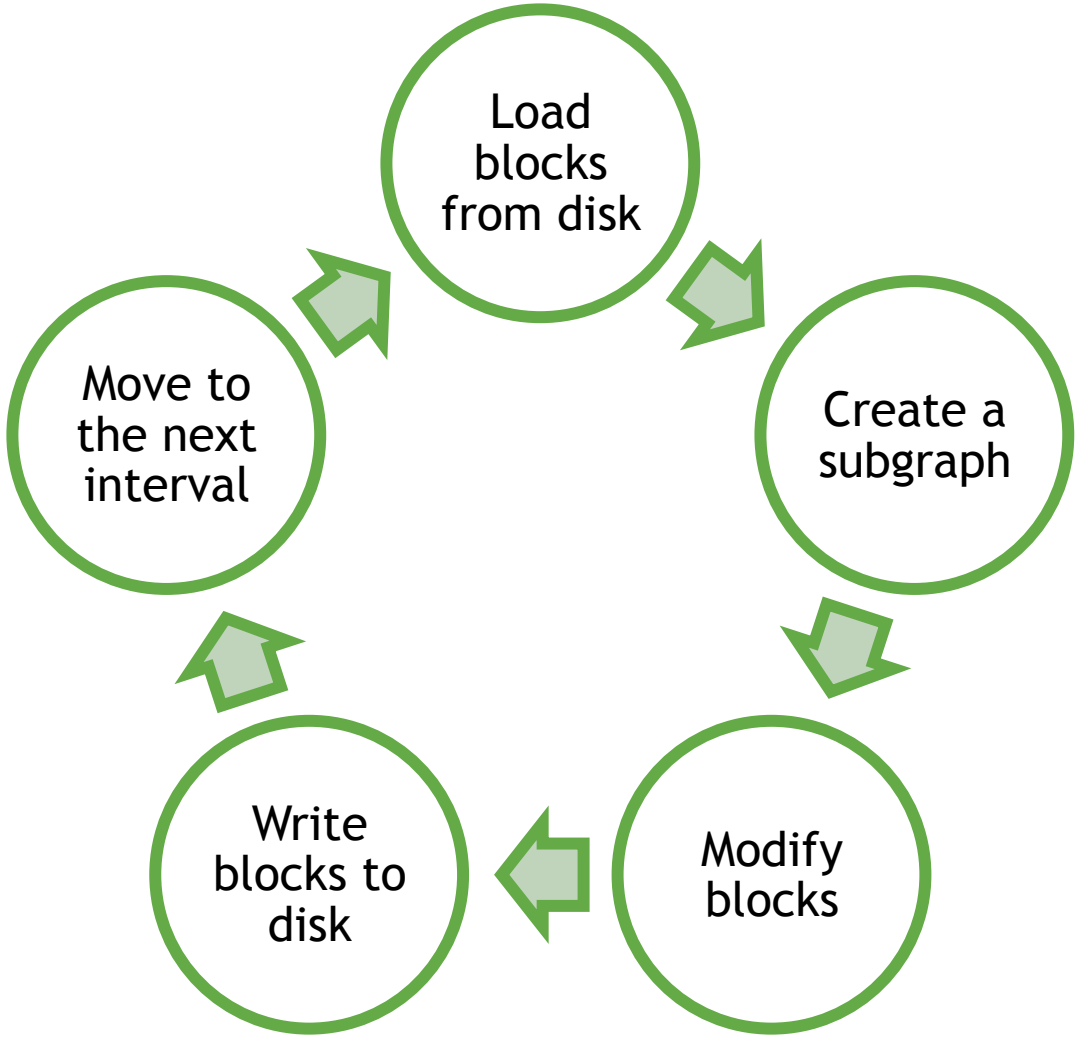
Sliding shards



PSW. Updating to Disk



PSW. Updating Graph to Disk

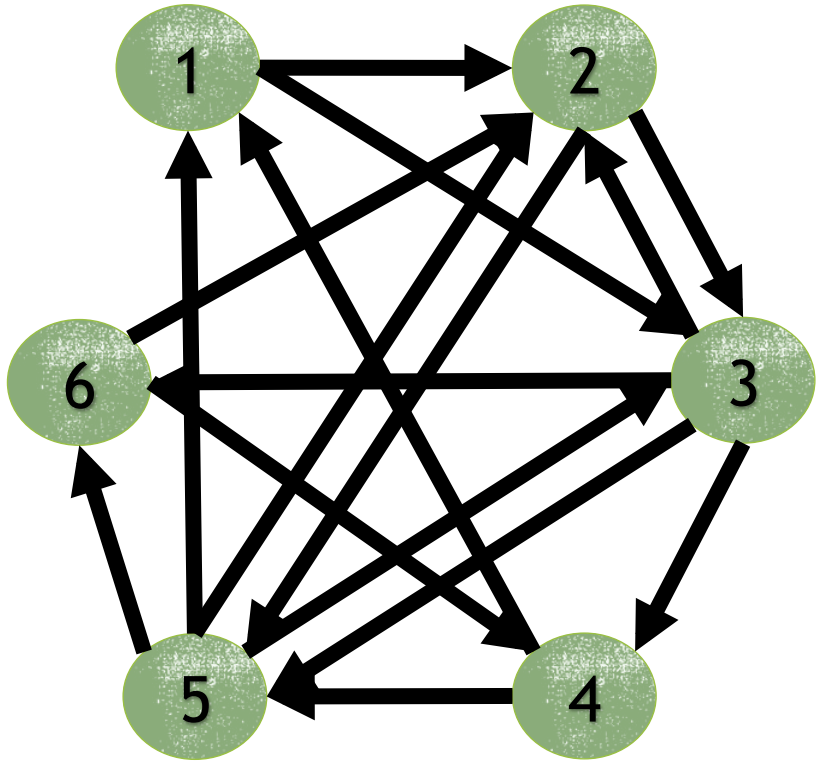


PSW. Load and Update

- Interval 3-4

Shard 1			Shard 2			Shard 3		
src	dst	value	src	dst	value	src	dst	value
1	2	0,273	1	3	0,364	2	5	0,55
3	2	0,22	2	3	0,273	3	5	0,9
4	1	1,54	3	4	0,8	3	6	1,2
5	1	0,55	5	3	0,2	4	5	0,3
5	2	0,66	6	4	1,9	5	6	1,1
6	2	0,88						

