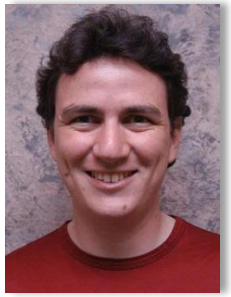
Thesis Defense

Large-Scale Graph Computation on Just a PC

Aapo Kyrölä

akyrola@cs.cmu.edu

Thesis Committee:



Carlos Guestrin
University of Washington & CMU

Guy Blelloch
CMU

Dave Andersen
CMU

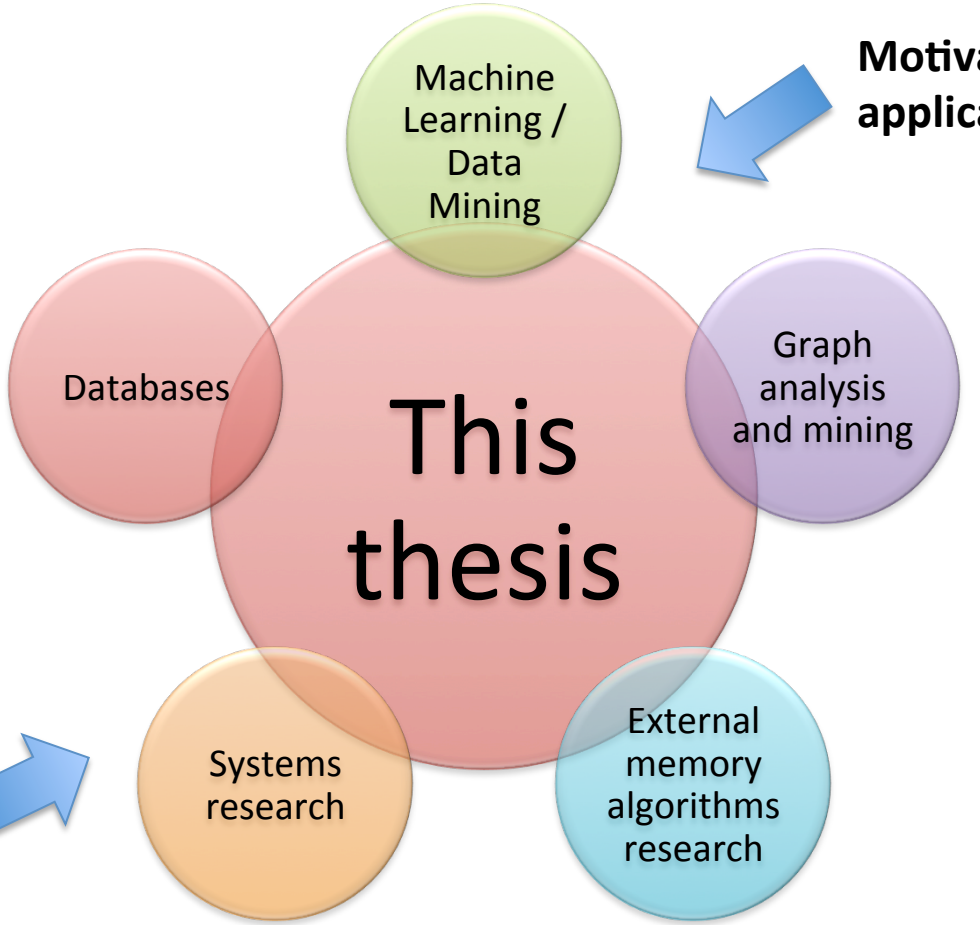
Alex Smola
CMU

Jure Leskovec
Stanford

Carnegie Mellon

Research Fields

GraphLab



Motivation and applications

Research contributions

Large-Scale **Graph** Computation on Just a PC

Why Graphs?

BigData with *Structure*: BigGraph

social graph

social graph

follow-graph

consumer-products graph

user-movie ratings graph

DNA interaction graph

WWW link graph

Communication networks (but "only 3 hops")

Large-Scale Graph Computation on a **Just a PC**

Why on a single machine?

Why use a cluster?

Two reasons:

1. One computer cannot handle my graph problem in a *reasonable* time.

2. I need to solve the problem very fast.

Why use a cluster?

Two reasons:

1. One computer cannot handle my graph problem in a *reasonable* time.

   Our work expands the space of feasible problems on one machine (PC):

   - Our experiments use the same graphs, or bigger, than previous papers on distributed graph computation. (+ we can do Twitter graph on a laptop)

2. I need to solve the problem very fast.

   Our work raises the bar on required performance for a "complicated" system.

Benefits of single machine systems

Assuming it can handle your big problems…

1. Programmer productivity
   - Global state, debuggers…
2. Inexpensive to install, administer, less power.
3. Scalability
   - Use cluster of single-machine systems to solve many tasks in parallel.

< 32K bits/sec

Large-Scale Graph Computation on Just a PC

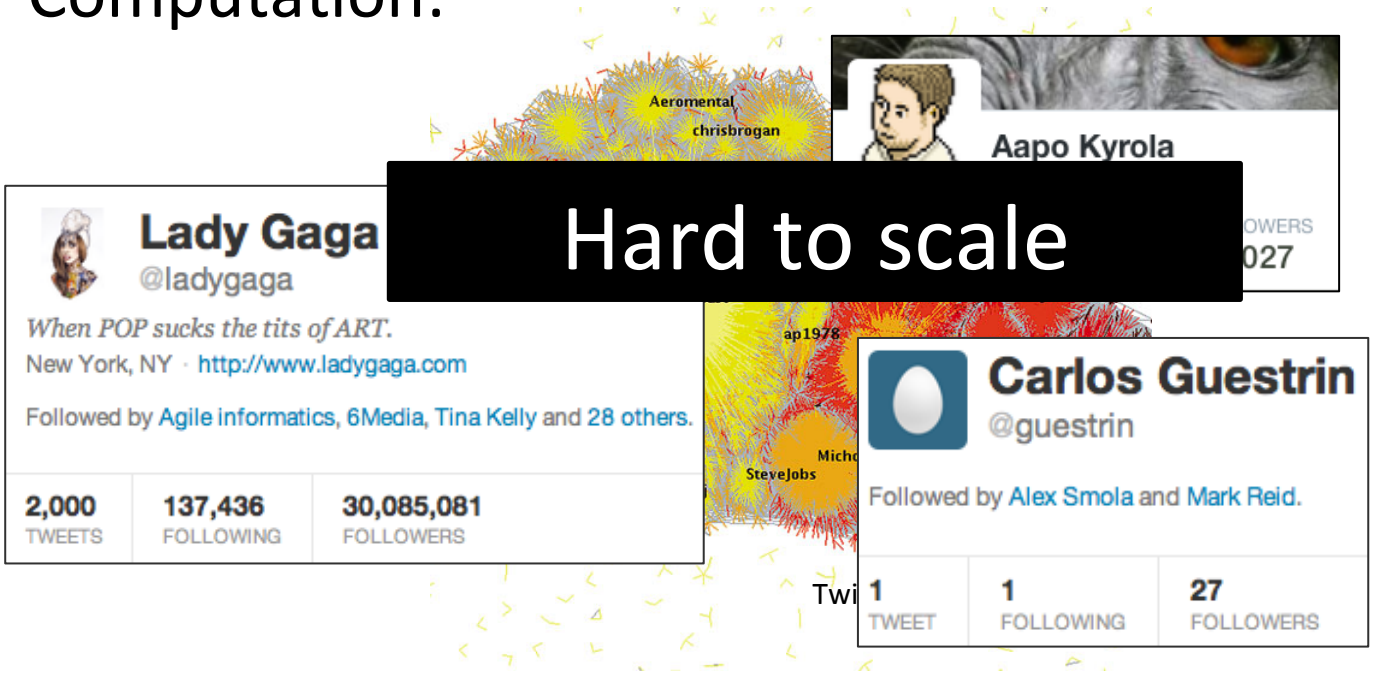
Computing on Big Graphs

Big Graphs != Big Data

Data size: facebook 140 billion connections ≈ 1 TB

Not a problem!

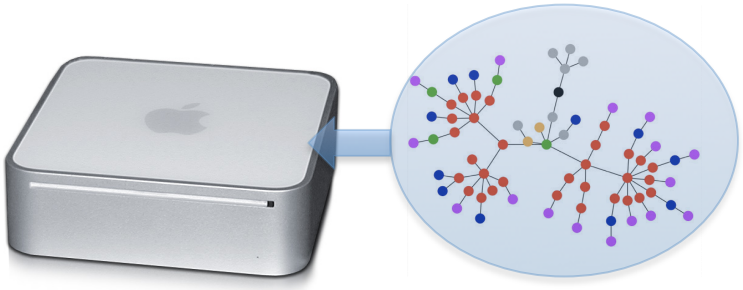
Computation:



Research Goal

Compute on graphs with billions of edges, in *a reasonable time,* on a single PC.

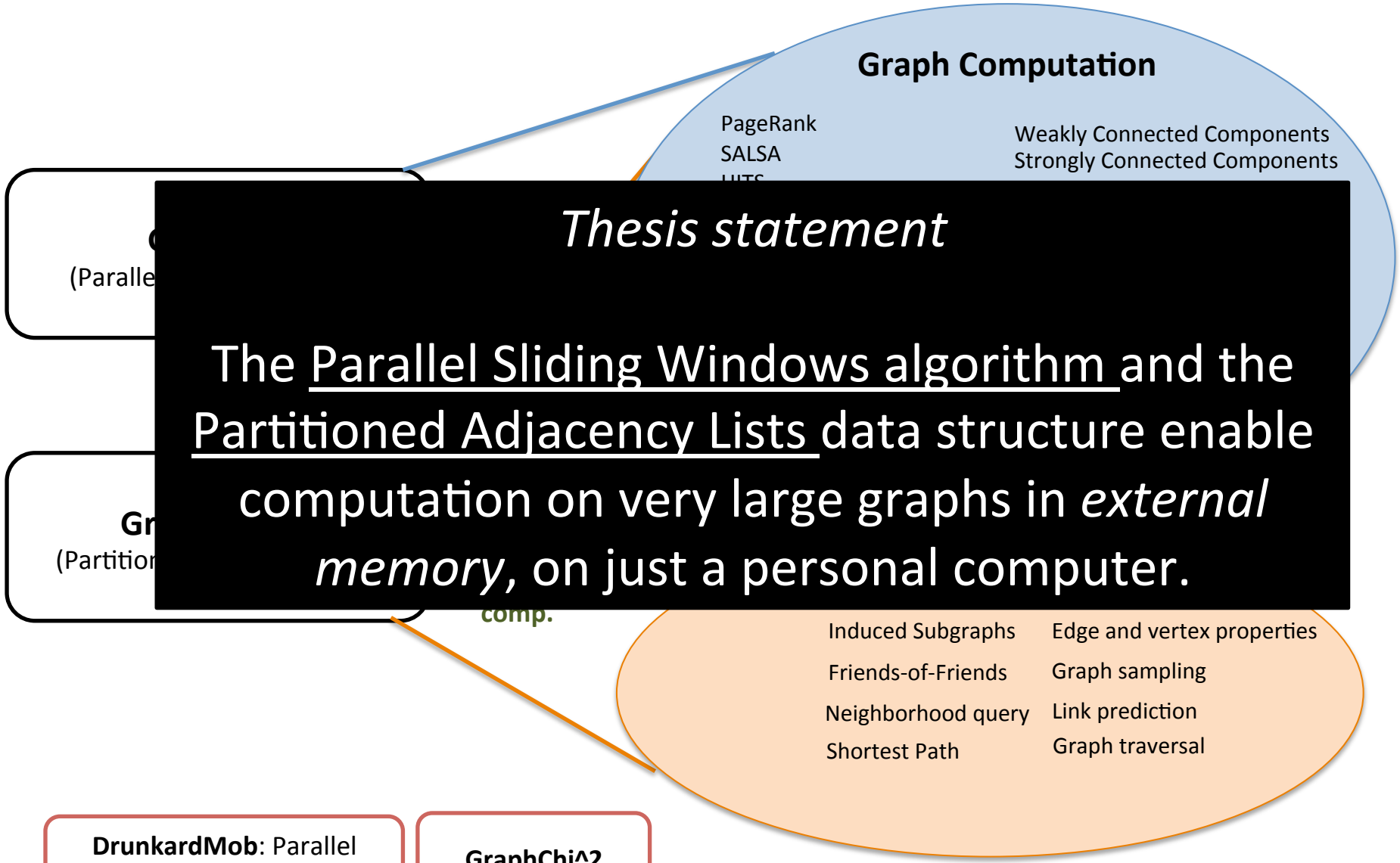
- *Reasonable* = close to numbers previously reported for distributed systems in the literature.



Experiment PC: Mac Mini (2012)

Terminology

- (Analytical) **Graph Computation:**
  - Whole graph is processed, typically for several iterations → *vertex-centric computation.*
  - Examples: Belief Propagation, Pagerank, Community detection, Triangle Counting, Matrix Factorization, Machine Learning…
- **Graph Queries (database)**
  - Selective graph queries (compare to SQL queries)
  - Traversals: shortest-path, friends-of-friends,…



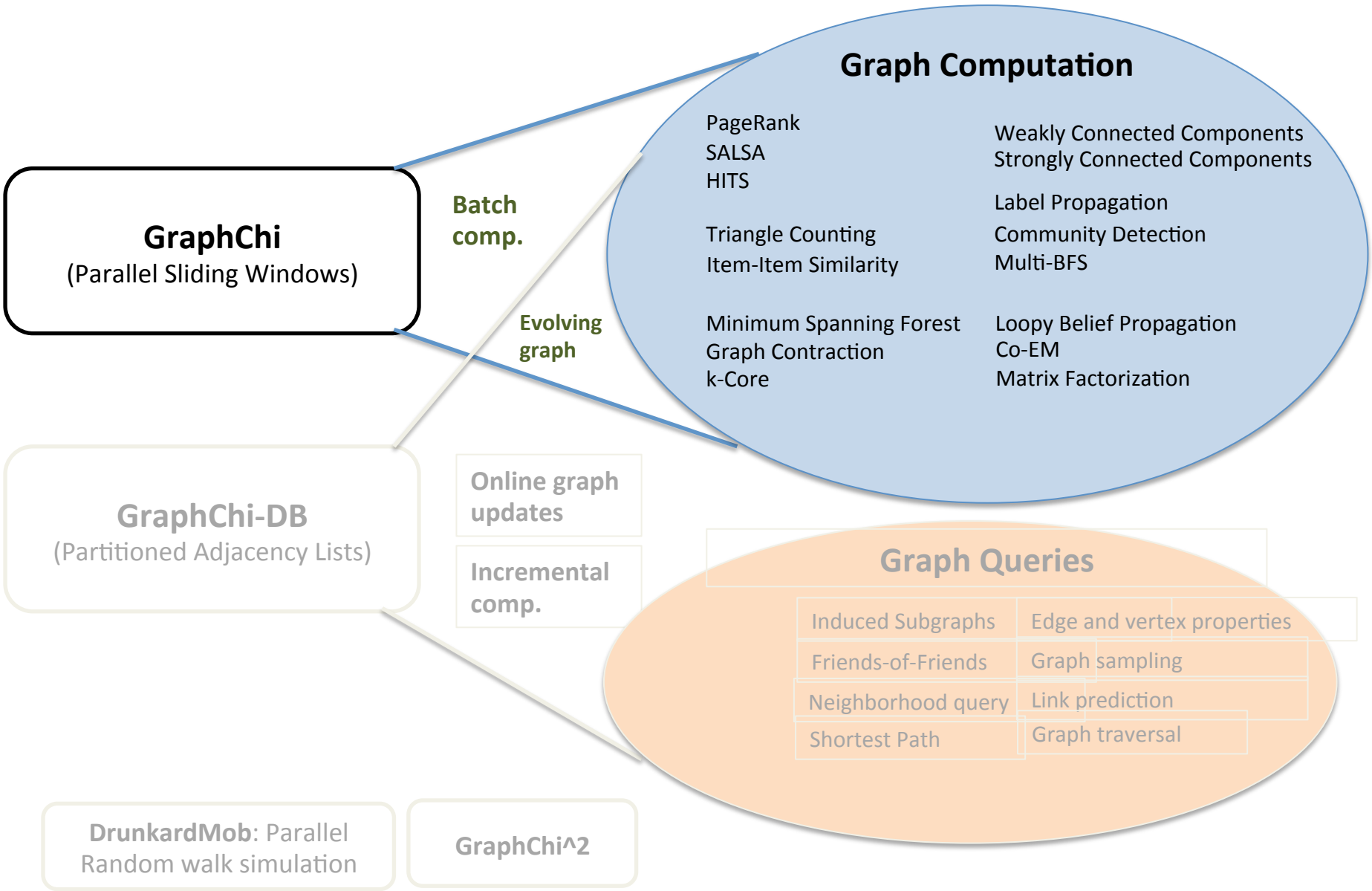
Thesis statement

The Parallel Sliding Windows algorithm and the Partitioned Adjacency Lists data structure enable computation on very large graphs in *external memory*, on just a personal computer.

DrunkardMob: Parallel Random walk simulation

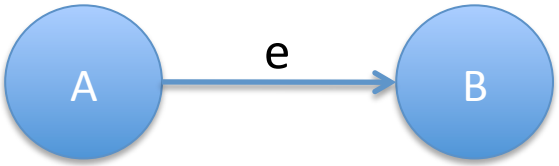
GraphChi^2

DISK-BASED GRAPH COMPUTATION

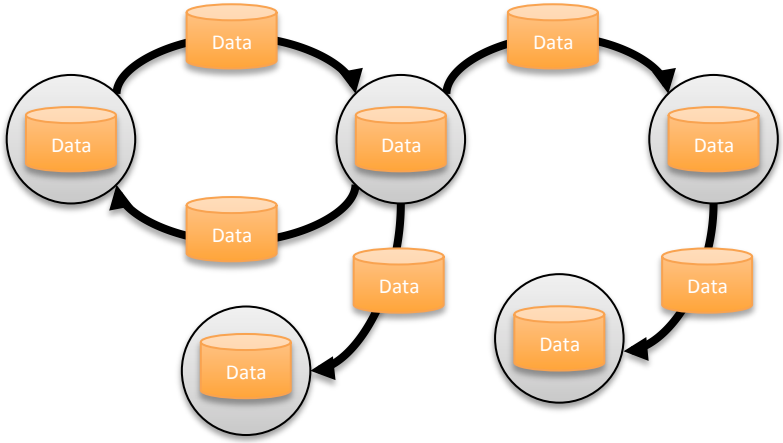

GraphChi
(Parallel Sliding Windows)
Batch comp.
Evolving graph
Graph Computation
PageRank
SALSA
HITS
Triangle Counting
Item-Item Similarity
Minimum Spanning Forest
Graph Contraction
k-Core
Weakly Connected Components
Strongly Connected Components
Label Propagation
Community Detection
Multi-BFS
Loopy Belief Propagation
Co-EM
Matrix Factorization
GraphChi-DB
(Partitioned Adjacency Lists)
Online graph updates
Incremental comp.
Graph Queries
Induced Subgraphs
Edge and vertex properties
Friends-of-Friends
Graph sampling
Neighborhood query
Link prediction
Shortest Path
Graph traversal
DrunkardMob: Parallel Random walk simulation
GraphChi^2

Computational Model

- Graph G = (V, E)
  - **directed edges**: e = (source, destination)
  - each edge and vertex **associated with a value** (user-defined type)
  - vertex and edge **values can be modified**
    - (structure modification also supported)

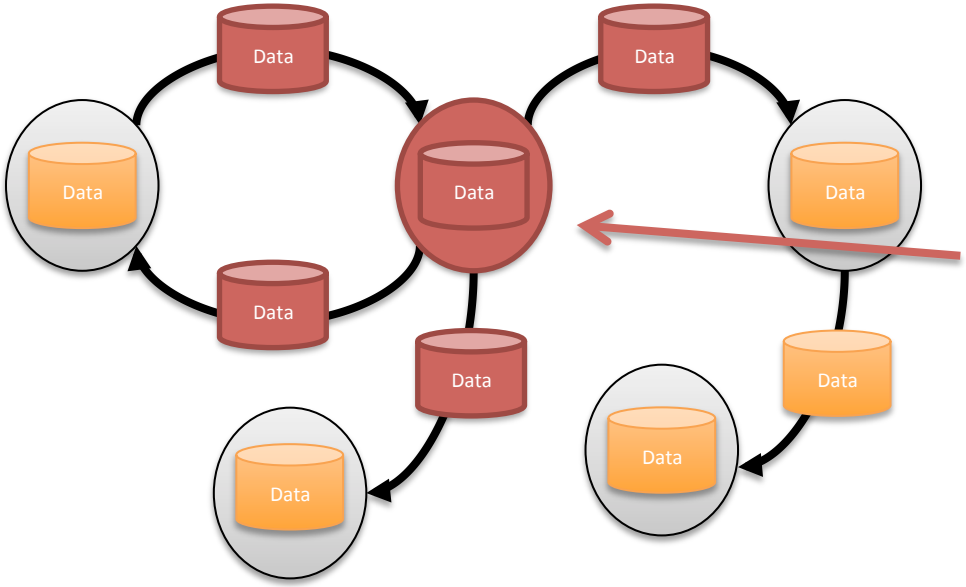


Terms: **e** is an **out-edge** of A, and **in-edge** of B.