Sliding shard | Memory shard | Sliding shard



PSW. Load and Update

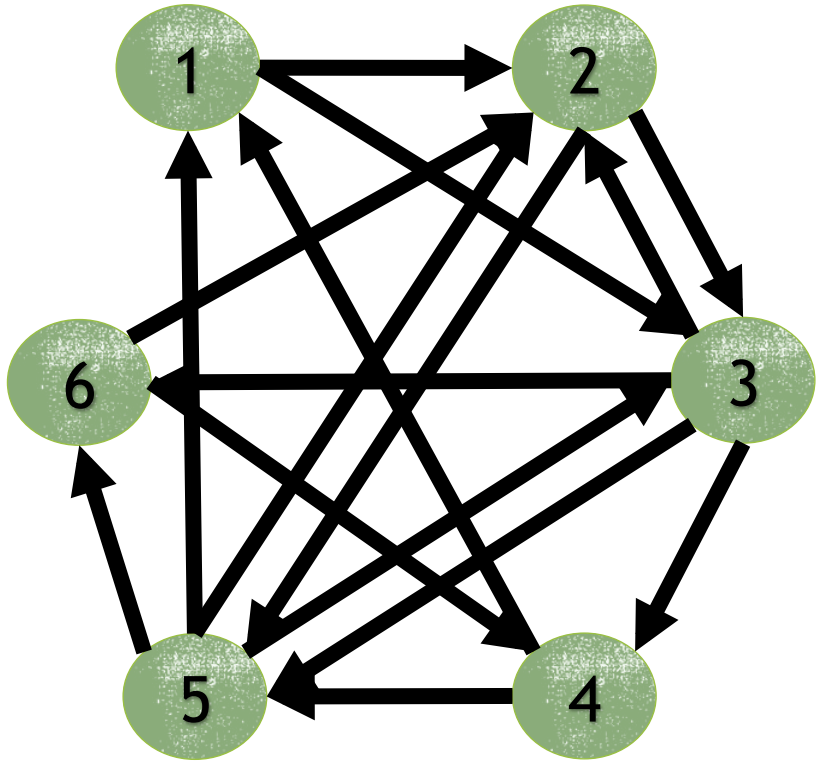
- Interval 5-6

Shard 1			Shard 2			Shard 3		
src	dst	value	src	dst	value	src	dst	value
1	2	0,273	1	3	0,364	2	5	0,55
3	2	0,32	2	3	0,73	3	5	0,8
4	1	1,4	3	4	0,66	3	6	1,22
5	1	0,55	5	3	0,32	4	5	0,53
5	2	0,66	6	4	1,29	5	6	1,1
6	2	0,88						