Vertex-centric Programming

```
function Pagerank(vertex)
    insum = sum(edge.value for edge in vertex.inedges)
    vertex.value = 0.85 + 0.15 * insum
    foreach edge in vertex.outedges:
        edge.value = vertex.value / vertex.num_outedges
```



MyFunc(vertex)
{ // modify neighborhood }

Computational Setting

Constraints:

A. Not enough memory to store the whole graph in memory, nor all the vertex values.

B. Enough memory to store *one* vertex and its edges w/ associated values.

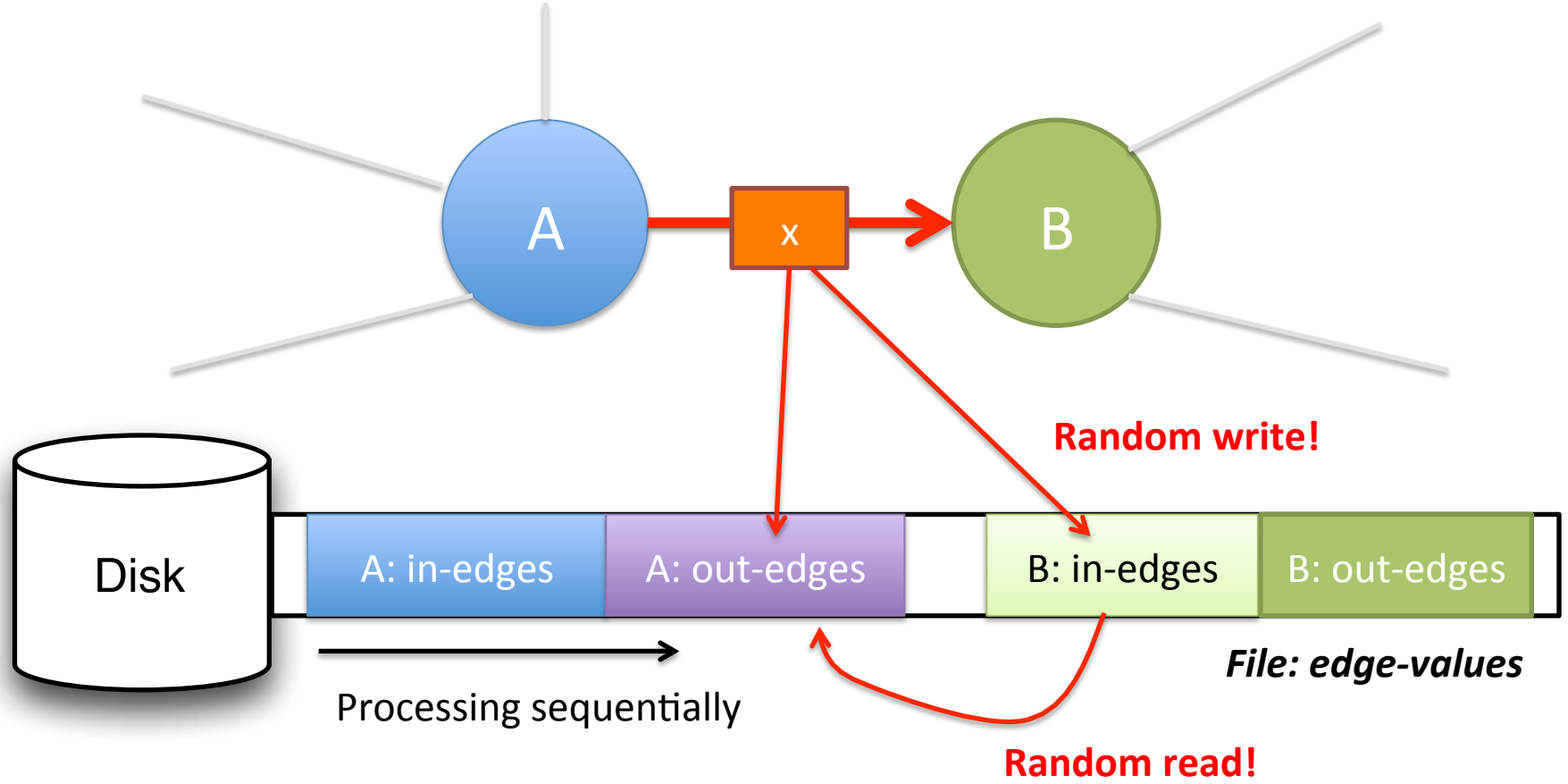
The Main Challenge of Disk-based Graph Computation:

Random Access

~ 100K reads / sec (commodity)
~ 1M reads / sec (high-end arrays)

<< **5-10 M** random edges / sec to achieve "reasonable performance"

Random Access Problem



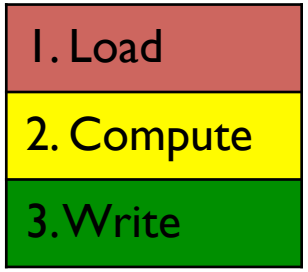
Moral: You can either access in- or out-edges sequentially, but not both!

Our Solution

Parallel Sliding Windows (PSW)

Parallel Sliding Windows: Phases

- PSW processes the graph one **sub-graph** a time:

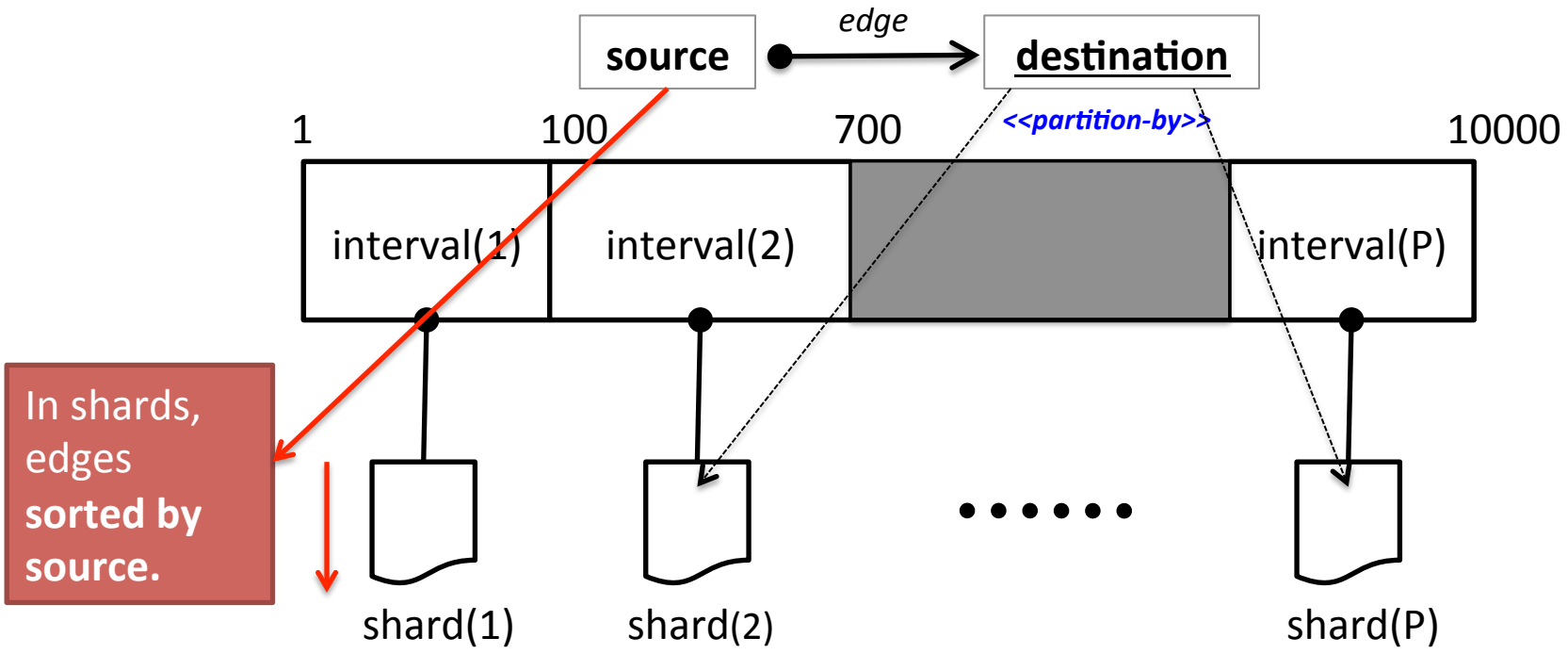


- In one **iteration**, the whole graph is processed.
  - And typically, next iteration is started.

PSW: Shards and Intervals

1. Load
2. Compute
3. Write

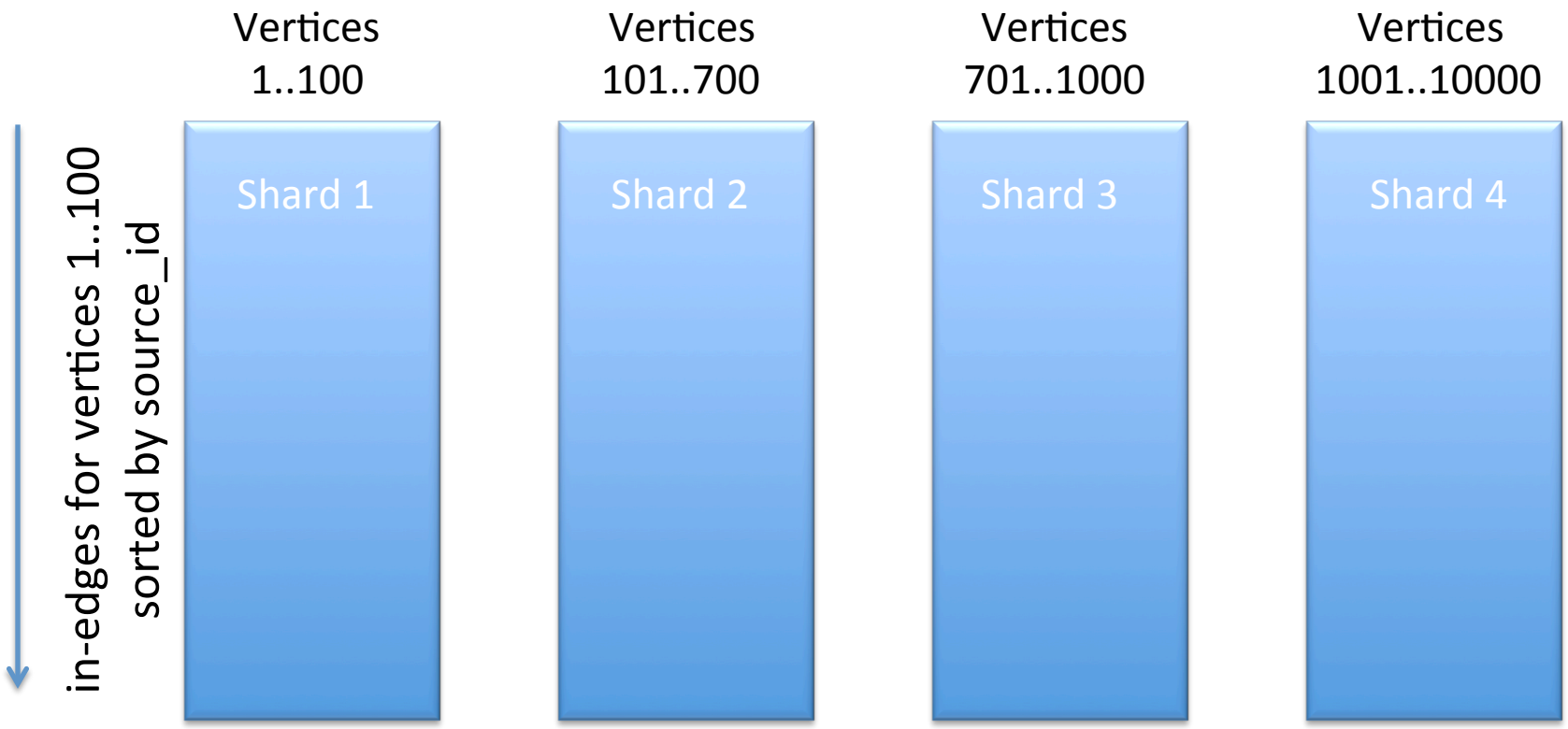
- Vertices are numbered from 1 to n
  - **P** intervals
  - **sub-graph =** interval of vertices



1. Load
2. Compute
3. Write

Example: Layout

Shard: in-edges for **interval** of vertices; sorted by source-id

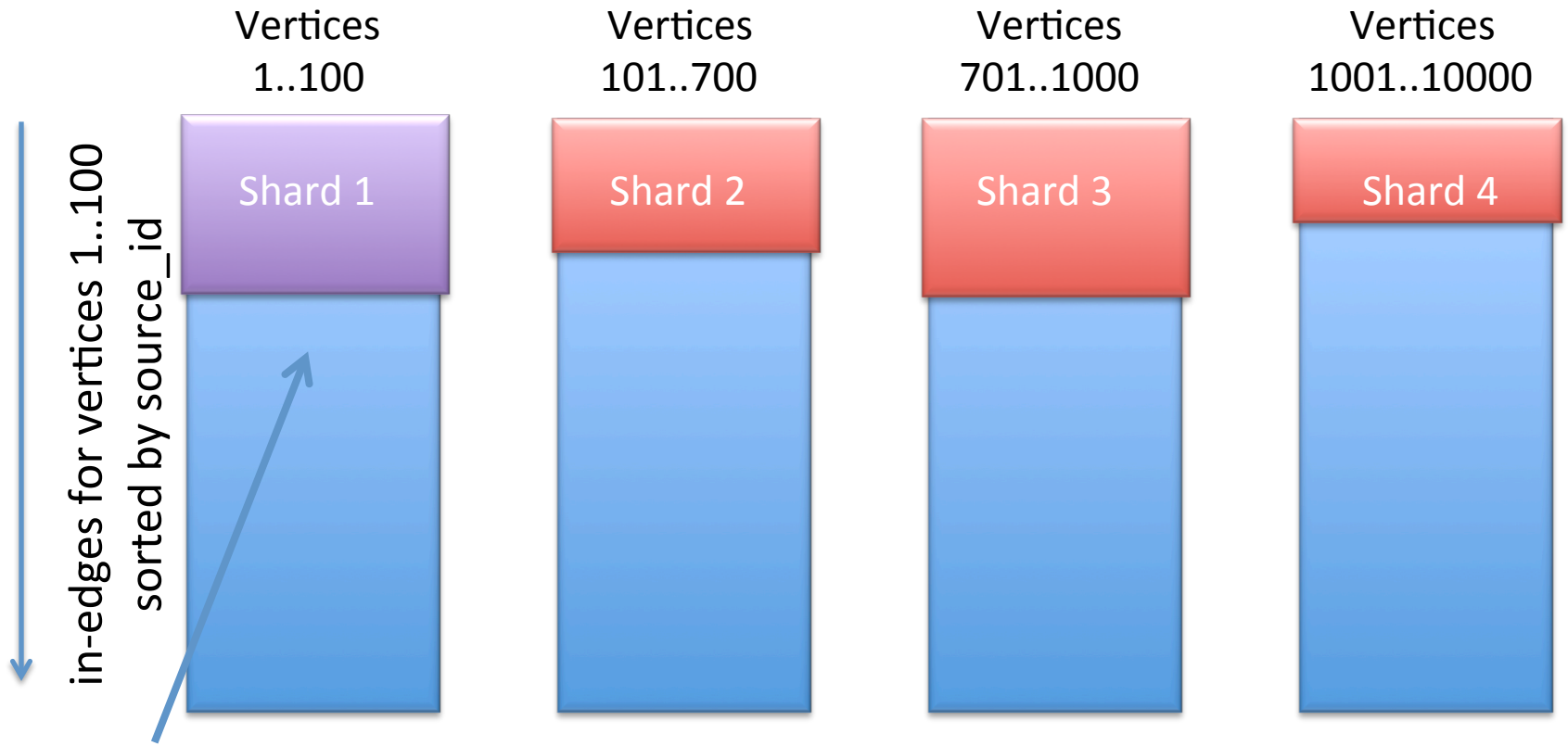


Shards small enough to fit in memory; balance size of shards

PSW: Loading Sub-graph

1. Load
2. Compute
3. Write

Load subgraph for vertices 1..100

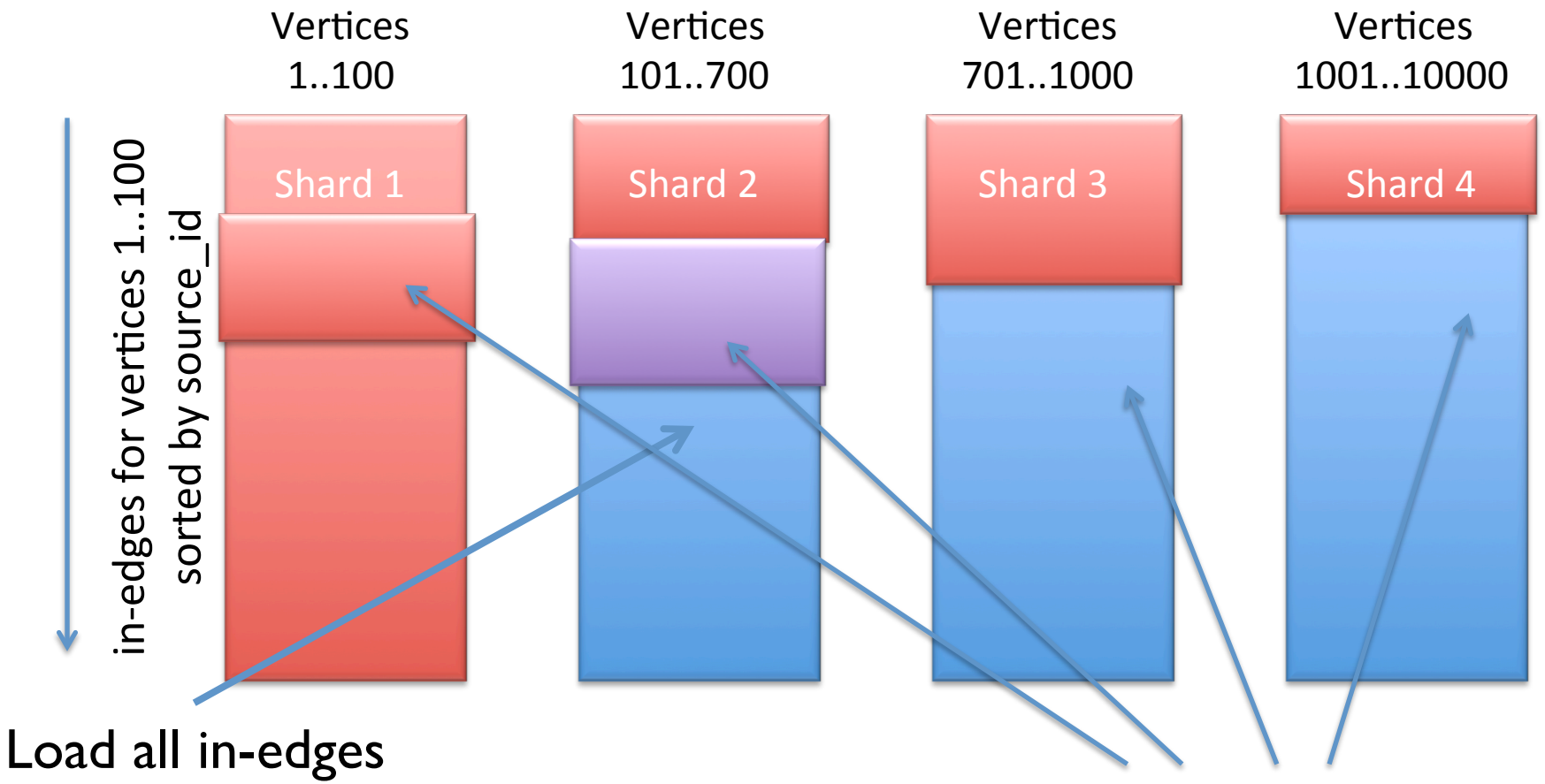


Load all in-edges in memory

What about out-edges?
Arranged in sequence in other shards

PSW: Loading Sub-graph

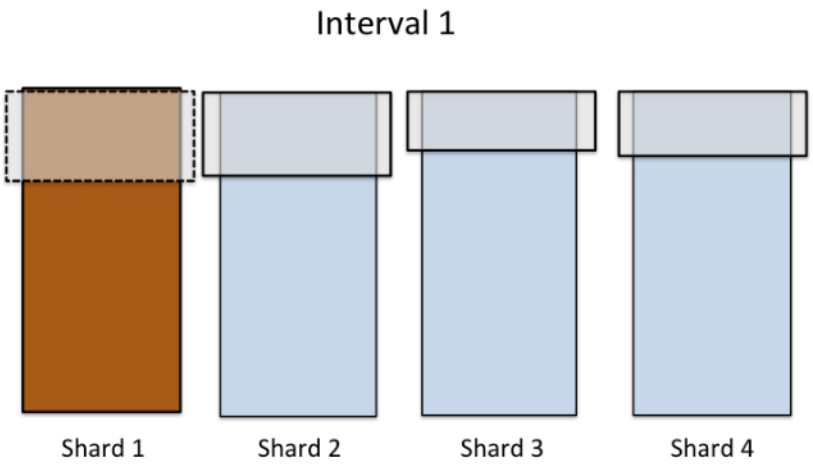
Load subgraph for vertices 101..700



Parallel Sliding Windows

Only P large reads and writes for each interval.

= P^2 random accesses on one full pass.



Works well on both SSD and magnetic hard disks!

Joint work:
Julian Shun

How PSW computes

"GAUSS-SEIDEL" / ASYNCHRONOUS

Chapter 6 + Appendix

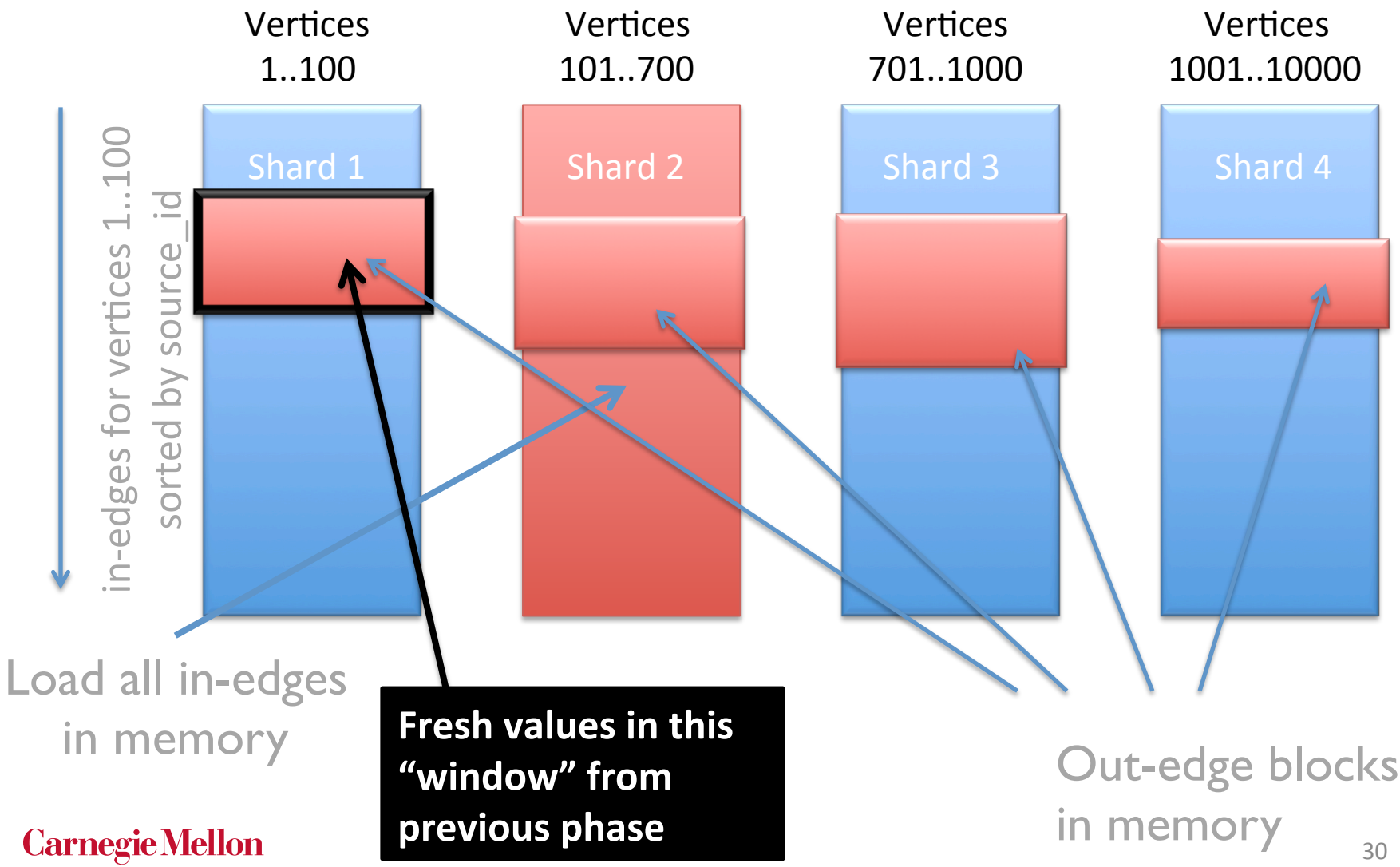
Synchronous vs. Gauss-Seidel

- Bulk-**Synchronous** Parallel (Jacobi iterations)
  - Updates see neighbors' values from *previous iteration*. [Most systems are synchronous]
- Asynchronous (**Gauss-Seidel** iterations)
  - Updates see most recent values.
  - GraphLab is asynchronous.

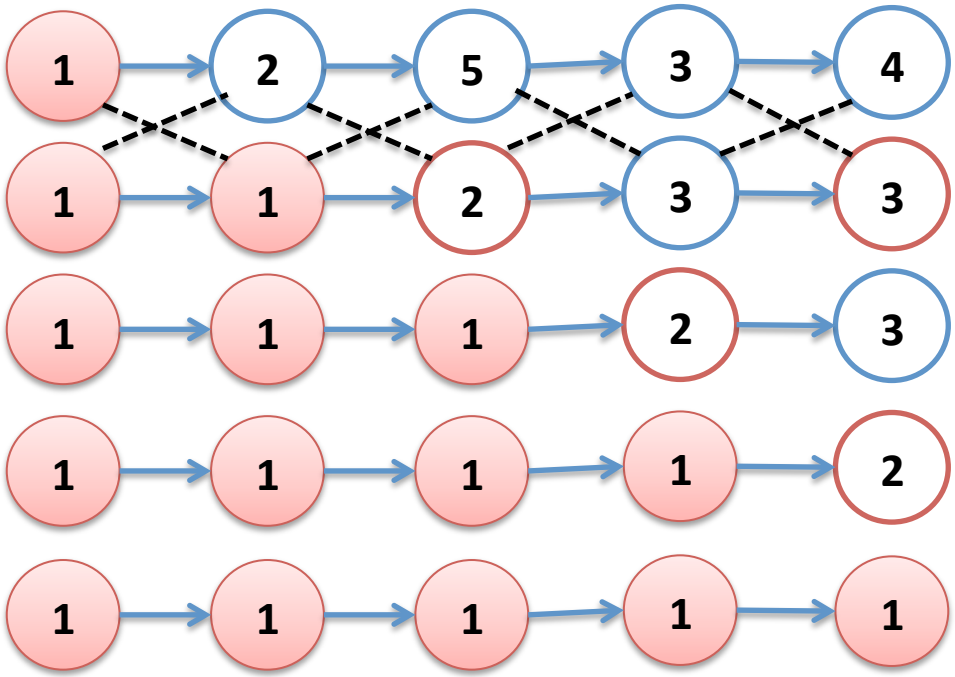
$$V_i^t \Leftarrow F(V_0^t, V_1^t, \ldots, V_{i-1}^t, V_i^{t-1}, V_{i+1}^{t-1}, \ldots)$$

PSW runs Gauss-Seidel

Load subgraph for vertices 101..700



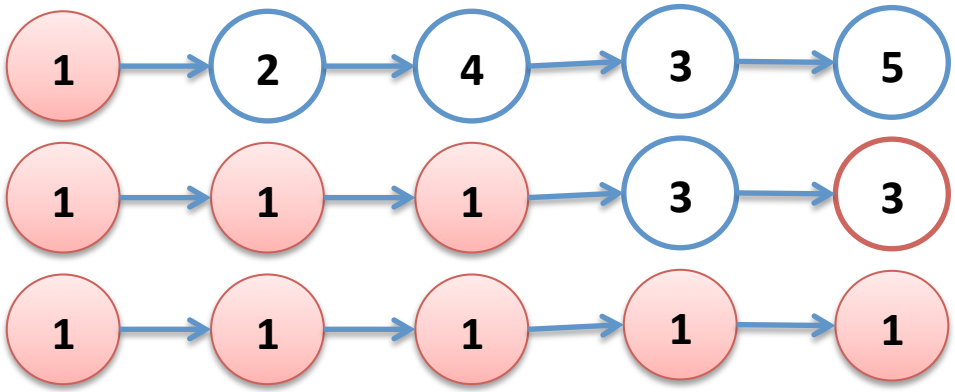
Synchronous (Jacobi)



Each vertex chooses minimum label of neighbor.

Bulk-Synchronous: requires graph diameter –many iterations to propagate the minimum label.

PSW is Asynchronous (*Gauss-Seidel*)



Each vertex chooses minimum label of neighbor.

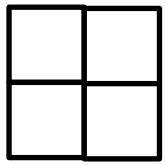
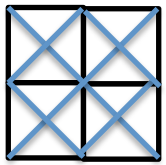
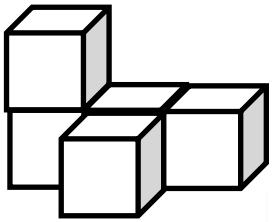
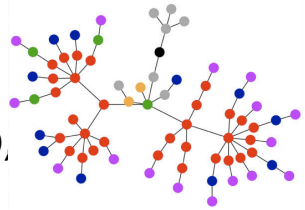
Gauss-Seidel: *expected* # iterations on random schedule on a chain graph
= (N - 1) / (e – 1)
≈ 60% of synchronous

Joint work: Julian Shun

Label Propagation

 # iterations

Side length = 100

	Synchronous	PSW: Gauss-Seidel (average, random schedule)
	199	~ 57
	100	~29
	298	~52
Natural graphs (web, social) 	graph diameter - 1	~ 0.6 * diameter

Open theoretical question!

PSW & External Memory Algorithms Research

- PSW is **a new technique for implementing many fundamental graph algorithms**
  - Especially simple (compared to previous work) for **directed graph problems**: PSW handles both in- and out-edges
- We propose new graph contraction algorithm based on PSW
  - Minimum-Spanning Forest & Connected Components
- … utilizing the Gauss-Seidel "acceleration"

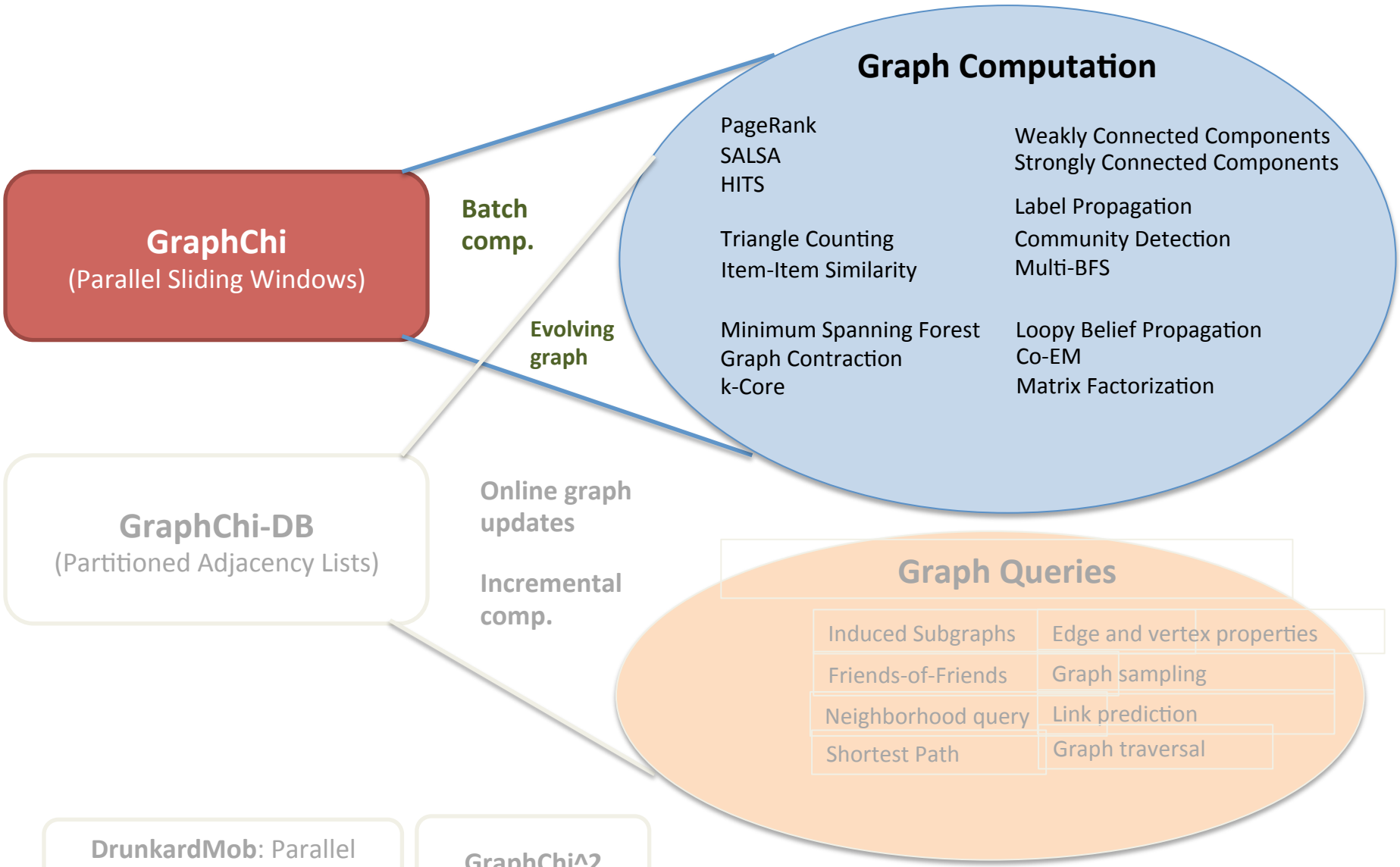
SEA 2014, Chapter 6 + Appendix

Consult the paper for a comprehensive evaluation:

- *HD vs. SSD*
- *Striping data across multiple hard drives*
- *Comparison to an in-memory version*
- *Bottlenecks analysis*
- *Effect of the number of shards*
- *Block size and performance.*

Sneak peek

GRAPHCHI: SYSTEM EVALUATION


GraphChi
(Parallel Sliding Windows)
Batch comp.
Evolving graph
Graph Computation
PageRank
SALSA
HITS
Triangle Counting
Item-Item Similarity
Minimum Spanning Forest
Graph Contraction
k-Core
Weakly Connected Components
Strongly Connected Components
Label Propagation
Community Detection
Multi-BFS
Loopy Belief Propagation
Co-EM
Matrix Factorization
GraphChi-DB
(Partitioned Adjacency Lists)
Online graph updates
Incremental comp.
Graph Queries
Induced Subgraphs
Edge and vertex properties
Friends-of-Friends
Graph sampling
Neighborhood query
Link prediction
Shortest Path
Graph traversal
DrunkardMob: Parallel Random walk simulation
GraphChi^2

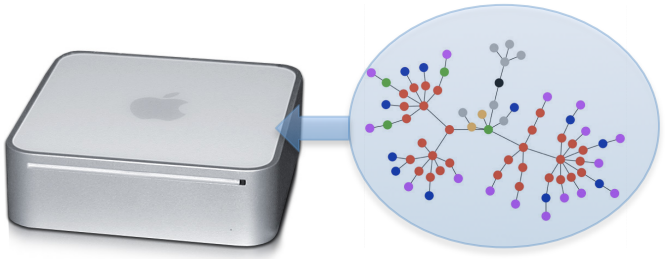
GraphChi

- C++ implementation: 8,000 lines of code
  - Java-implementation also available
- Several optimizations to PSW (see paper).

Source code and examples:
http://github.com/graphchi

Experiment Setting

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



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

See the paper for more comparisons.

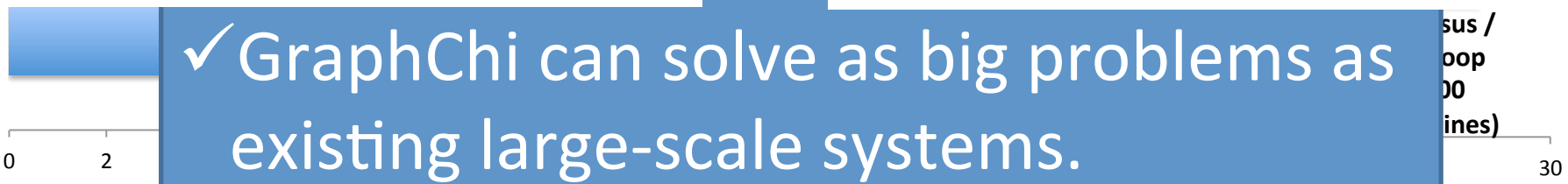
Comparison to Existing Systems

PageRank

WebGraph Belief Propagation (U Kang et al.)

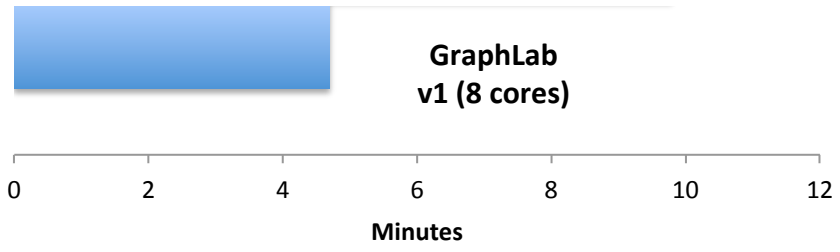
Twitter-2010 (1.5B edges)

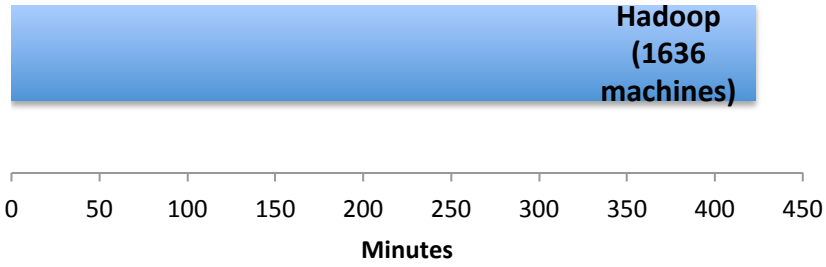
Yahoo-web (6.7B edges)



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

Matrix





Notes: comparison results do not include time to transfer the data to cluster, preprocessing, or the time to load the graph from disk. GraphChi computes asynchronously, while all but GraphLab synchronously.

PowerGraph Comparison

- **PowerGraph / GraphLab 2** outperforms previous systems by a wide margin on natural graphs.
- With 64 more machines, 512 more CPUs:
  - **Pagerank**: 40x faster than GraphChi
  - **Triangle counting**: 30x faster than GraphChi.

vs.

GraphChi

GraphChi has good performance / CPU.

In-memory Comparison

- **Total runtime** comparison to 1-shard GraphChi, with initial load + output write taken into account

Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Commun			
Matrix fa			
Matrix fa			

However, sometimes better algorithm available for in-memory than external memory / distributed.

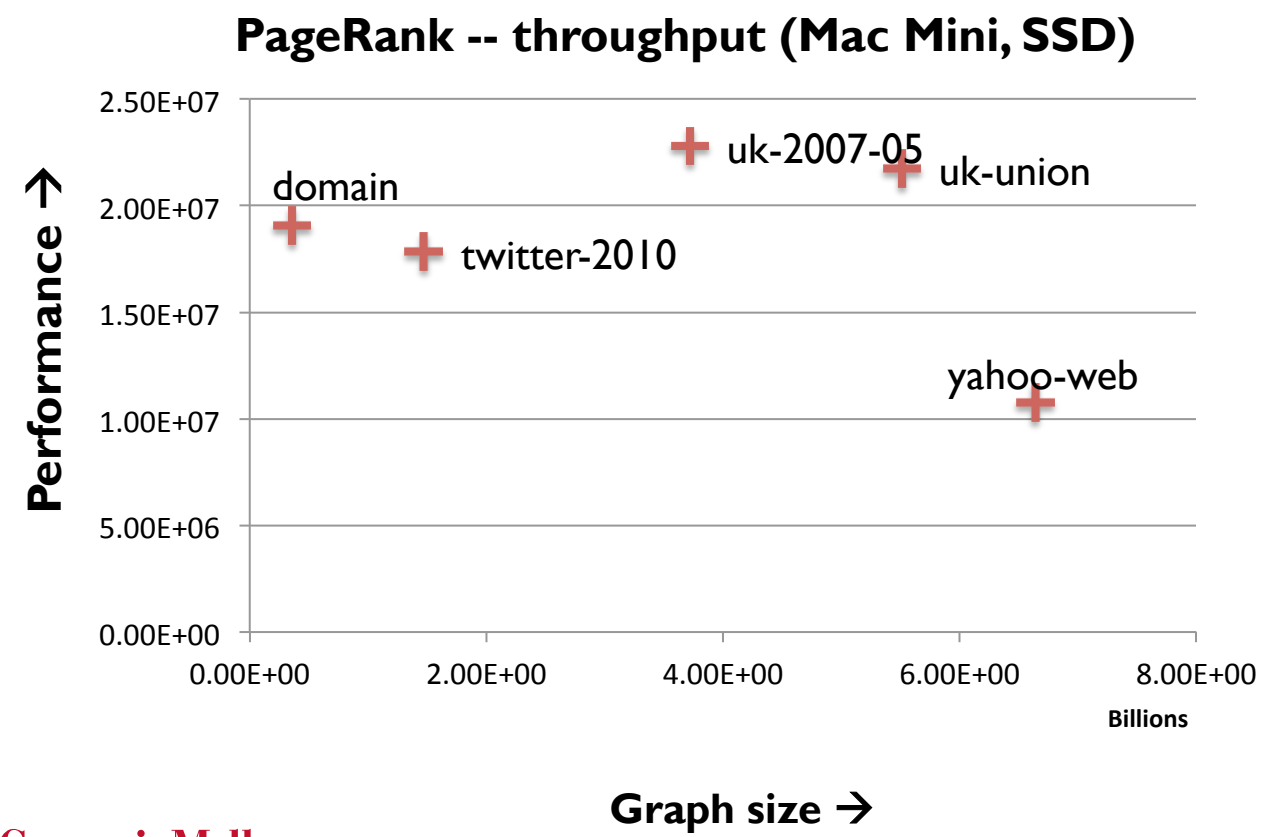
- Comparison
  - - 5 iterations of Pageranks / Twitter (1.5B edges)

GraphChi	Mac Mini – SSD	790 secs
Ligra (J. Shun, Blelloch)	40-core Intel E7-8870	15 secs

Ligra (J. Shun, Blelloch)	8-core Xeon 5550	230 s + preproc 144 s
PSW – inmem version, 700 shards (see Appendix)	8-core Xeon 5550	100 s + preproc 210 s

Scalability / Input Size [SSD]

- Throughput: number of edges processed / second.