Sliding shard

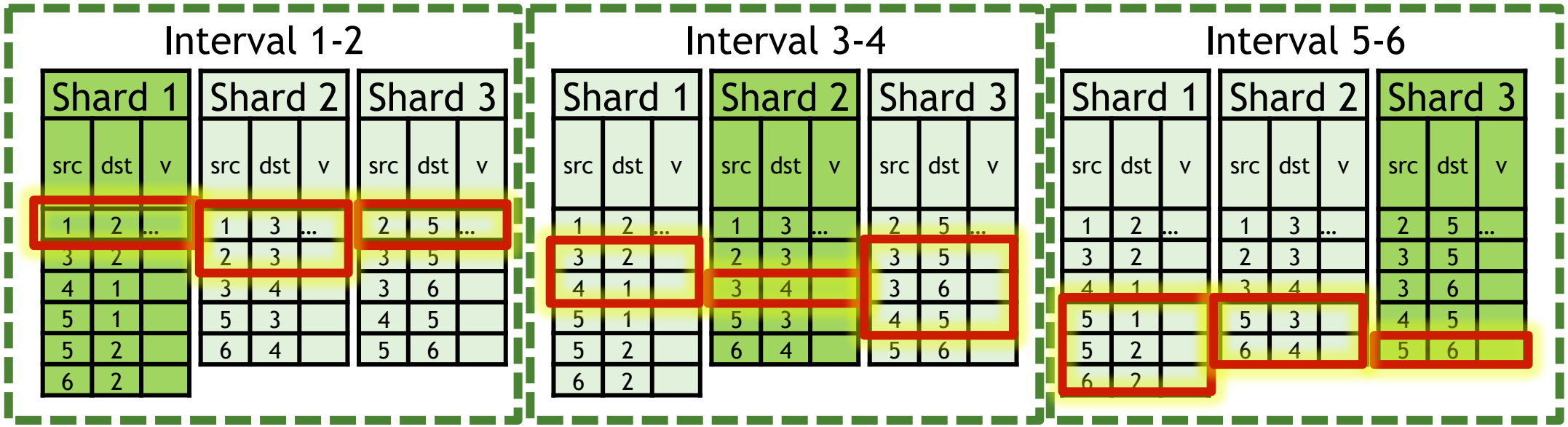
Sliding shard

Memory shard

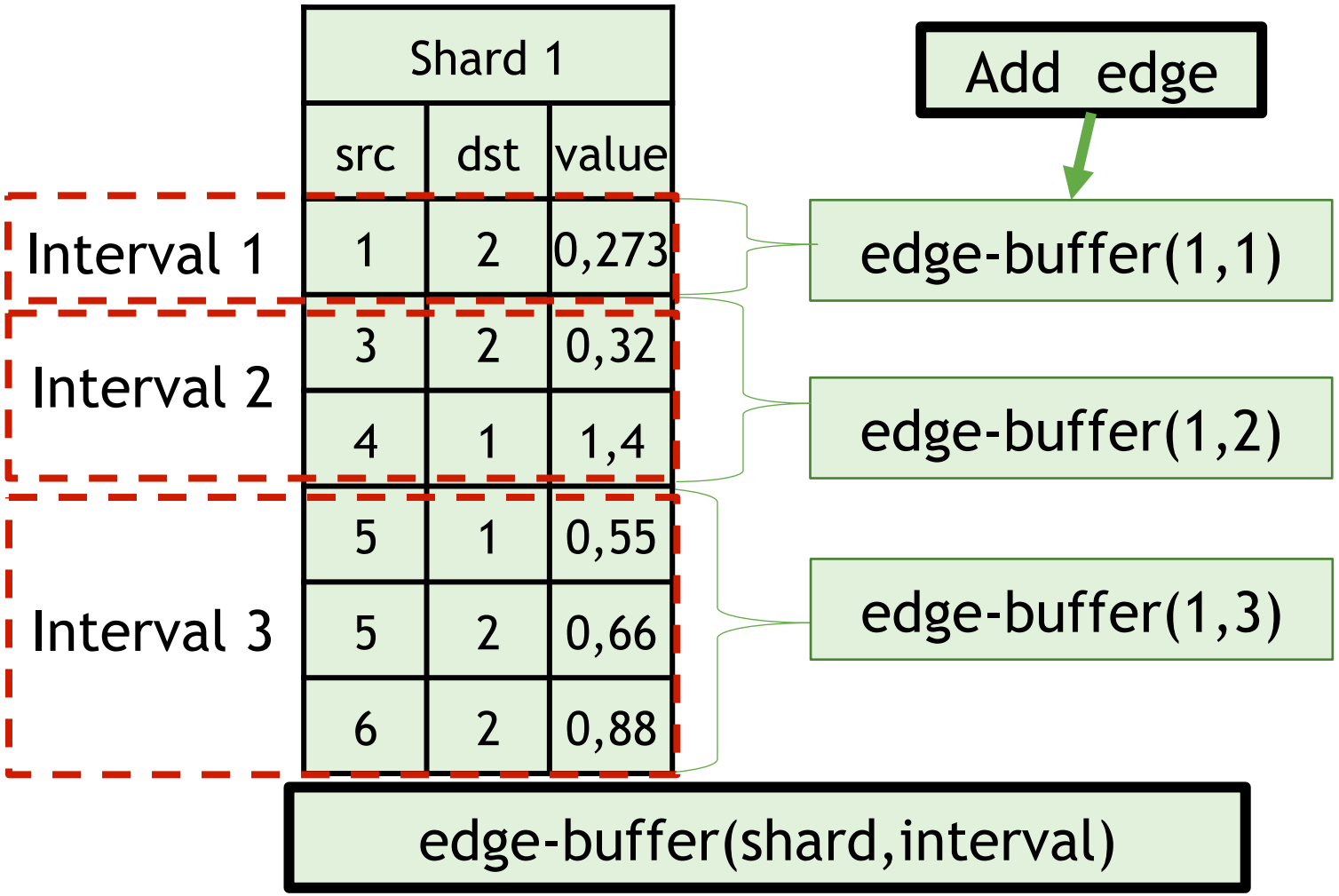


PSW. Parallel Updates

- External determinism = each execution of PSW produces exactly the same result
- Vertices that have edges with both end-points in the same interval are flagged as critical, and are updated in sequential order.
- Non-critical edges can be updated in parallel



PSW. Evolving Graphs



- After each iteration if edge-buffer exceeds limit => write buffered edges to disk
- Merge buffered edges with edges on the disk
- If merged shard doesn't fit in main memory => split it into 2 shards

PSW. Evolving Graphs

Remove edge

- Flag edge
- Ignore it
- Permanently delete when rewriting to disk

I/O Cost Analysis

- Cost of an algorithm is #block_transfers from disk to main memory
- B – size of the block transfer
- Total data size - |E| edge objects (∀ edge is stored once)
- P - #intervals

$2|E|/B \leq cost\downarrow B(E) \leq 4|E|/B + \Theta(P\uparrow 2)$

$|E|/B$ - cost of read/write of all edges

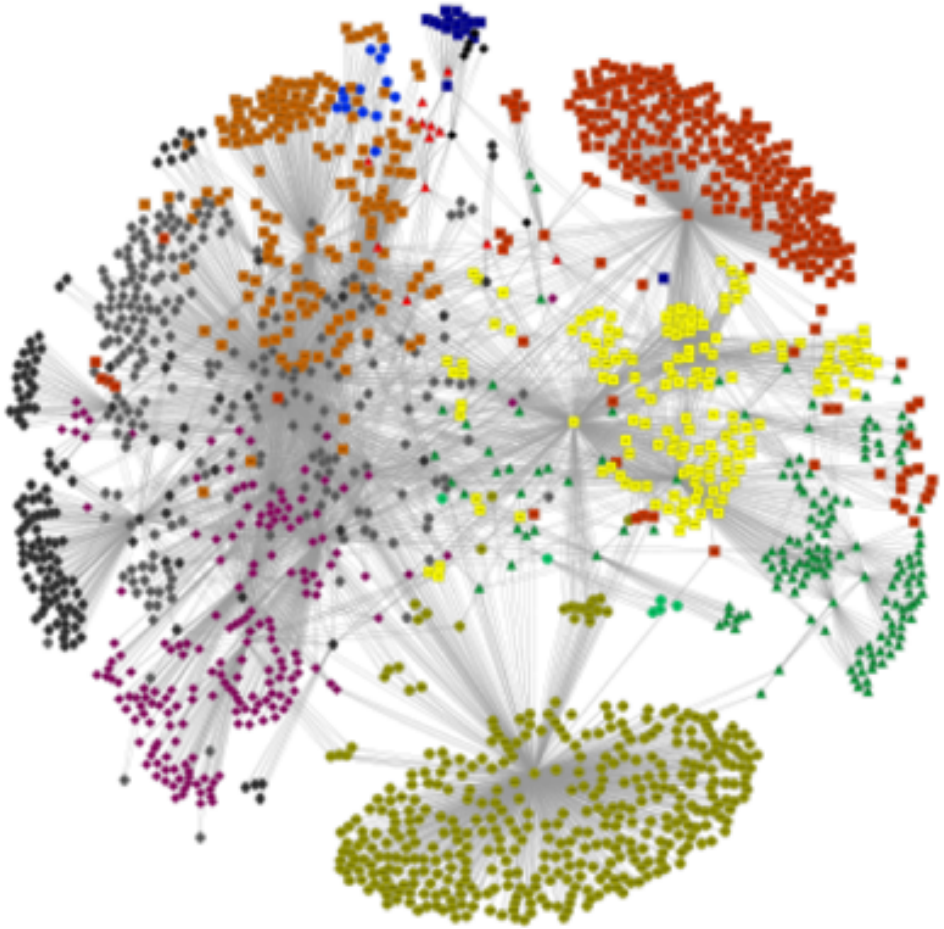
$P\uparrow 2$ - PSW needs P random seeks to load from P-1 sliding shards per interval