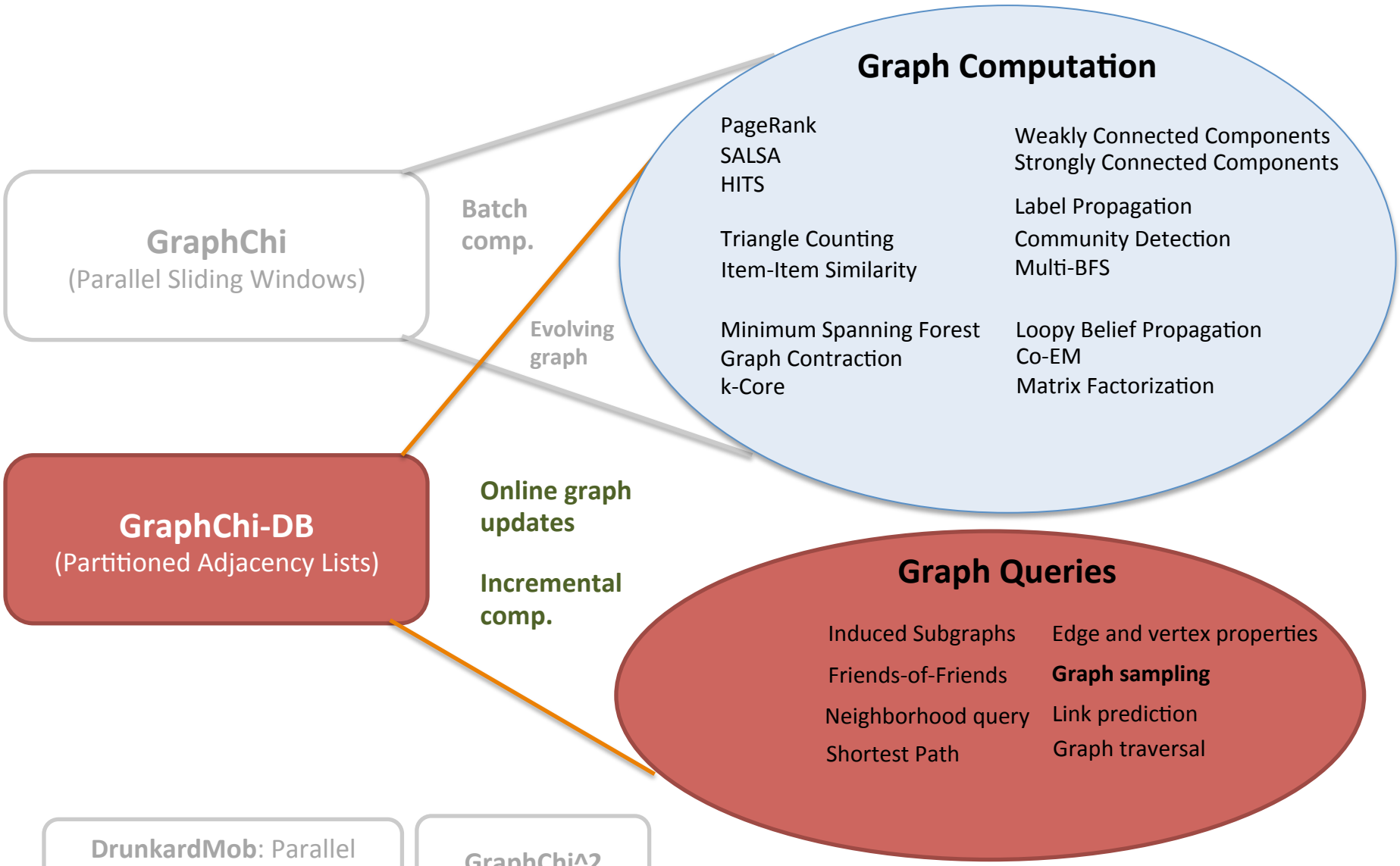


Conclusion: the throughput remains roughly constant when graph size is increased.

GraphChi with hard-drive is ~ 2x slower than SSD (if computational cost low).

New work

GRAPHCHI-DB


GraphChi
(Parallel Sliding Windows)
Batch comp.
Evolving graph
Graph Computation
PageRank
SALSA
HITS
Triangle Counting
Item-Item Similarity
Minimum Spanning Forest
Graph Contraction
k-Core
Weakly Connected Components
Strongly Connected Components
Label Propagation
Community Detection
Multi-BFS
Loopy Belief Propagation
Co-EM
Matrix Factorization
GraphChi-DB
(Partitioned Adjacency Lists)
Online graph updates
Incremental comp.
Graph Queries
Induced Subgraphs
Friends-of-Friends
Neighborhood query
Shortest Path
Edge and vertex properties
Graph sampling
Link prediction
Graph traversal
DrunkardMob: Parallel Random walk simulation
GraphChi^2

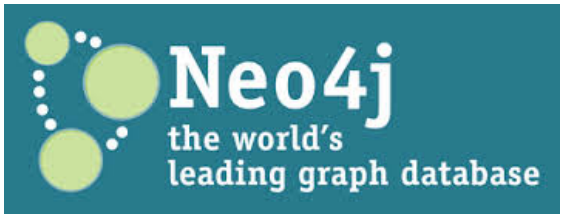
Research Questions

- What if there is lot of metadata associated with edges and vertices?
- How to do graph queries efficiently while retaining computational capabilities?
- How to add edges efficiently to the graph?

Can we design a graph *database* based on GraphChi?

Existing Graph Database Solutions

1) Specialized single-machine graph databases



Problems:
- Poor performance with data >> memory
- No/weak support for analytical computation

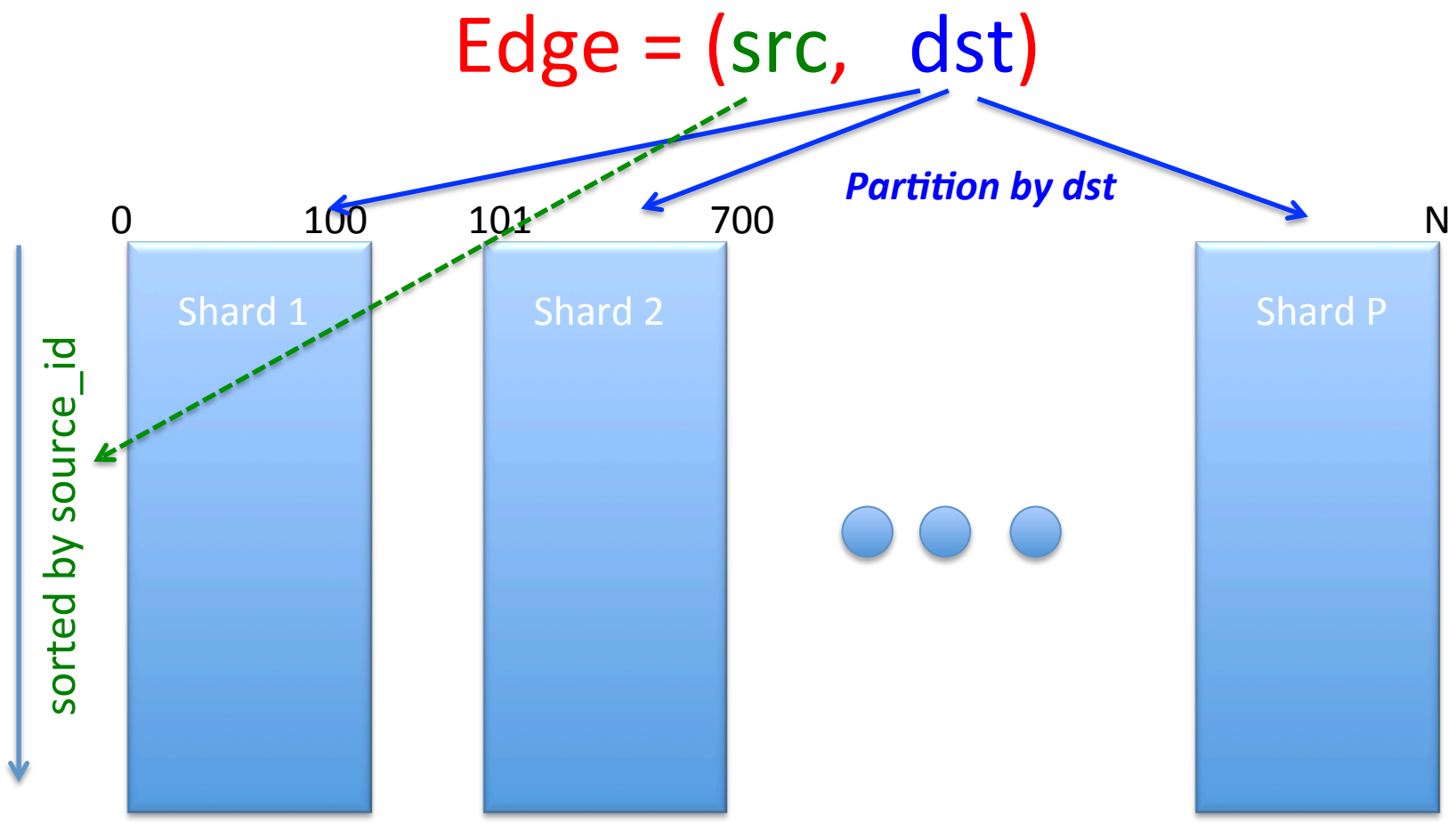
2) Relational / key-value databases as graph storage

Problems:
- Large indices
- In-edge / out-edge dilemma
- No/weak support for analytical computation

Our solution

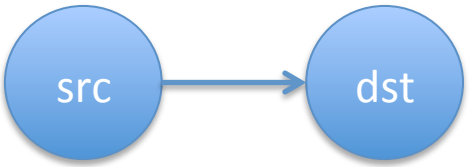
PARTITIONED ADJACENCY LISTS (PAL): DATA STRUCTURE

Review: Edges in Shards



Shard Structure (Basic)

Source	Destination
1	8
1	193
1	76420
3	12
3	872
7	193
7	212
7	89139
....	....



Shard Structure (Basic)

Compressed Sparse Row (CSR)

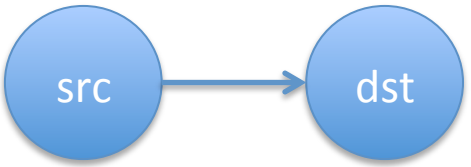
Problem 1:

How to find in-edges of a vertex quickly?

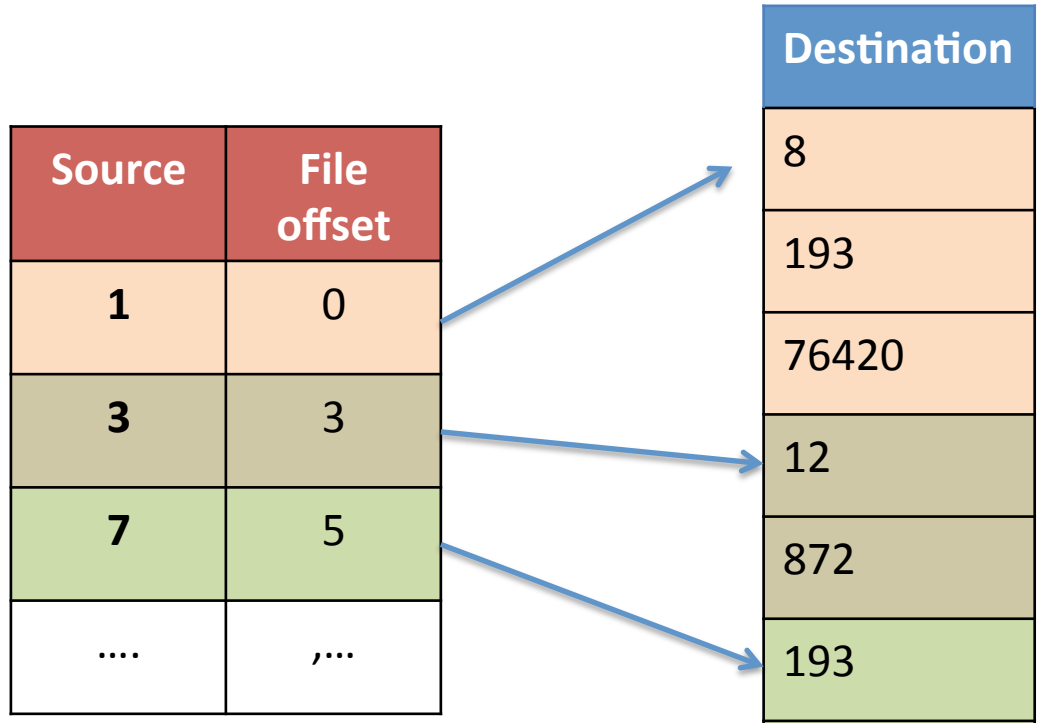
Note: We know the shard, but edges in random order.

Destination
8
193
76420
12
872
193
212
89139
....

Edge-array

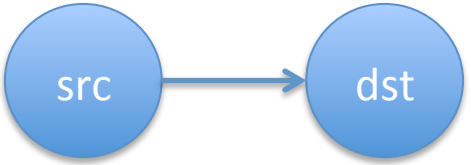


PAL: In-edge Linkage



Pointer-array

Edge-array

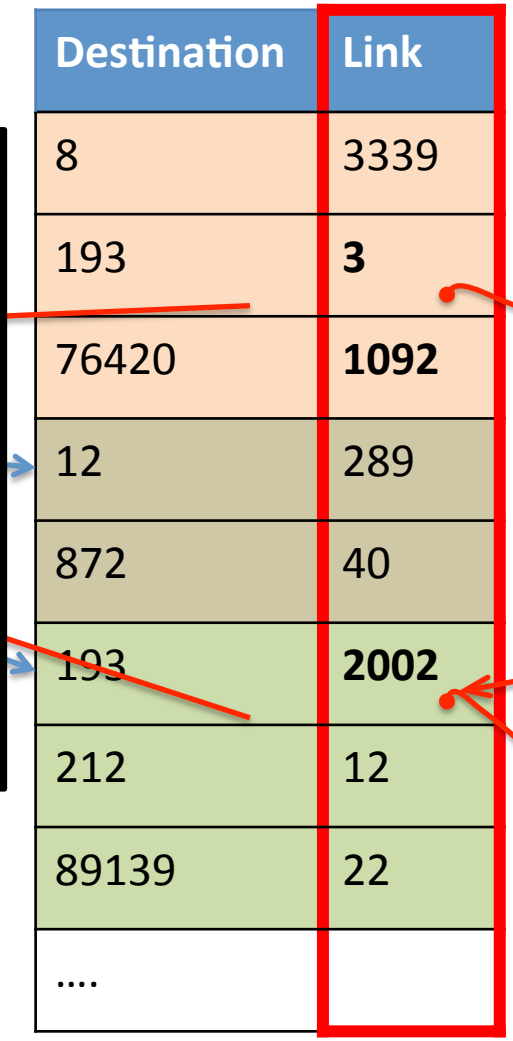


PAL: In-edge Linkage

Problem 2:

How to find out-edges quickly?

Note: Sorted inside a shard, but partitioned across all shards.



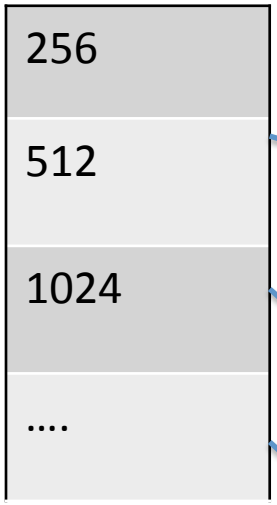
Edge-array

+ Index to the first in-edge for each vertex in interval.

Augmented linked list for in-edges

PAL: Out-edge Queries

Option 1:
Sparse index (inmem)



Problem: Can be big -- O(V)
→ Binary search on disk slow.

Source	File offset
1	0
3	3
7	5
....	,...

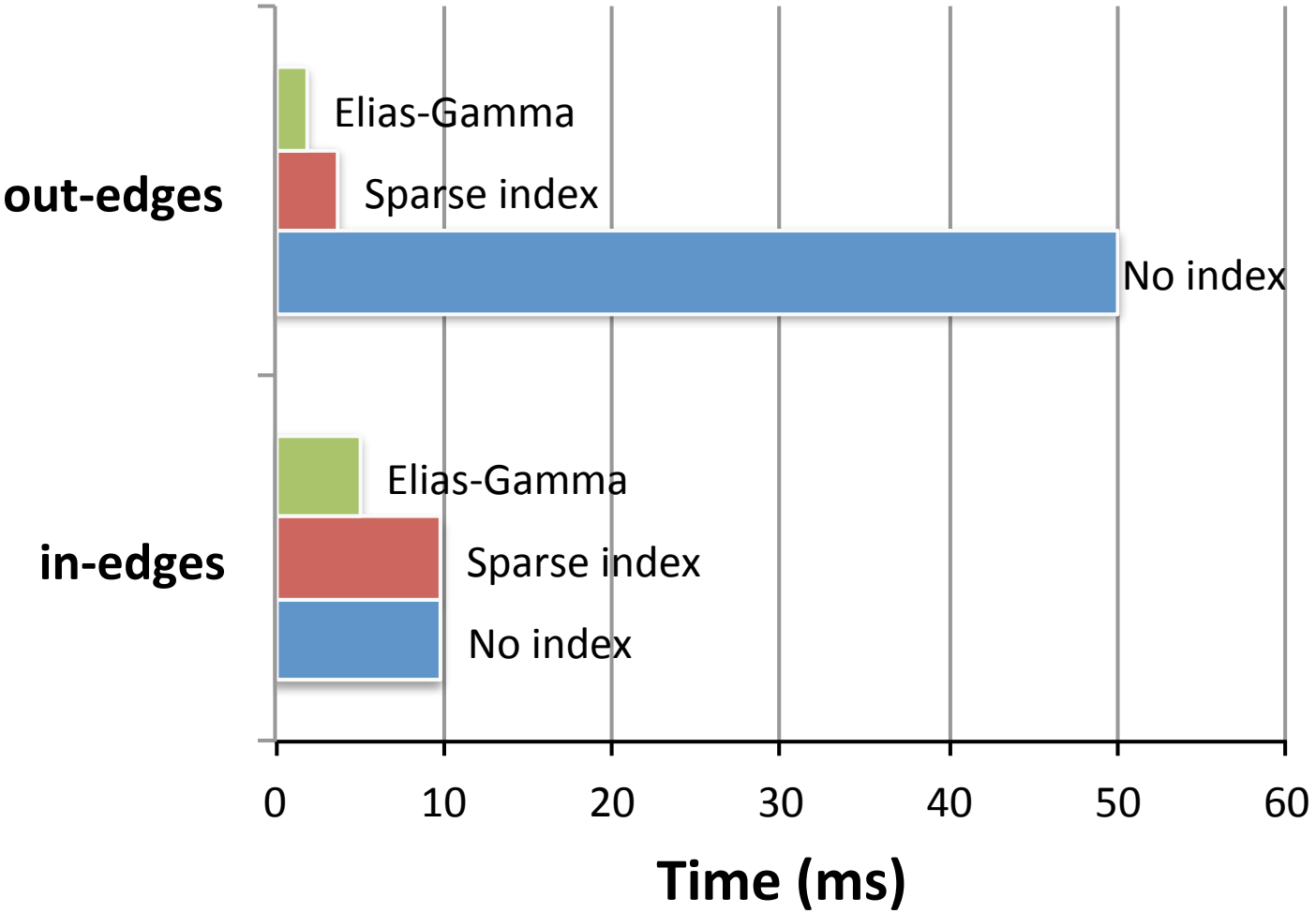
Pointer-array

Destination	Next-in-offset
8	3339
193	**3**
76420	**1092**
12	289
872	40
193	**2002**
212	12
89139	22
....	