If update edges in both directions – 2 writes, else 1

If edge is non-critical – 2 reads, else 1

Applications

- PageRank
- Graph Mining
  - Connected components
  - Community detection
  - Triangle counting
- Collaborative filtering
- Probabilistic graphical model

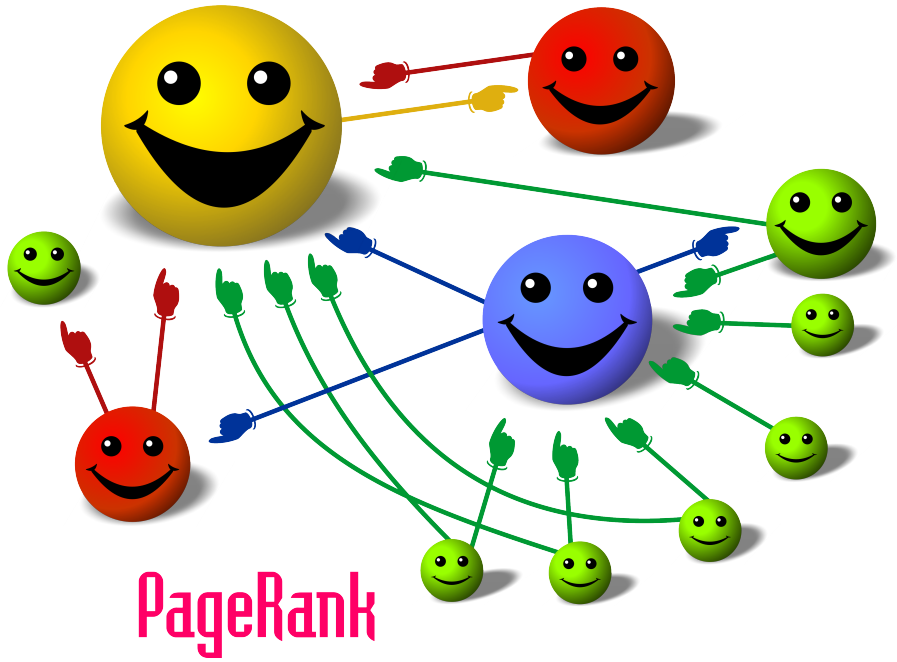


Applications. PageRank

```
def update(vertex):
    var sum←0
    for e in vertex.inEdges():
            sum += e.weight * neighborRank(e)
    vertex.setValue(0.15 + 0.85 * sum)
    broadcast(vertex)
```

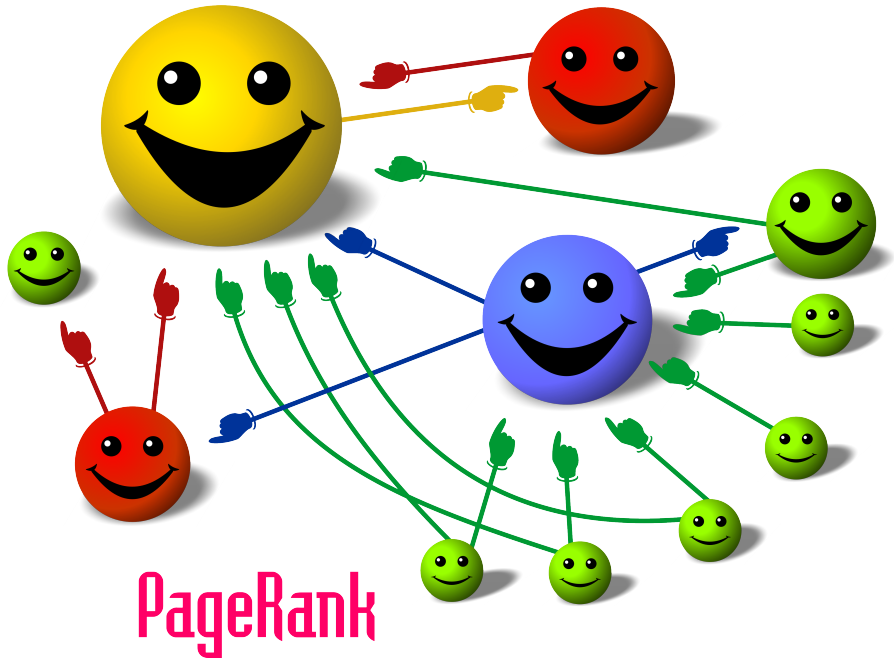
```
def neighborRank(edge):
        return edge.weight*edge.neighbor_rank
```

```
def broadcast(vertex):
        for e in vertex.outEdges():
                e.neighbor_rank = vertex.getValue()
```



Applications. PageRank’

```
float[] in_mem_vert
def update(vertex):
        var sum←0
        for e in vertex.inEdges():
                sum += e.weight * neighborRank(e)
        vertex.setValue(0.15 + 0.85 * sum)
def neighborRank(edge):
        return edge.weight*in_mem_vert[edge.vertex_id]
//def broadcast(vertex):
```