Edge-array

Option 2:
Delta-coding
with unary code
(Elias-Gamma)
→ **Completely in-memory**

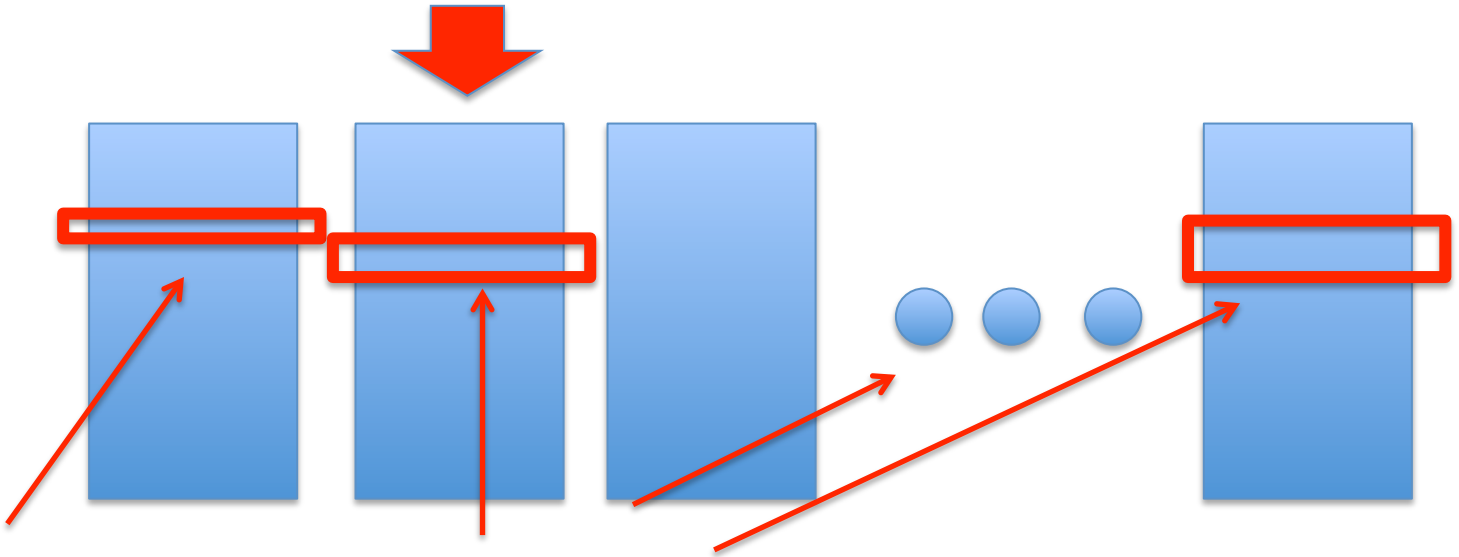
Experiment: Indices



Median latency, Twitter-graph, 1.5B edges

Queries: I/O costs

In-edge query: only one shard



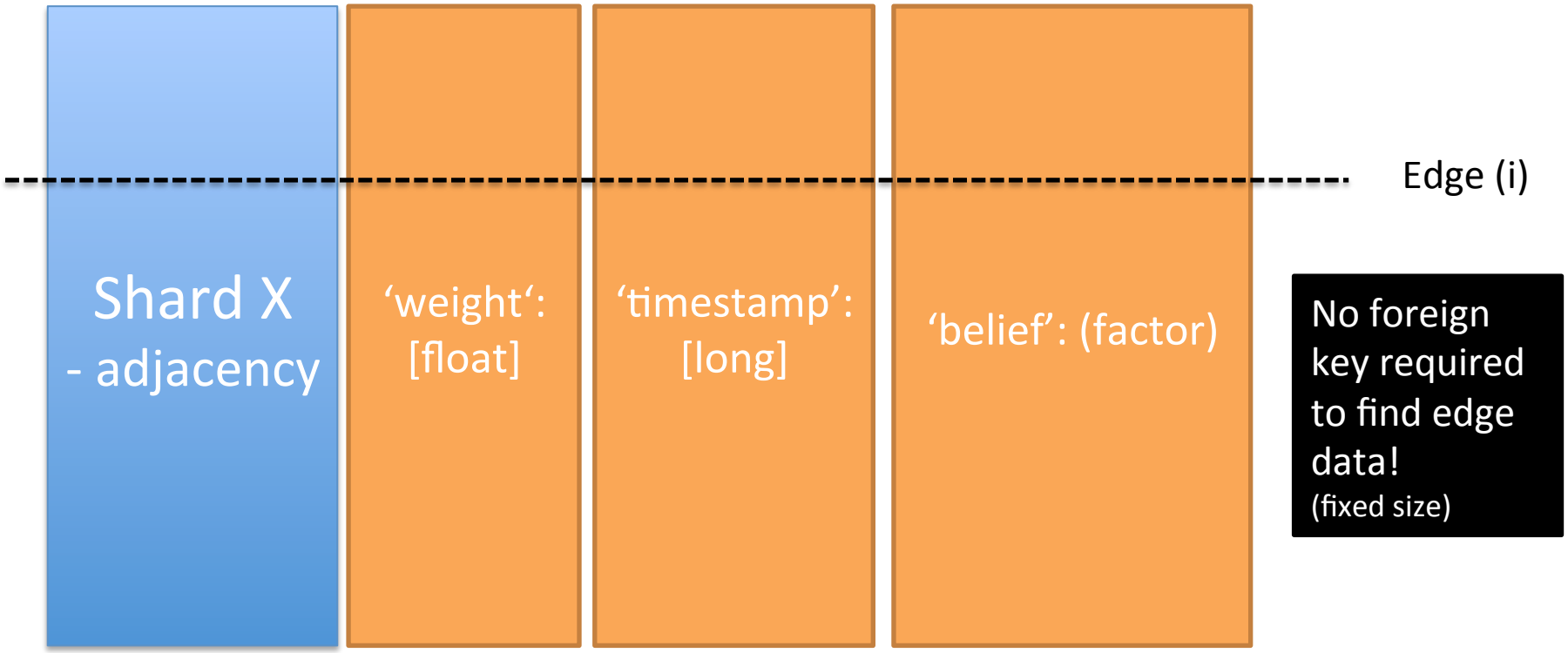
Out-edge query: each shard that has edges

Trade-off:
More shards → Better locality for in-edge queries, worse for out-edge queries.

$$\text{io-cost}[\text{inquery(v)}] \leq 1 + \min\left(\text{indeg(v)}, \frac{E}{PB}\right)$$

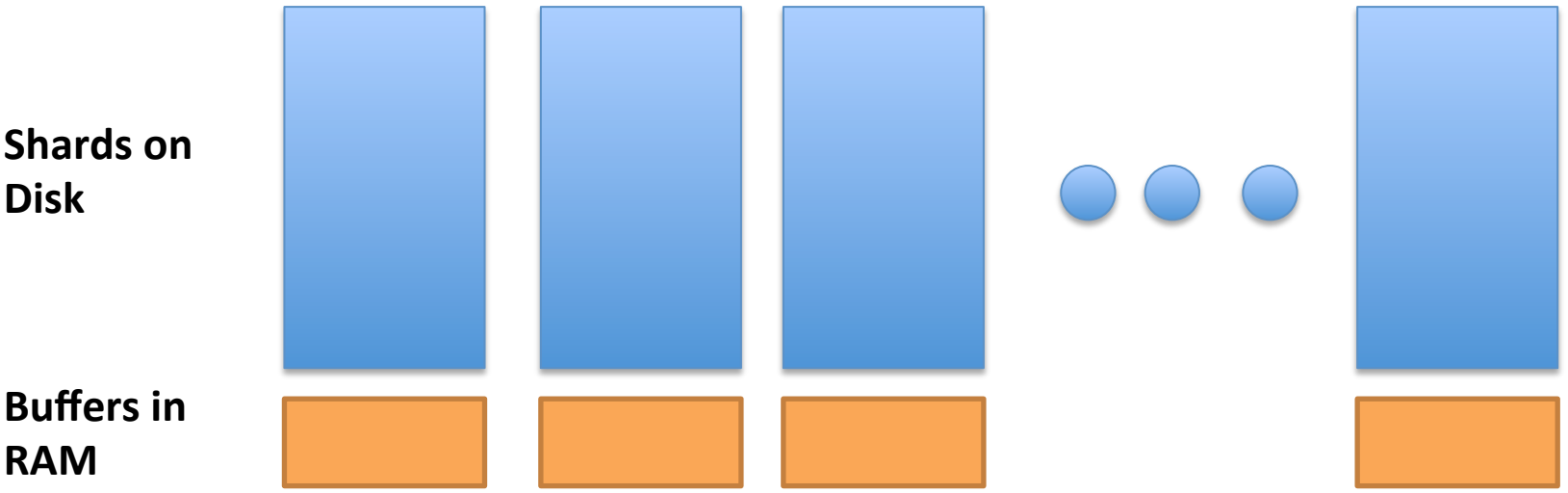
$$\text{io-cost}[\text{outquery(v)}] \leq \min(P, \text{outdeg(v)}) + \lfloor \frac{\text{outdeg(v)}}{B} \rfloor$$

Edge Data & Searches

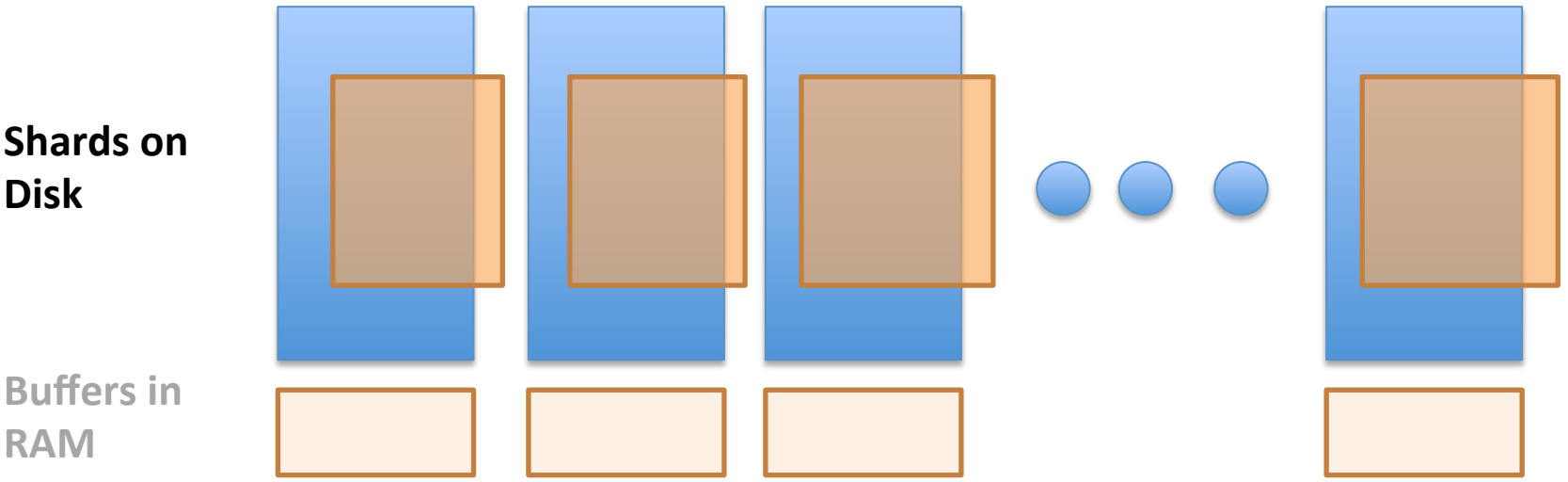


Note: vertex values stored similarly.

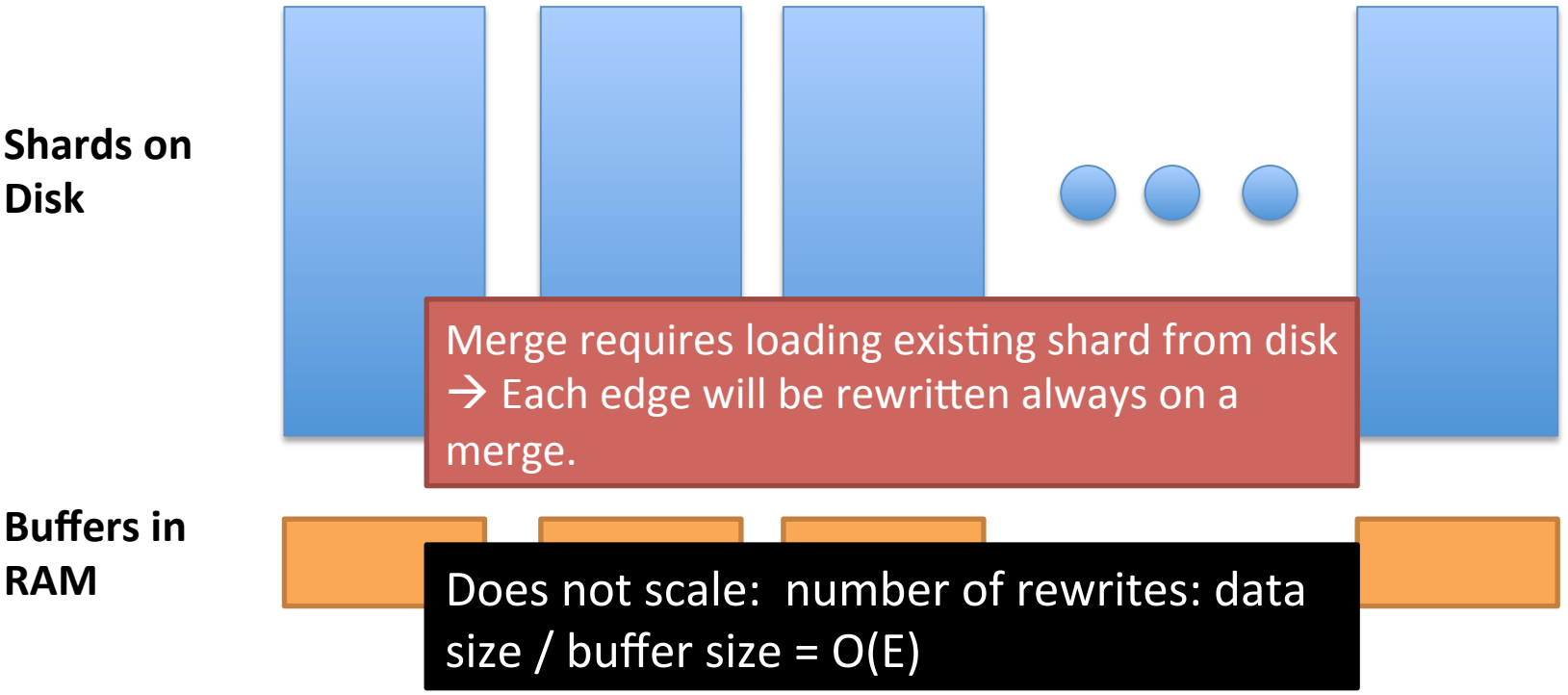
Efficient Ingest?



Merging Buffers to Disk

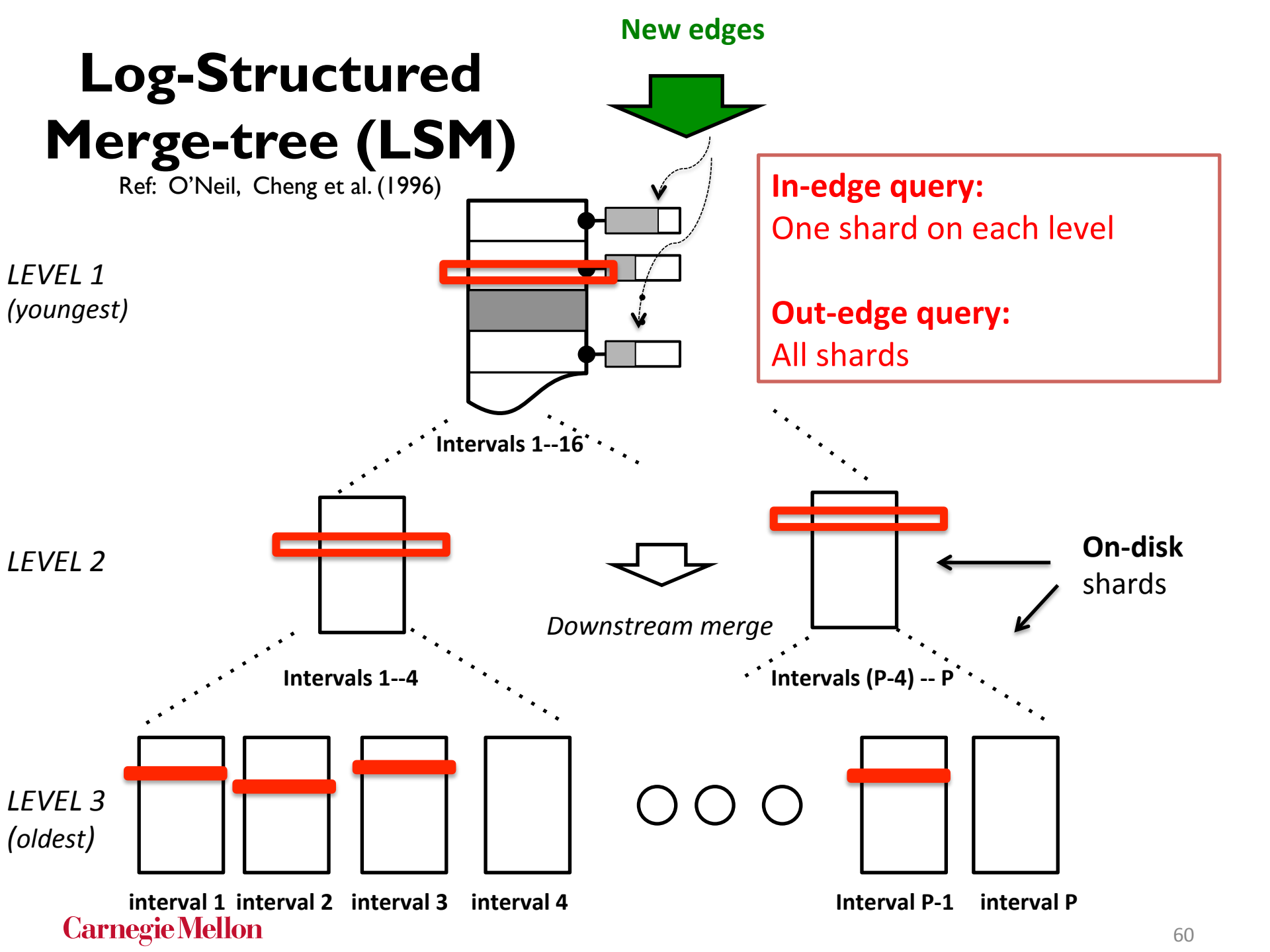


Merging Buffers to Disk (2)

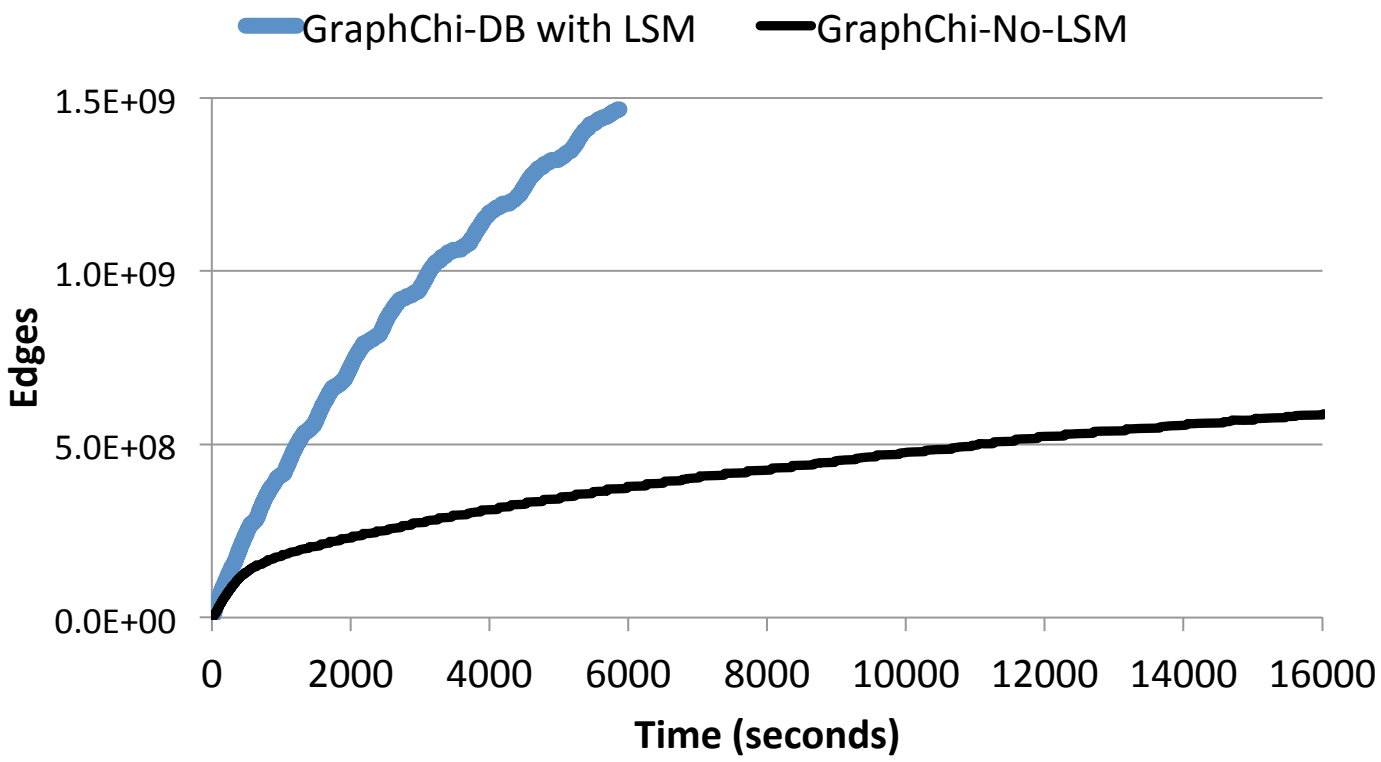


Log-Structured Merge-tree (LSM)

Ref: O'Neil, Cheng et al. (1996)



Experiment: Ingest



Advantages of PAL

- Only sparse and implicit indices
  - Pointer-array usually fits in RAM with Elias-Gamma.

  → Small database size.
- Columnar data model
  - Load only data you need.
  - Graph structure is separate from data.
  - *Property graph model*
- Great insertion throughput with LSM

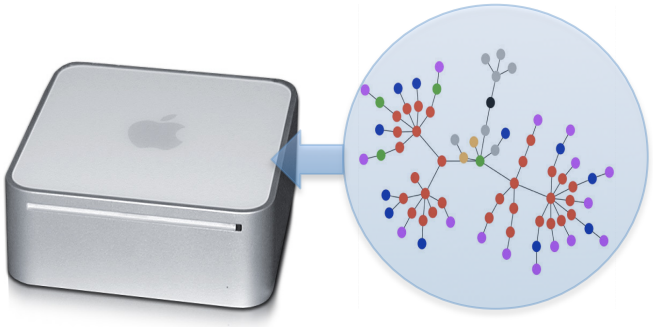
  →Tree can be adjusted to match workload.

EXPERIMENTAL COMPARISONS

GraphChi-DB: Implementation

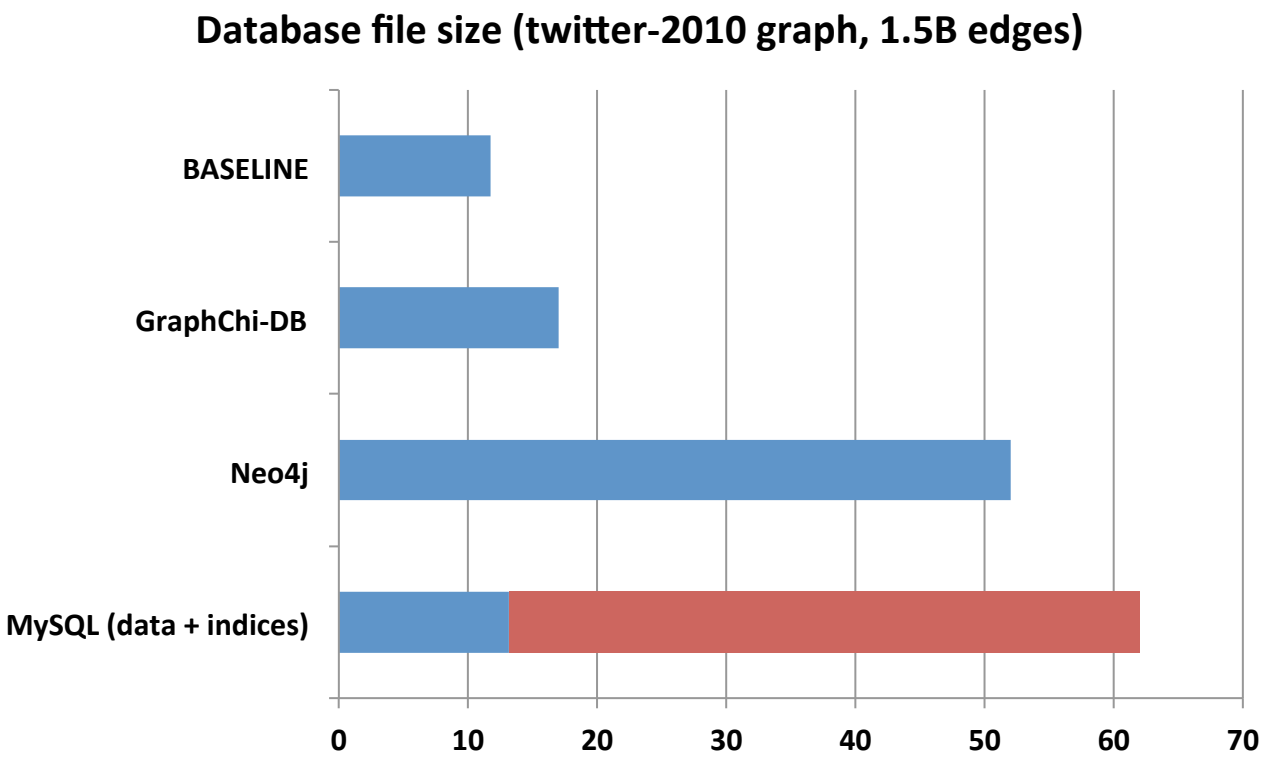
- Written in Scala
- Queries & Computation
- Online database

All experiments shown in this talk done on Mac Mini (8 GB, SSD)



Source code and examples: http://github.com/graphchi

Comparison: Database Size



Baseline: 4 + 4 bytes / edge.

Comparison: Ingest

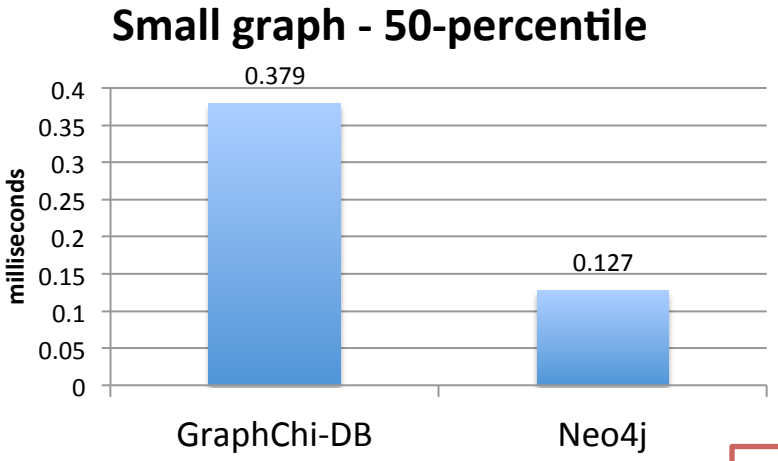
System	Time to ingest 1.5B edges
GraphChi-DB (ONLINE)	**1 hour 45 mins**
Neo4j (batch)	45 hours
MySQL (batch)	3 hour 30 minutes (including index creation)

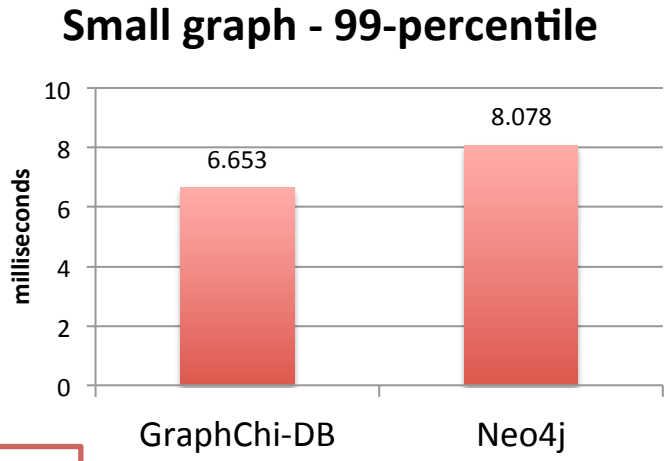
If running **Pagerank simultaneously**, GraphChi-DB takes 3 hour 45 minutes

See thesis for **shortest-path** comparison.

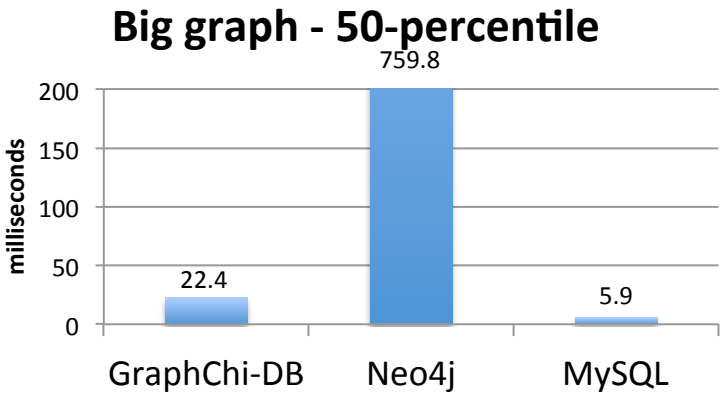
Comparison: Friends-of-Friends Query

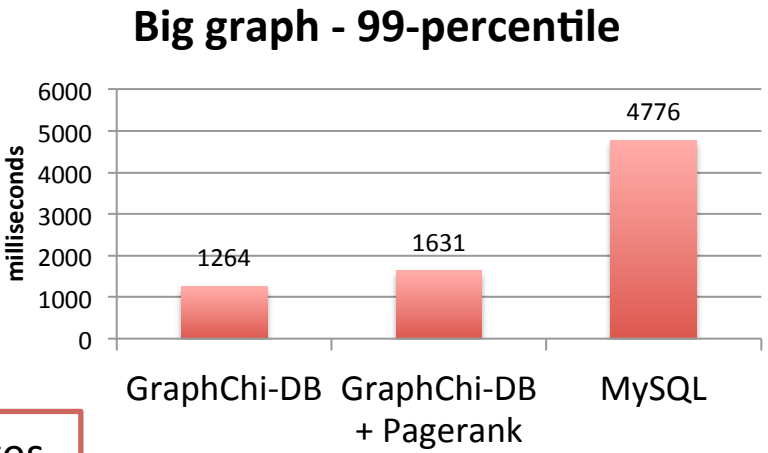
Latency percentiles over 100K random queries





68M edges





1.5B edges

LinkBench: Online Graph DB Benchmark by Facebook

- Concurrent read/write workload
  - But only single-hop queries ("friends").
  - 8 different operations, mixed workload.
  - Best performance with 64 parallel threads
- Each edge and vertex has:
  - Version, timestamp, type, random string payload.

	GraphChi-DB (Mac Mini)	MySQL+FB patch, server, SSD-array, 144 GB RAM
Edge-update (95p)	22 ms	25 ms
Edge-get-neighbors (95p)	18 ms	9 ms
Avg throughput	**2,487 req/s**	**11,029 req/s**
Database size	*350 GB*	*1.4 TB*

See full results in the thesis.

Summary of Experiments

- Efficient for **mixed read/write** workload.
  - See Facebook LinkBench experiments in thesis.
  - LSM-tree → trade-off read performance (**but, can adjust).**
- State-of-the-art performance for graphs that **are much larger than RAM.**
  - Neo4J's linked-list data structure good for RAM.
  - DEX performs poorly in practice.

More experiments in the thesis!

Discussion

Greater Impact

Hindsight

Future Research Questions

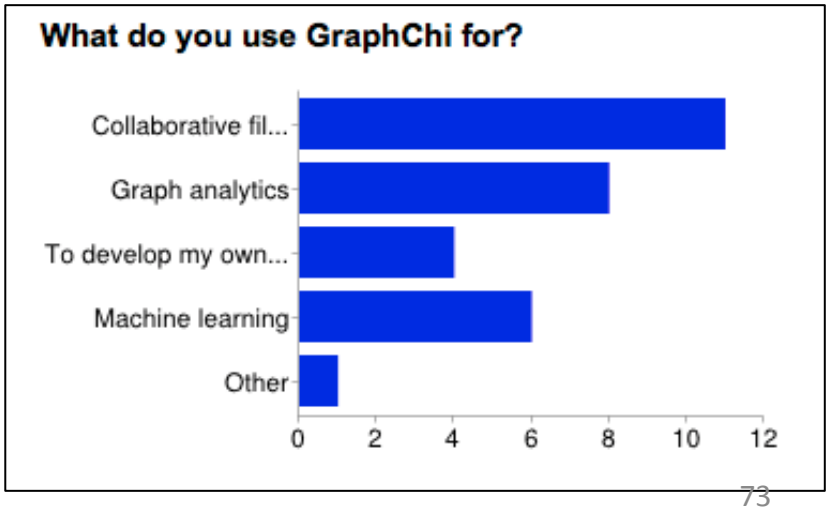
GREATER IMPACT

Impact: "Big Data" Research

- GraphChi's OSDI 2012 paper has received over 85 citations in just 18 months (Google Scholar).
  - Two major direct descendant papers in top conferences:
    - X-Stream: SOSP 2013
    - TurboGraph: KDD 2013
- Challenging the mainstream:
  - You can do a lot on just a PC → focus on right data structures, computational models.

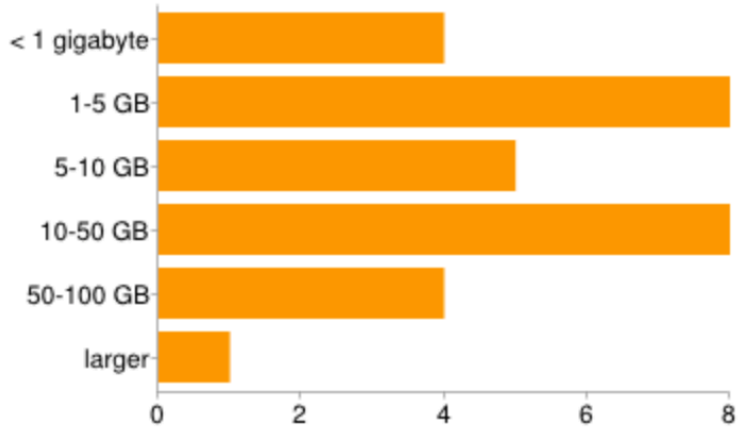
Impact: Users

- GraphChi's (C++, Java, -DB) have gained a lot of users
  - Currently ~50 unique visitors / day.
- Enables 'everyone' to tackle big graph problems
  - Especially the recommender toolkit (by Danny Bickson) has been very popular.
  - Typical users: students, non-systems researchers, small companies…

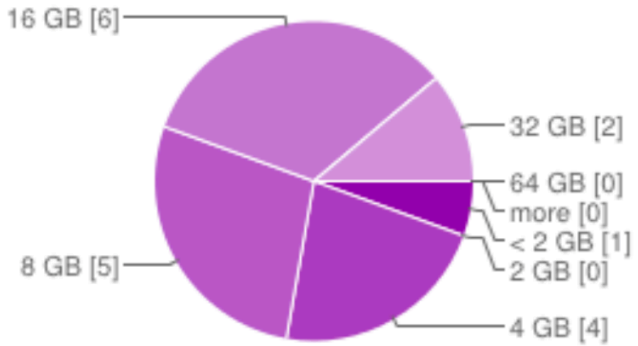


Impact: Users (cont.)

How big datasets do you use?



How much memory does your computer (that you use for GraphChi) have?



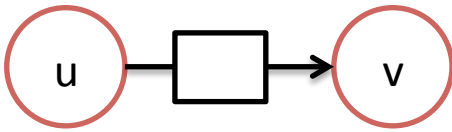
*I work in a [EU country] public university. I **can't use a a distributed computing cluster** for my research ... it **is too expensive.***
Using GraphChi I was able to perform my experiments on my laptop. I thus have to admit that GraphChi saved my research. (...)

EVALUATION: HINDSIGHT

What is GraphChi Optimized for?

- Original target algorithm: **Belief Propagation** on Probabilistic Graphical Models.

1. Changing value of an edge (both in- and out!).
2. Computation process whole, or most of the graph on each iteration.
3. Random access to all vertex's edges.
   - Vertex-centric vs. edge centric.
4. Async/Gauss-Seidel execution.

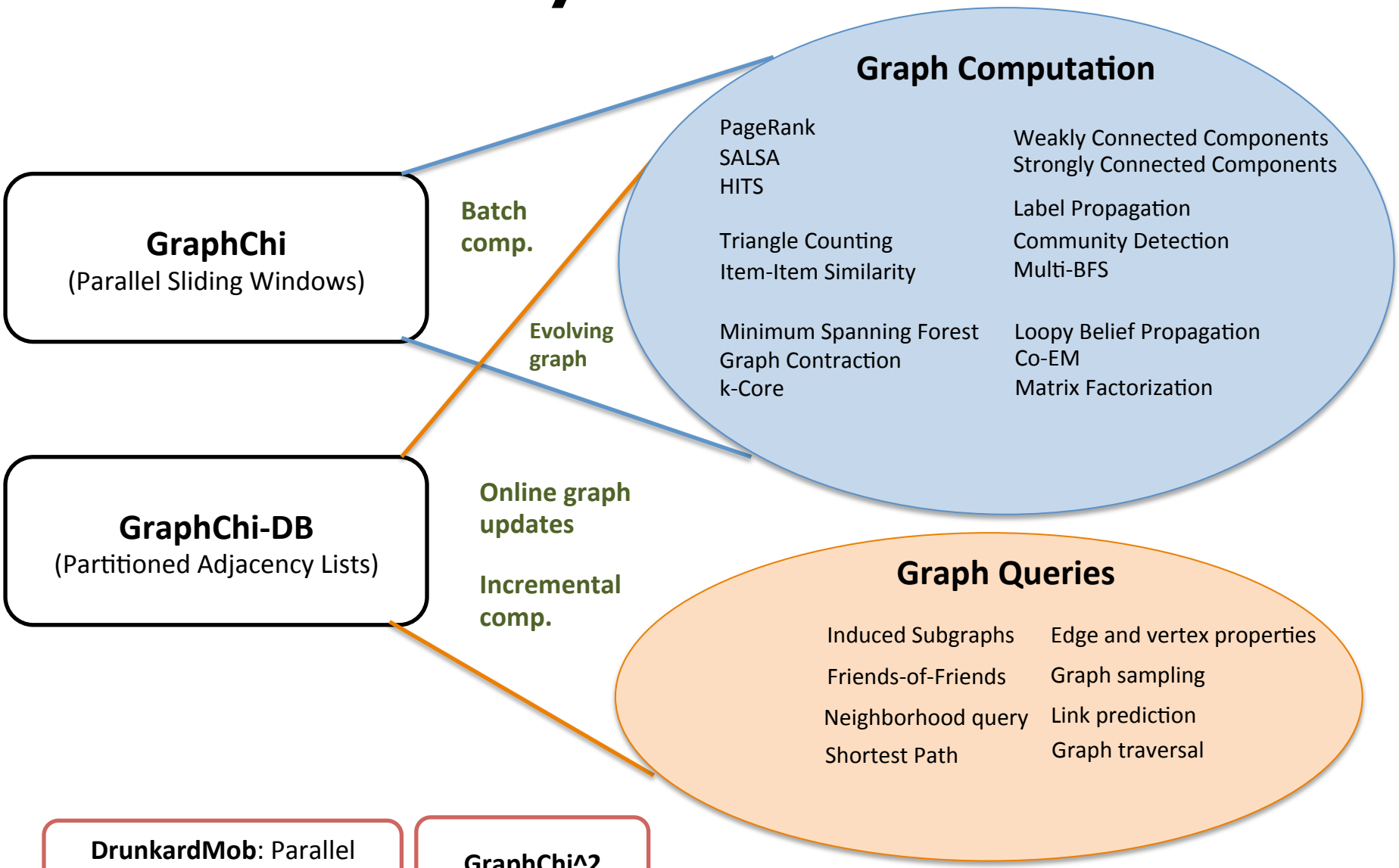


GraphChi Not Good For

- Very large vertex state.
- Traversals, and two-hop dependencies.
  - Or dynamic scheduling (such as Splash BP).
- High diameter graphs, such as planar graphs.
  - Unless the computation itself has short-range interactions.
- Very large number of iterations.
  - Neural networks.
  - LDA with Collapsed Gibbs sampling.
- No support for implicit graph structure.

+ Single PC performance is limited.

Versatility of PSW and PAL



Future Research Directions

- Distributed Setting
  1. Distributed PSW (one shard / node)
     1. PSW is inherently sequential
     2. Low bandwidth in the Cloud
  2. Co-operating GraphChi(-DB)'s connected with a Parameter Server
- New graph programming models and Tools
  - Vertex-centric programming sometimes too local: for example, two-hop interactions and many traversals cumbersome.
  - Abstractions for learning graph structure; Implicit graphs.
  - Hard to debug, especially async → Better tools needed.
- Graph-aware optimizations to GraphChi-DB.
  - Buffer management.
  - Smart caching.
  - Learning configuration.

Semi-External Memory Setting

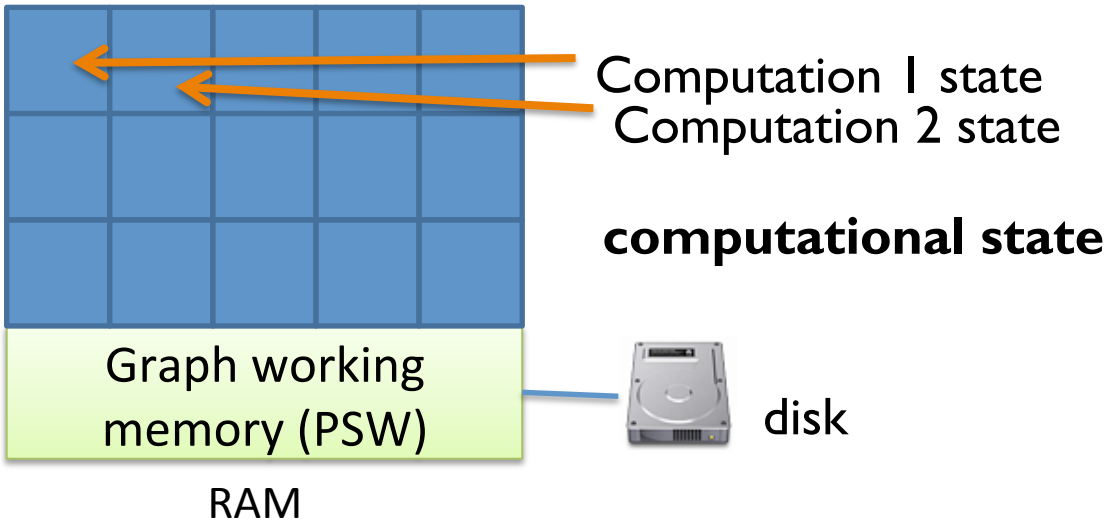
WHAT IF WE HAVE PLENTY OF MEMORY?