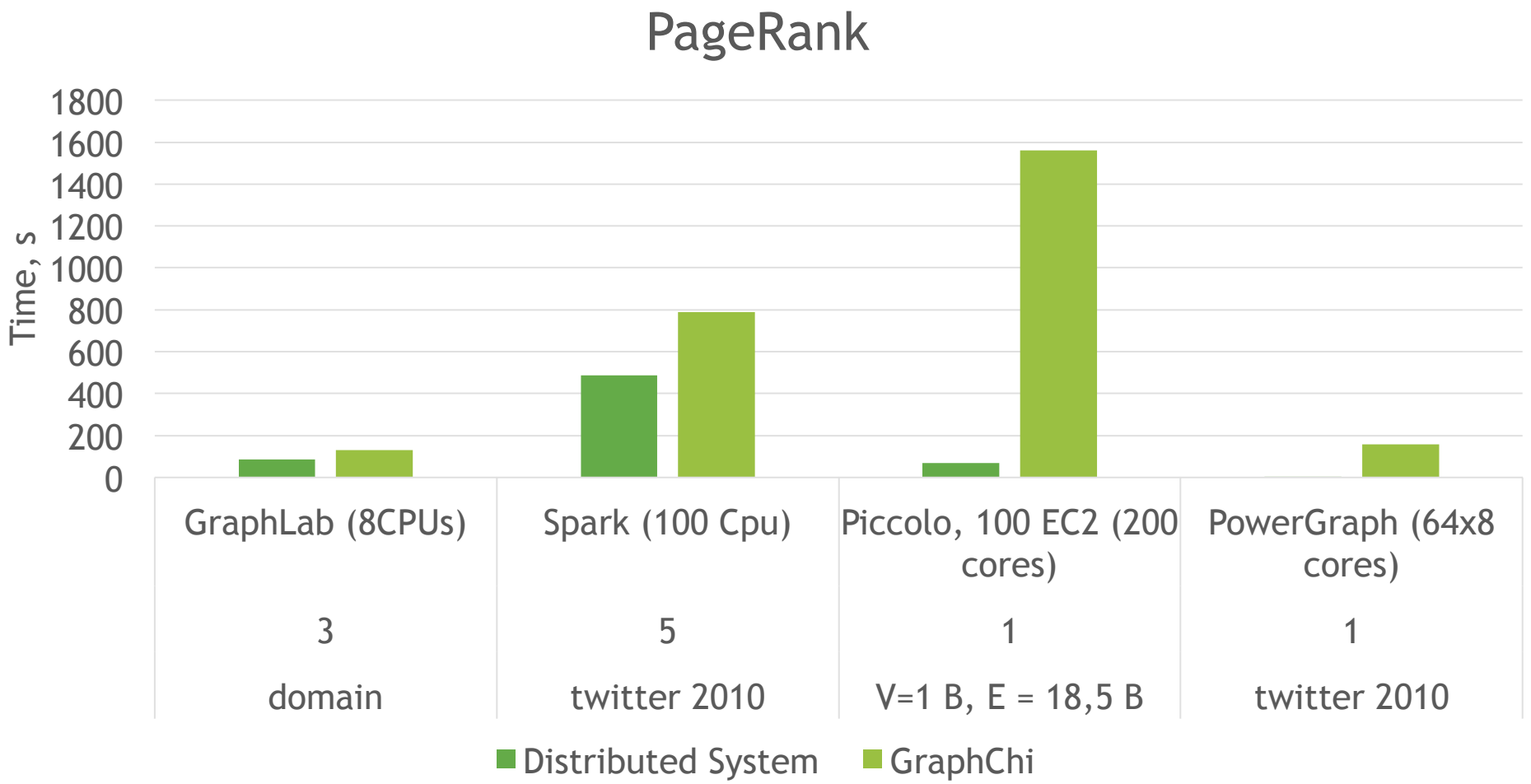


Experiments. Setup

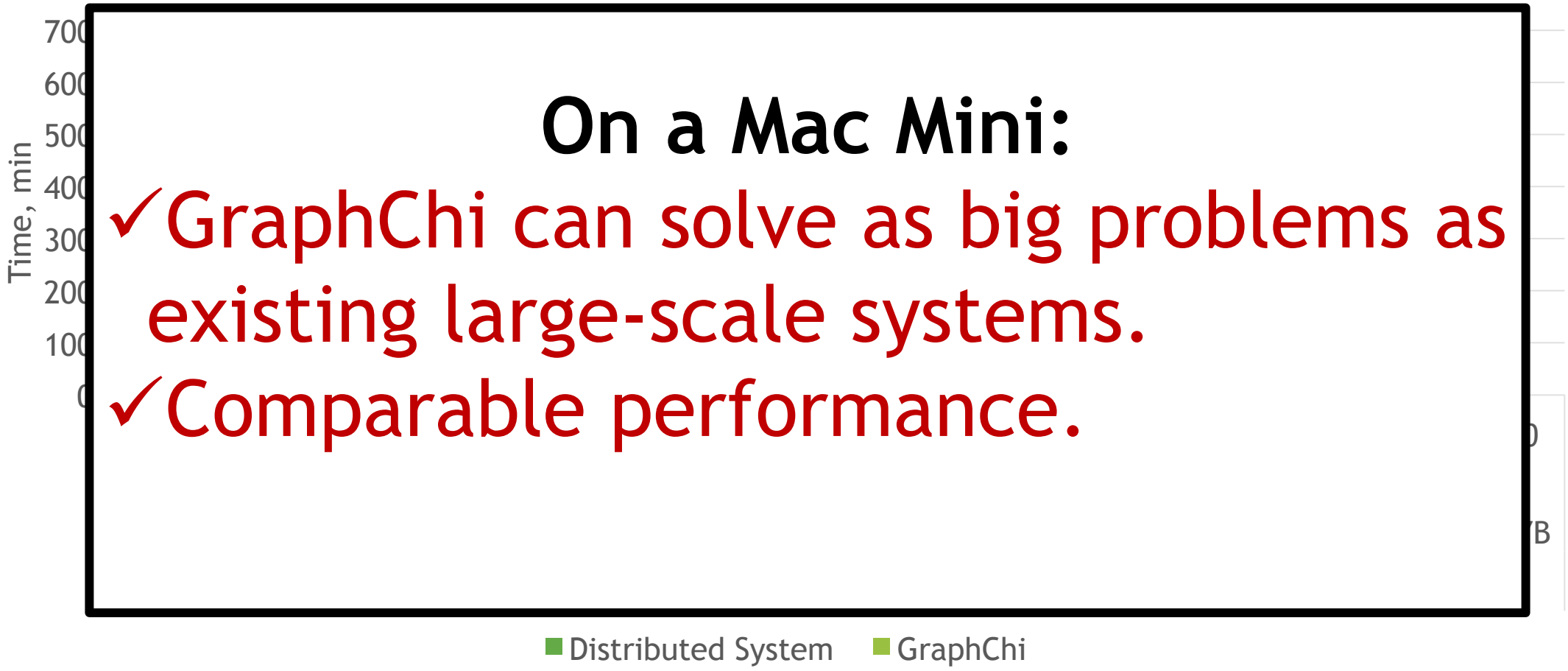
- Mac Mini (Apple Inc.)
  - 8 GB RAM
  - 256 GB SSD
  - 1TB hard drive
  - Intel Core i5, 2.5 GHz
  - C++