Observations

- The I/O performance of PSW is only weakly affected by the amount of RAM.
  - Good: works with very little memory.
  - Bad: Does not benefit from more memory
    - Simple trick: cache some data.
- Many graph algorithms have O(V) state.
  - Update function accesses neighbor vertex state.
    - Standard PSW: 'broadcast' vertex value via edges.
    - Semi-external: Store vertex values in memory.

Using RAM efficiently

- Assume that enough RAM to store many O(V) algorithm states in memory.
  - But not enough to store the whole graph.



Parallel Computation Examples

- **DrunkardMob** algorithm (Chapter 5):
  - Store billions of random walk states in RAM.
- **Multiple Breadth-First-Searches:**
  - Analyze neighborhood sizes by starting hundreds of random BFSes.
- Compute in parallel **many different recommender algorithms** (or with different parameterizations).
  - See Mayank Mohta, Shu-Hao Yu's Master's project.

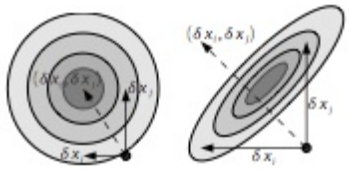
CONCLUSION

Summary of Published Work

GraphLab: Parallel Framework for Machine Learning (with J. Gonzaled, Y.Low, D. Bickson, C.Guestrin)	UAI 2010
Distributed GraphLab: Framework for Machine Learning and Data Mining in the Cloud (-- same --)	VLDB 2012
GraphChi: Large-scale Graph Computation on Just a PC (with C.Guestrin, G. Blelloch)	**OSDI 2012**
DrunkardMob: Billions of Random Walks on Just a PC	**ACM RecSys 2013**
Beyond Synchronous: New Techniques for External Memory Graph Connectivity and Minimum Spanning Forest (with Julian Shun, G. Blelloch)	**SEA 2014**
GraphChi-DB: Simple Design for a Scalable Graph Database – on Just a PC (with C. Guestrin)	(submitted / arxiv)
Parallel Coordinate Descent for L1-regularized :Loss Minimization (Shotgun) (with J. Bradley, D. Bickson, C.Guestrin)	**ICML 2011**

GraphLab

THESIS



First author papers in bold.

Summary of Main Contributions

- Proposed Parallel Sliding Windows, a new algorithm for external memory graph computation → GraphChi
- Extended PSW to design Partitioned Adjacency Lists to build a scalable graph database → GraphChi-DB
- Proposed DrunkardMob for simulating billions of random walks in parallel.
- Analyzed PSW and its Gauss-Seidel properties for fundamental graph algorithms → New approach for EM graph algorithms research.

Thank You!

ADDITIONAL SLIDES

Economics

Equal throughput configurations (based on OSDI'12)

	GraphChi (40 Mac Minis)	PowerGraph (64 EC2 cc1.4xlarge)
Investments	67,320 $	-
Operating costs		
Per node, hour	0.03 $	1.30 $
Cluster, hour	1.19 $	52.00 $
Daily	28.56 $	1,248.00 $

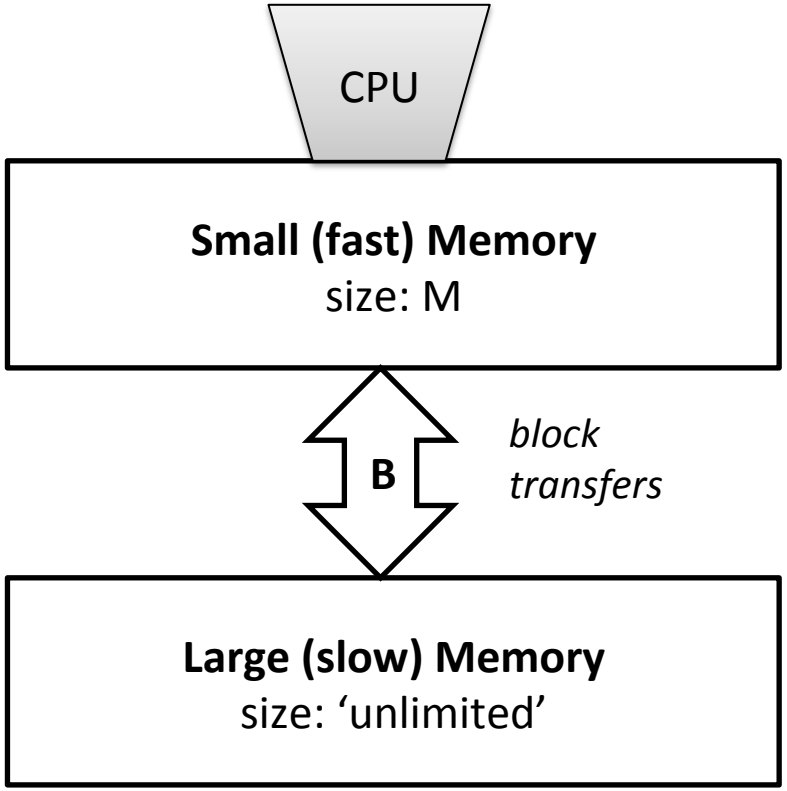
Assumptions:

- *Mac Mini: 85W (typical servers 500-1000W)*
- *Most expensive US energy: 35c / KwH*

It takes about 56 days to recoup Mac Mini investments.

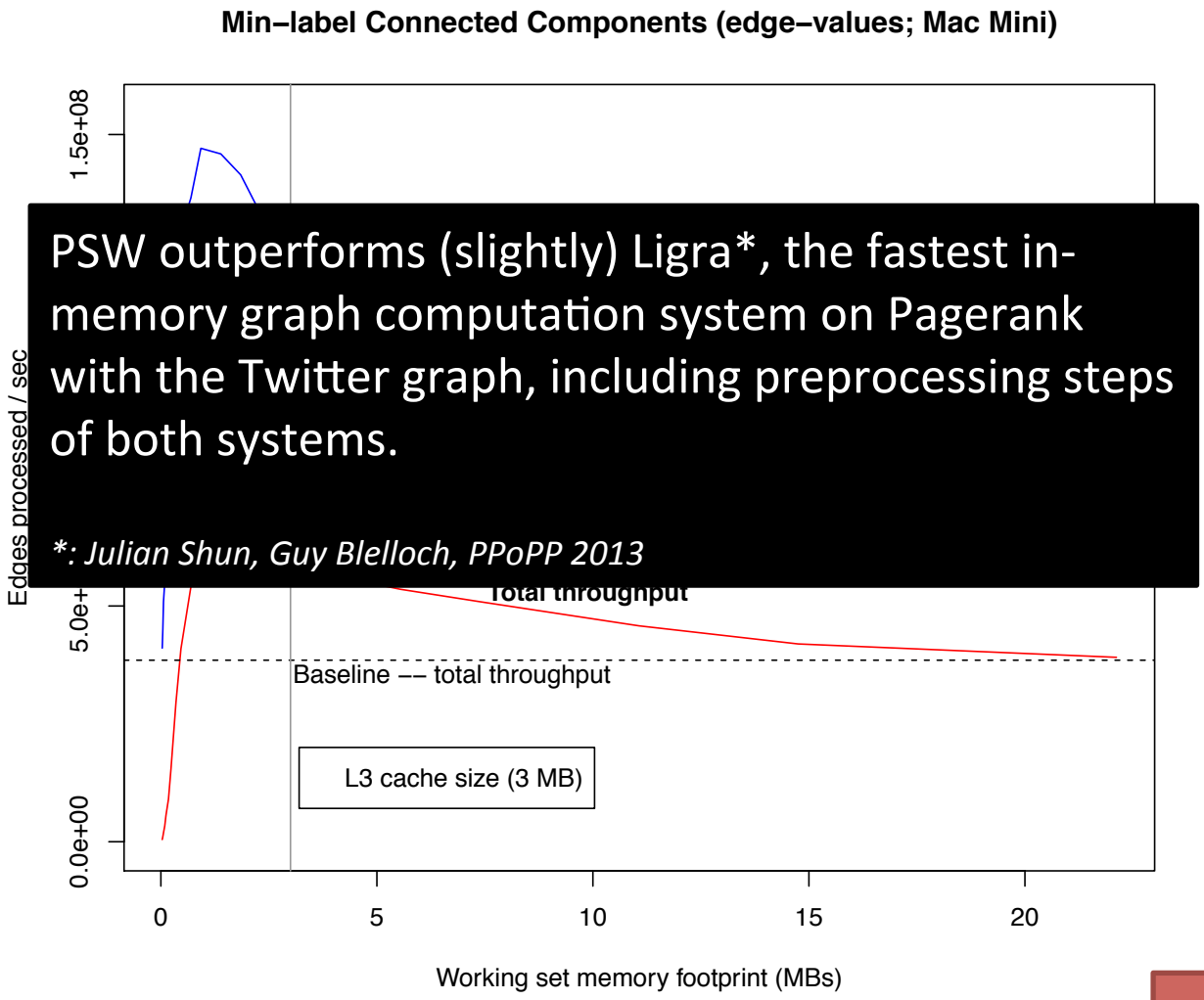
PSW for In-memory Computation

- External memory setting:
  - Slow memory = hard disk / SSD
  - Fast memory = RAM
- In-memory:
  - Slow = RAM
  - Fast = CPU caches



Does PSW help in the in-memory setting?

PSW for in-memory



PSW outperforms (slightly) Ligra*, the fastest in-memory graph computation system on Pagerank with the Twitter graph, including preprocessing steps of both systems.

**: Julian Shun, Guy Blelloch, PPoPP 2013*

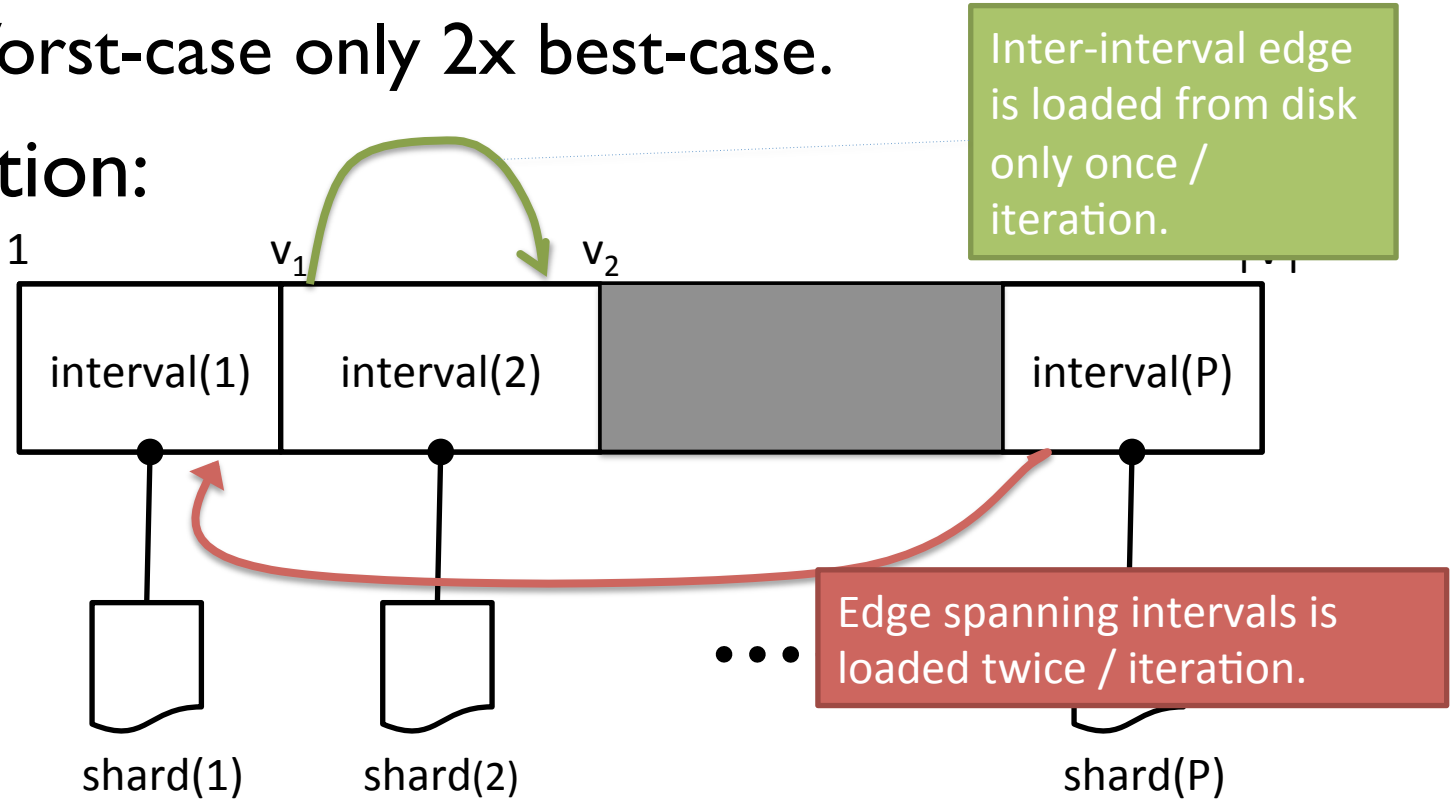
Appendix

Remarks (sync vs. async)

- Bulk-Synchronous is embarrassingly parallel
  - But needs twice the amount of space
- Async/G-S helps with high diameter graphs
- Some algorithms converge much better asynchronously
  - Loopy BP, see Gonzalez et al. (2009)
  - Also Bertsekas & Tsitsiklis *Parallel and Distributed Optimization (1989)*
- Asynchronous sometimes difficult to reason about and debug
- Asynchronous can be used to implement BSP

I/O Complexity

- See the paper for theoretical analysis in the Aggarwal-Vitter's I/O model.
  - Worst-case only 2x best-case.
- Intuition:



Impact of Graph Structure

- Algos with long range information propagation, need relatively small diameter → would require too many iterations
- Per iteration cost not much affected

- Can we optimize partitioning?
  - Could help thanks to Gauss-Seidel (faster convergence inside "groups") → topological sort
  - Likely too expensive to do on single PC

Graph Compression: Would it help?

- Graph Compression methods (e.g Blelloch et al., WebGraph Framework) can be used to compress edges to 3-4 bits / edge (web), ~ 10 bits / edge (social)
  - But require graph partitioning → requires a lot of memory.
  - Compression of large graphs can take days (personal communication).
- Compression problematic for evolving graphs, and associated data.
- GraphChi can be used to compress graphs?
  - Layered label propagation (Boldi et al. 2011)

Previous research on (single computer) Graph Databases

- 1990s, 2000s saw interest in object-oriented and graph databases:
  - GOOD, GraphDB, HyperGraphDB…
  - Focus was on modeling, graph storage on top of relational DB or key-value store
- RDF databases
  - Most do not use graph storage but store triples as relations + use indexing.
- Modern solutions have proposed graph-specific storage:
  - Neo4j: doubly linked list
  - TurboGraph: adjacency list chopped into pages
  - DEX: compressed bitmaps (details not clear)

LinkBench

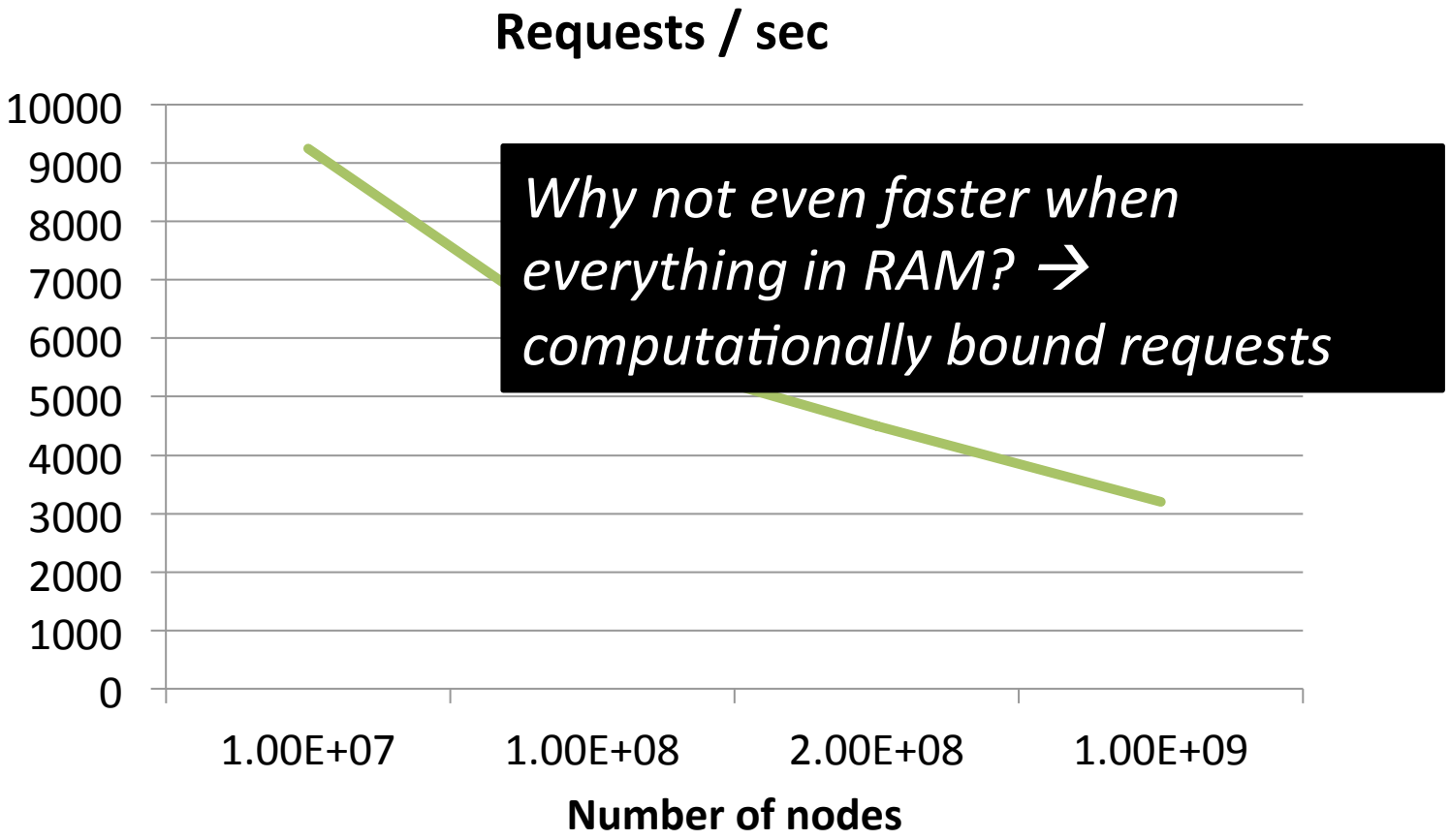
	GraphChi-DB *laptop (SSD)*			MySQL + FB patch *server (SSD-array) [9]*		
	p50	p75	p95	50	p75	p95
node_get	2	4	34	0.6	1	9
node_insert	0.1	0.1	0.1	3	5	12
node_update	2	4	34	3	6	14
edge_ins-or-upd.	0.7	2	15	7	14	25
edge_delete	0.1	0.9	7	1	7	19
edge_update	1	3	22	7	14	25
edge_getrange	8	19	250	1	1	10
edge_outnbrs	0.4	3	18	0.8	1	9
Avg throughput	2,487 req/s			11,029 req/s		

Table 4.2: LinkBench online database benchmark. Latencies are in milliseconds. Note: for clarity we have modified the request names from the original. JVM's garbage collection pauses cause the high 95-percentiles.

Comparison to FB (cont.)

- GraphChi load time 9 hours, FB's 12 hours
- GraphChi database about 250 GB, FB > 1.4 terabytes
  - However, about 100 GB explained by different variable data (payload) size
- Facebook/MySQL via JDBC, GraphChi embedded
  - But MySQL native code, GraphChi-DB Scala (JVM)
- Important CPU bound bottleneck in sorting the results for high-degree vertices

LinkBench: GraphChi-DB performance / Size of DB



Possible Solutions

1. Use SSD as a memory-extension? [SSDAlloc, NSDI'11]

Too many small objects, need millions / sec.

2. Compress the graph structure to fit into RAM? [→ WebGraph framework]

Associated values do not compress well, and are mutated.