Graph	Vertices	Edges	P (shards)	Preprocessing
live-journal	4.8M	69M	3	0.5 min
netflix	0.5M	99M	20	1 min
twitter-2010	42M	1.5B	20	2 min
uk-2007-05	106M	3.7B	40	31 min
uk-union	133M	5.4B	50	33 min
yahoo-web	1.4B	6.6B	50	37 min

GraphChi vs. Distributed Systems



GraphChi vs. Distributed Systems



On a Mac Mini:

- ✓GraphChi can solve as big problems as existing large-scale systems.
- ✓Comparable performance.

Scalability & Performance

- ✓GraphChi is 2 times slower with HDs than with SSD
- ✓Performance can be improved by using multiple hard drives

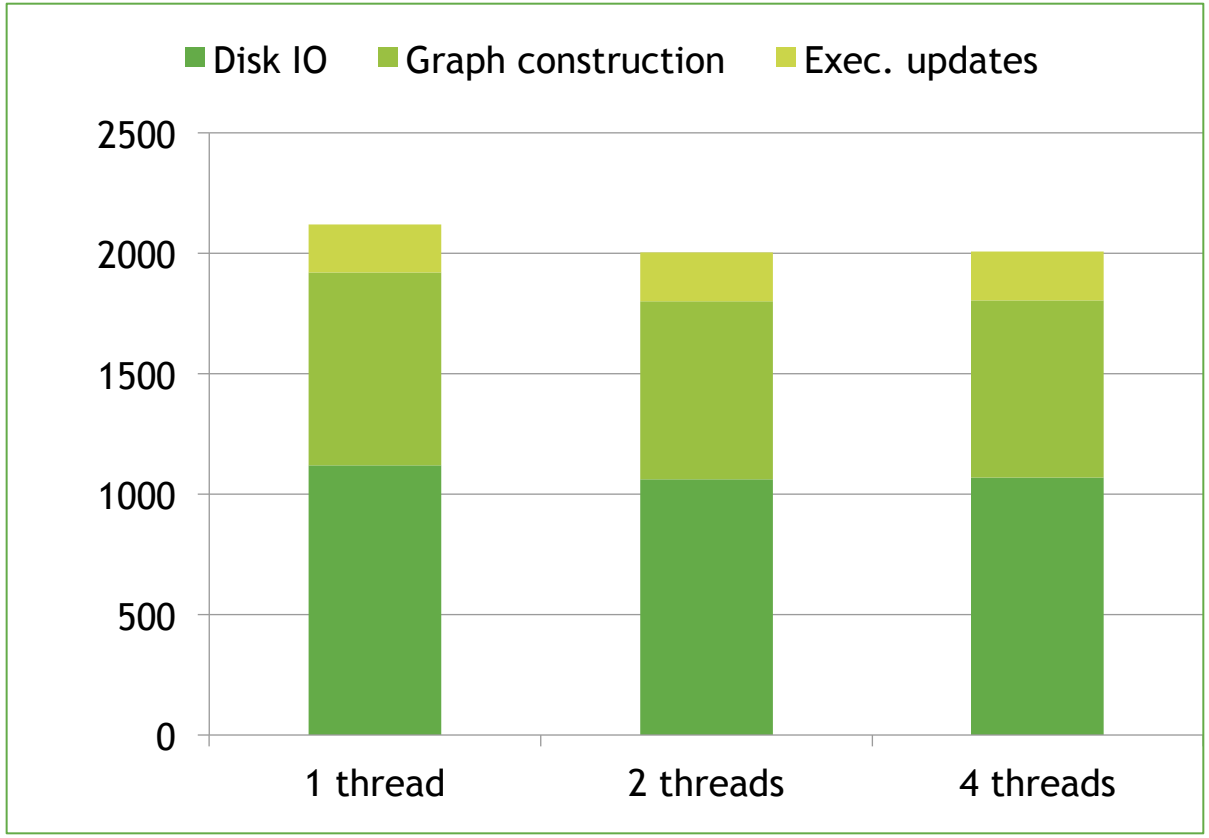
Scalability & Performance

$2|E|/B \leq cost_{\downarrow B}(E) \leq 4|E|/B + \Theta(P_{\uparrow 2})$

- ✓Number of blocks affect performance of both SSD and HD
- ✓P (#intervals) in the order of dozens causes little effect on performance

Bottlenecks

- Cost of constructing the sub-graph in memory is almost as large as the I/O cost on an SSD
  - Graph construction requires a lot of random access in RAM → memory bandwidth becomes a bottleneck.

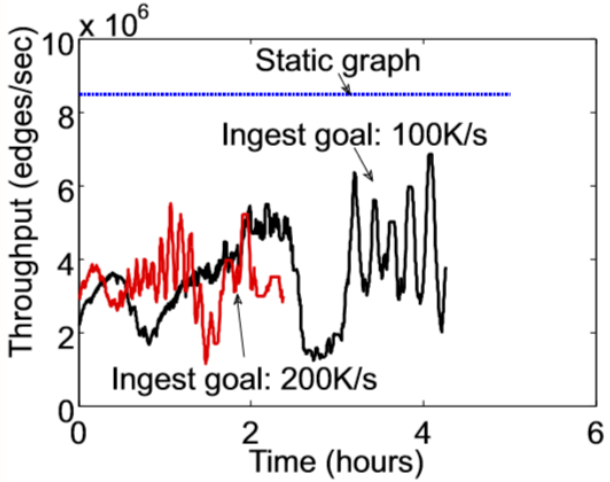


Connected Components on Mac Mini / SSD

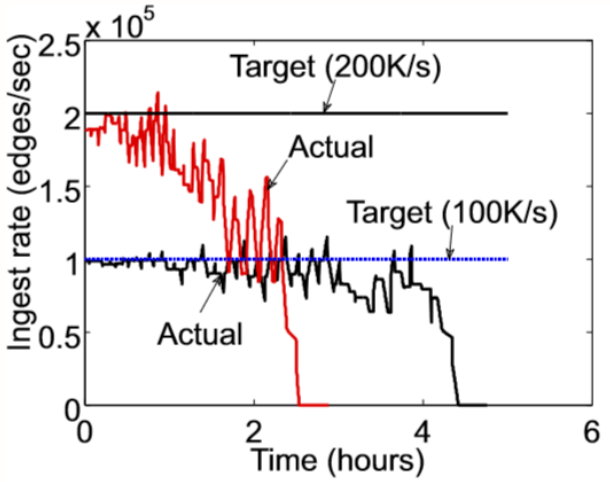
In-memory VS. Disk

Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

Evolving Graphs



(a) Evolving Graph: Throughput



(b) Ingest rate

- Insert edges from twitter-2000. With rates 100K edges and 200 K edges per second
- PageRank simultaneously
- Throughput in evolving case ~ 50% compared to normal execution

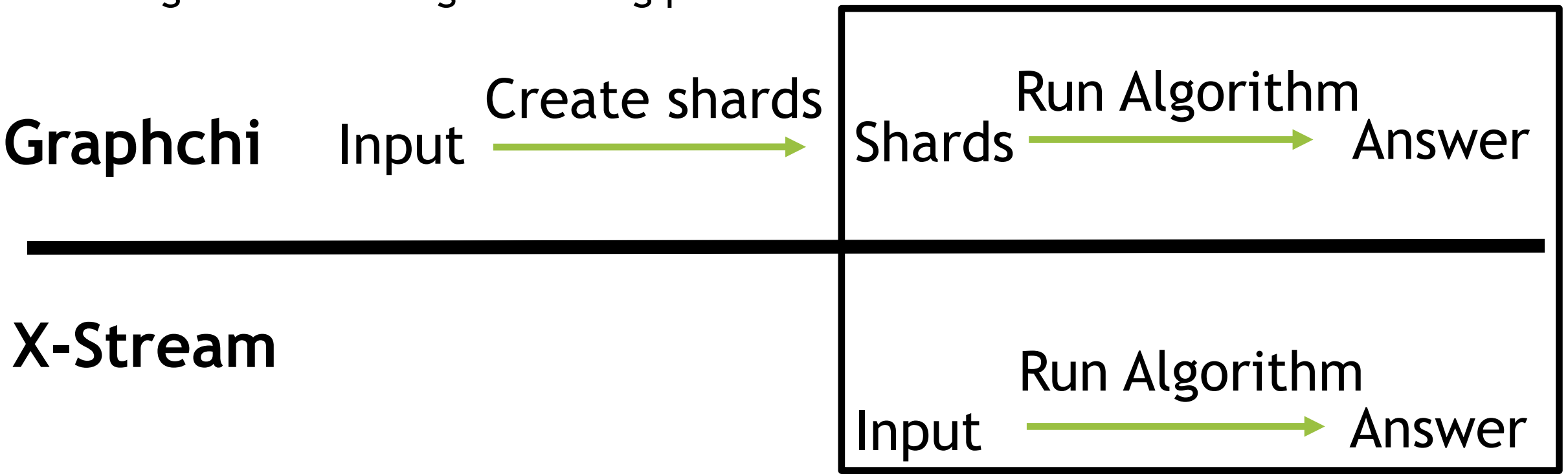
✓GraphChi can handle a very quickly growing graph on just a PC

GraphChi is not good for

- Very large vertex state
- Traversals
- High diameter graphs, such as planar graphs
  - Unless the computation itself has short-range interactions
- Very large number of iterations
- No support for implicit graph structure

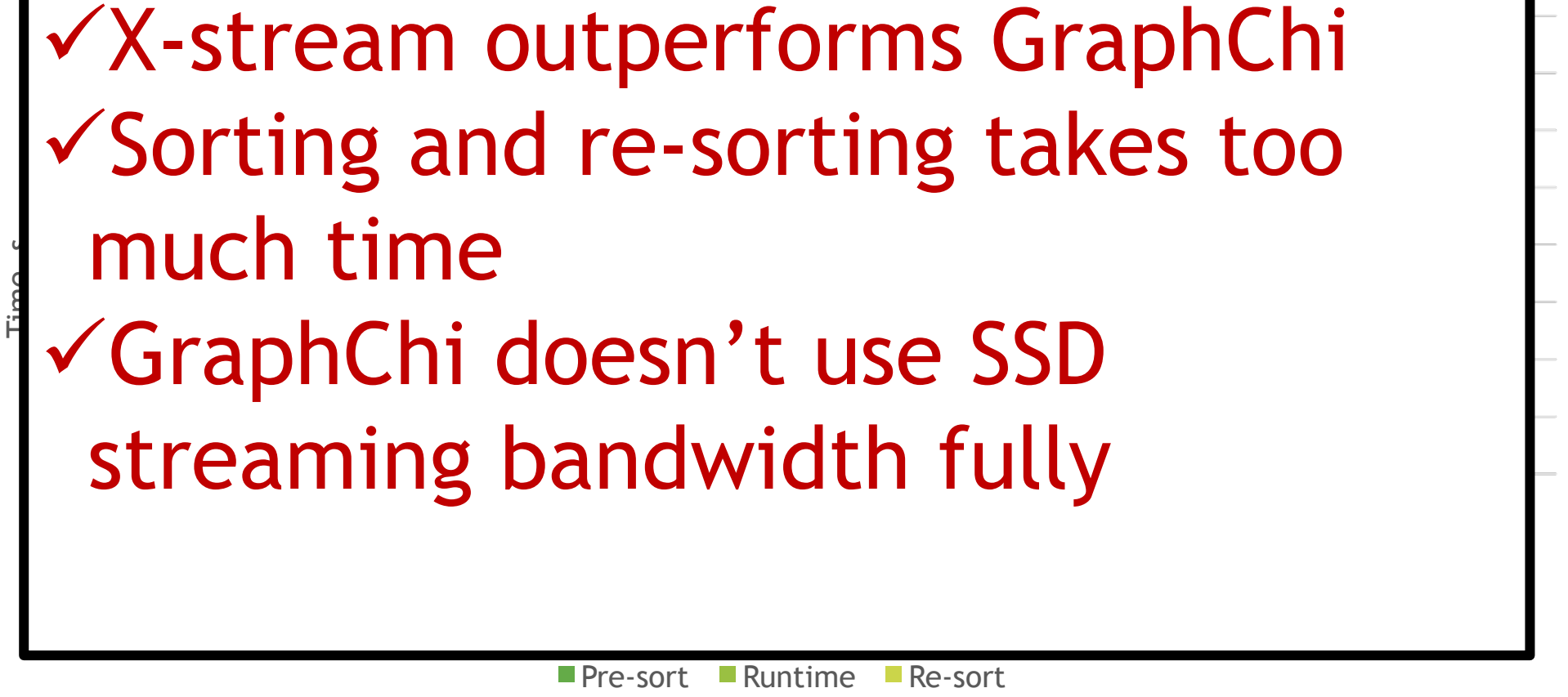
X-Stream VS. GraphChi

X-Stream. Single-machine system.
But edge-centric using streaming partitions



X-Stream VS. GraphChi

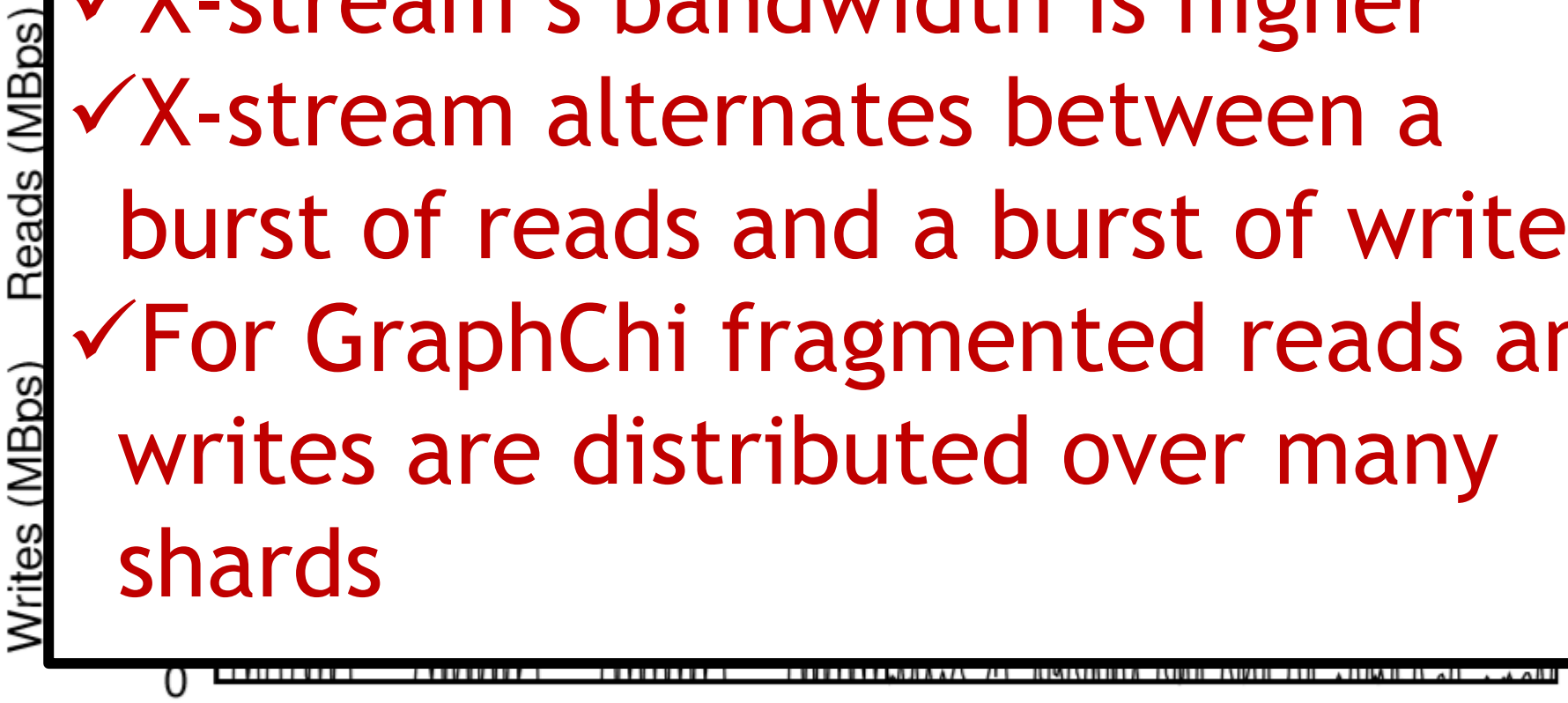
- ✓X-stream outperforms GraphChi
- ✓Sorting and re-sorting takes too much time
- ✓GraphChi doesn't use SSD streaming bandwidth fully



Sequential Access Bandwidth

GraphChi	X-stream
shards	partitions
All vertices and edges	Only vertices
must fit in memory	
More shards than partitions	
More random access for GraphChi	

Disk Bandwidth



- ✓X-stream's bandwidth is higher
- ✓X-stream alternates between a burst of reads and a burst of writes
- ✓For GraphChi fragmented reads and writes are distributed over many shards

GraphChi's drawbacks

- GraphChi has to sort
  - graph in pre-processing phase for shard partitioning
  - edges in the shard by destination after loading the shard into memory
- Incomplete usage of available streaming bandwidth from the SSD
- Constrained to the computational model (vertex-centric)

X-stream's drawbacks

- Trade-off: fewer random access to the edge list VS. streaming a large number of unrelated edges
- X-stream can perform suboptimaly with some graphs
- X-stream is bad for graphs of high diameter

Conclusion. What is GraphChi?

- Vertex-centric model
- Asynchronous computation
- Parallel Sliding Windows – new method, which can process a graph from disk, providing a small amount of random access
  - Performs well on both SSDs and HDs
- GraphChi uses a novel out-of-core data structure
  - Shards - partitions of the graph, which can fit in main memory
- GraphChi can efficiently solve large-scale graph problems on a consumer PC

Thanks!

Questions?