3. Cluster the graph and handle each cluster separately in RAM?

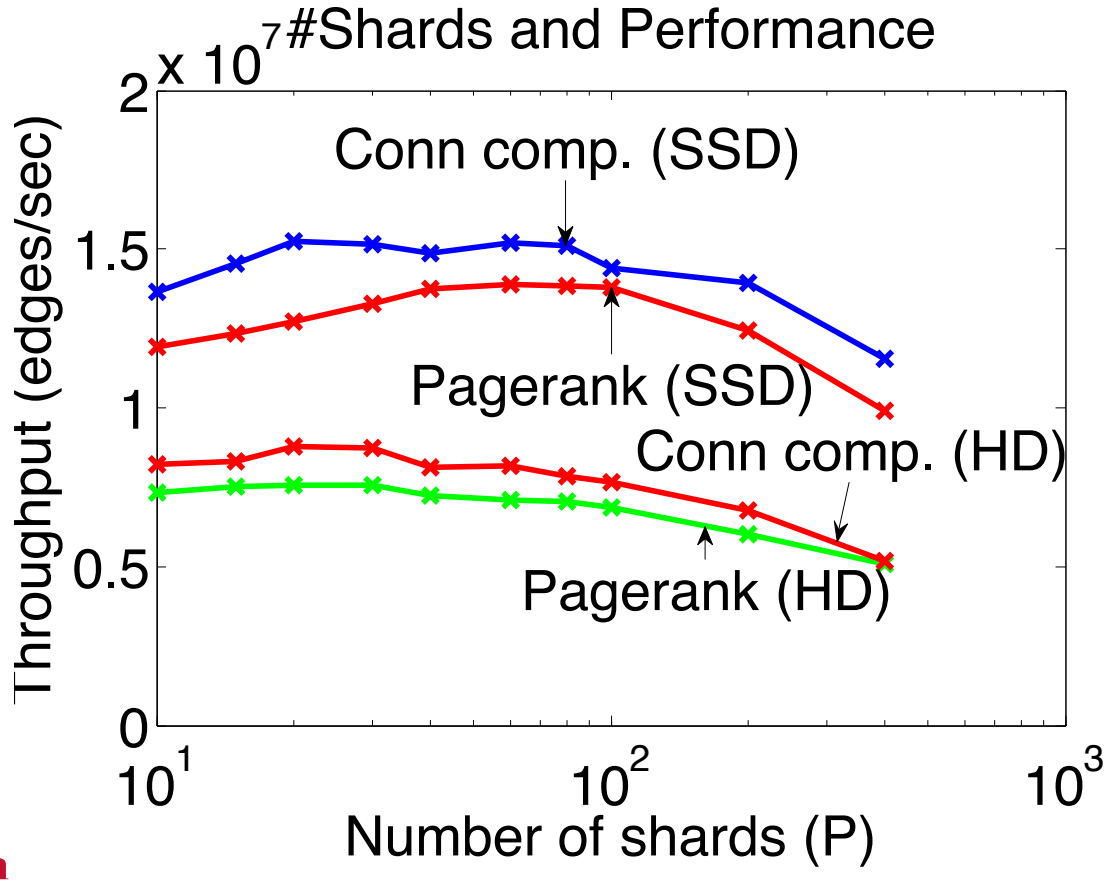
Expensive; The number of inter-cluster edges is big.

4. Caching of hot nodes?

Unpredictable performance.

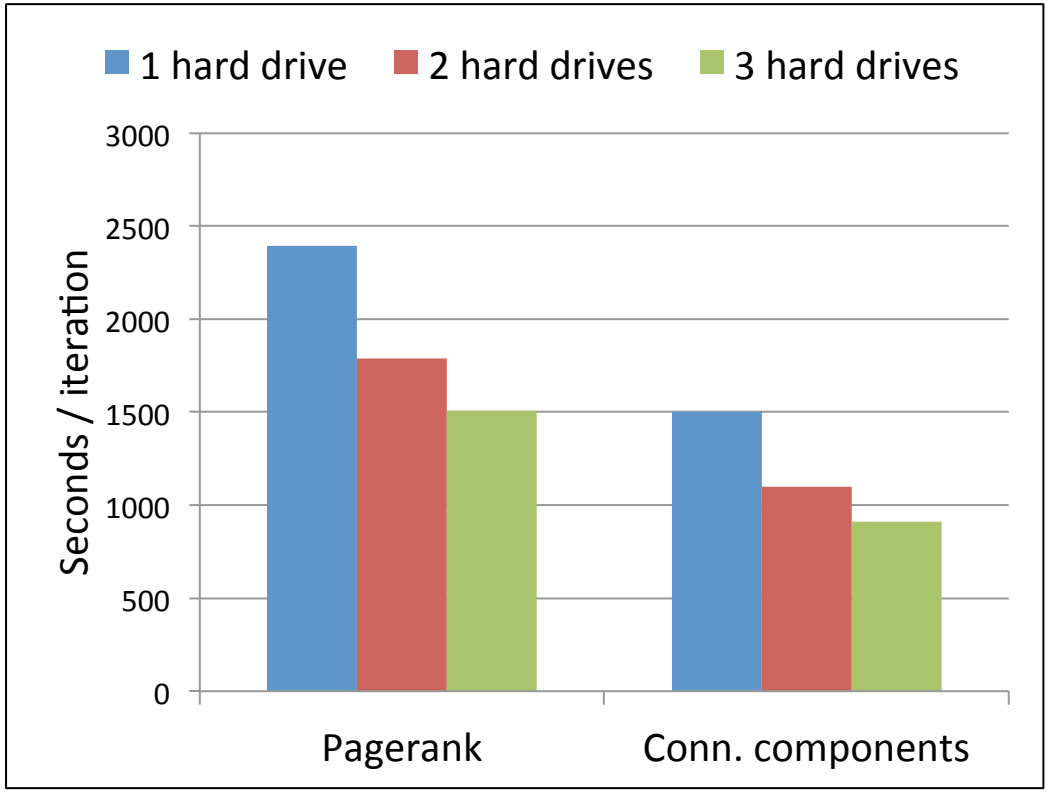
Number of Shards

- If P is in the "dozens", there is not much effect on performance.



Multiple hard-drives (RAIDish)

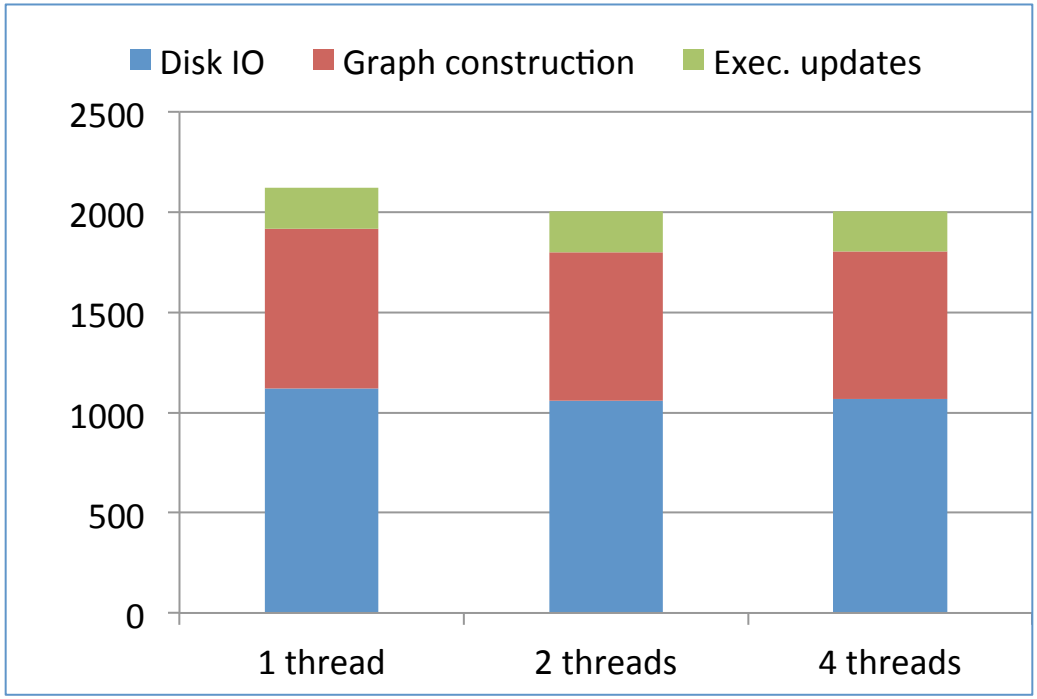
- GraphChi supports *striping* shards to multiple disks → Parallel I/O.



Experiment on a 16-core AMD server (from year 2007).

Bottlenecks

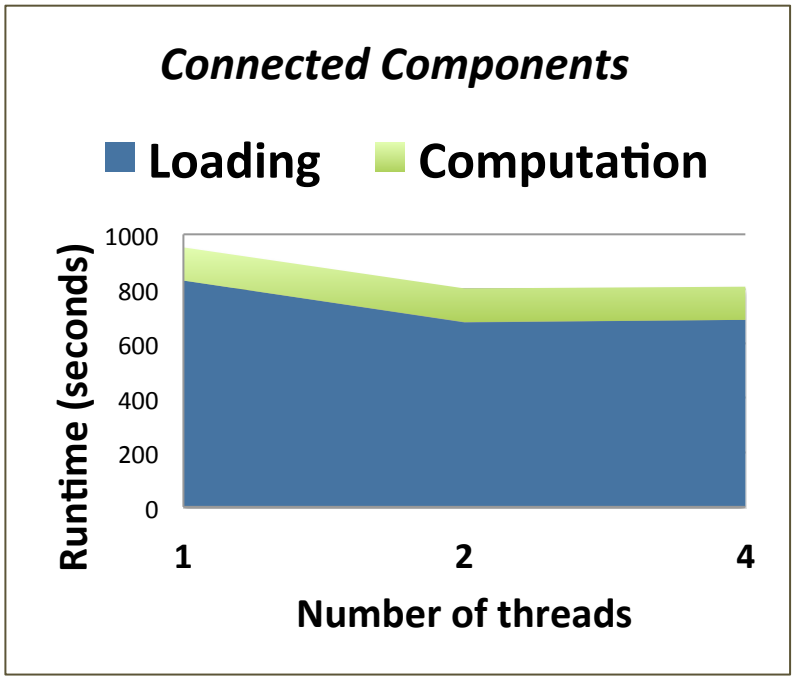
- Cost of constructing the sub-graph in memory is almost as large as the I/O cost on an SSD
  - Graph construction requires a lot of random access in RAM → memory bandwidth becomes a bottleneck.

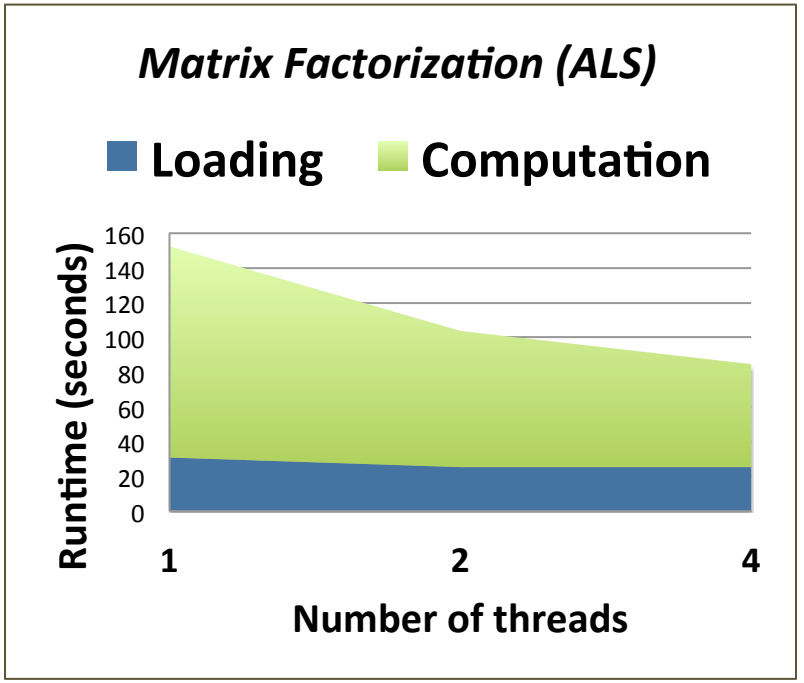


Connected Components on Mac Mini / SSD

Bottlenecks / Multicore

- Computationally intensive applications benefit substantially from parallel execution.
- GraphChi saturates SSD I/O with 2 threads.





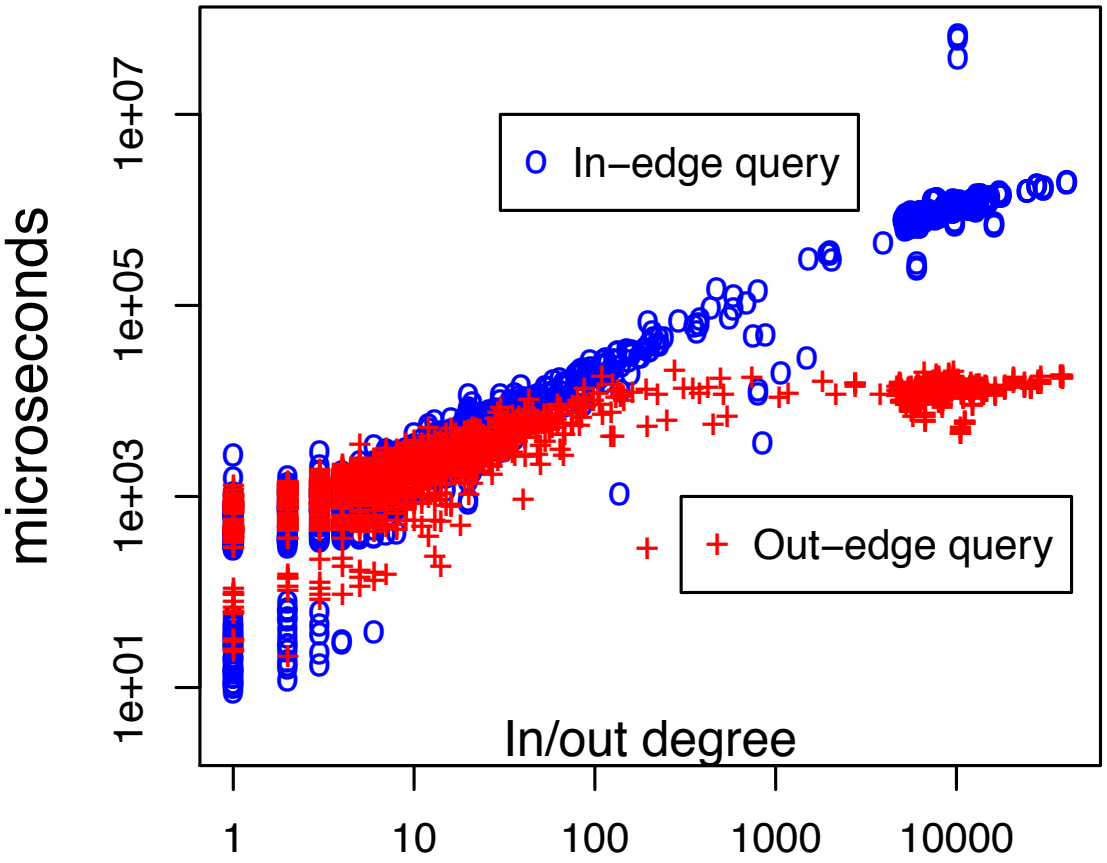
Experiment on MacBook Pro with 4 cores / SSD.

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

Table 3: Relative performance of an in-memory version of GraphChi compared to the default SSD-based implementation on a selected set of applications, on a Mac Mini. Timings include the time to load the input from disk and write the output into a file.

Experiment: Query latency



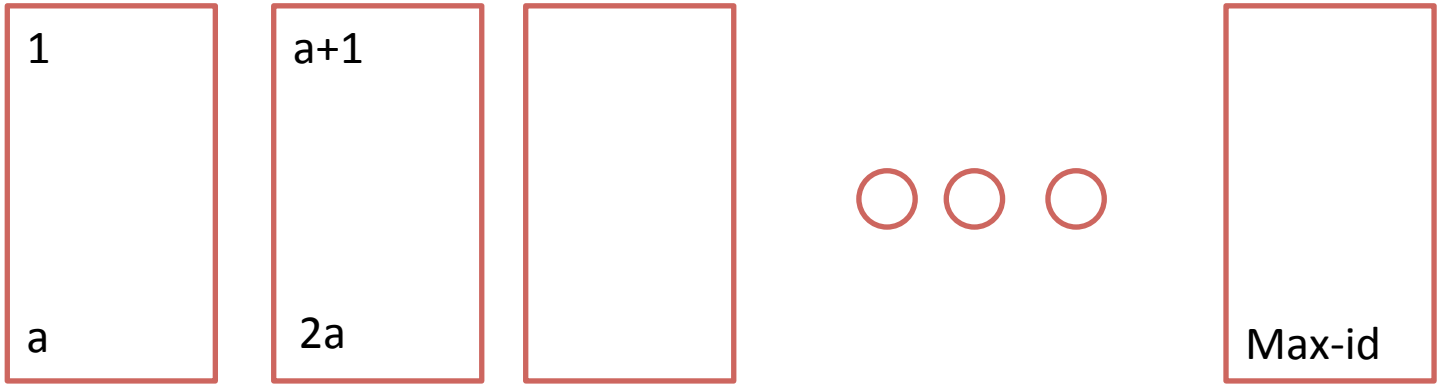
See thesis for I/O cost analysis of in/out queries.

Example: Induced Subgraph Queries

- **Induced subgraph** for vertex set S contains all edges in the graph that have both endpoints in S.
- Very fast in GraphChi-DB:
  - Sufficient to query for out-edges
  - Parallelizes well → multi-out-edge-query
- Can be used for **statistical graph analysis**
  - Sample induced neighborhoods, induced FoF neighborhoods from graph

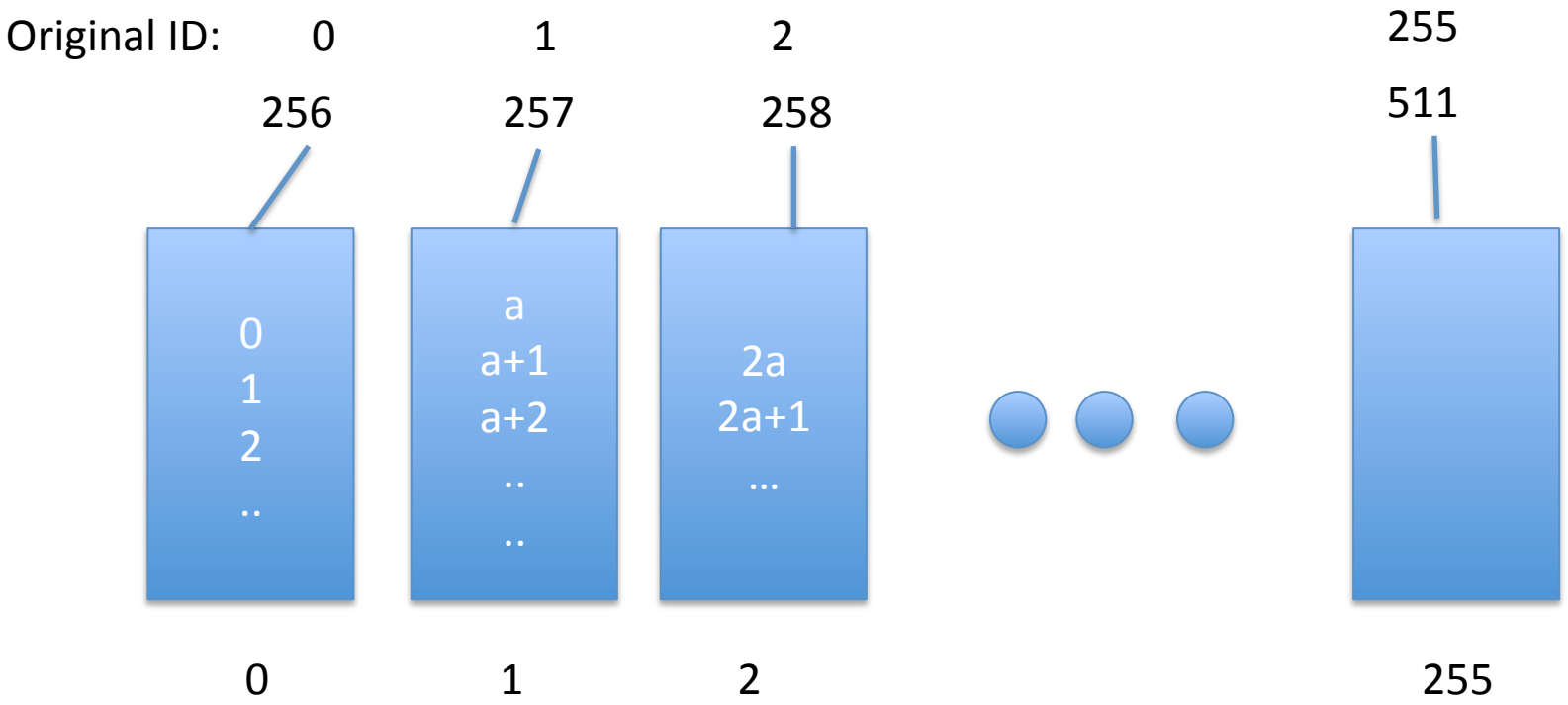
Vertices / Nodes

- Vertices are partitioned similarly as edges
  - Similar "data shards" for columns
- Lookup/update of vertex data is O(1)
- No merge tree here: Vertex files are "dense"
  - Sparse structure could be supported

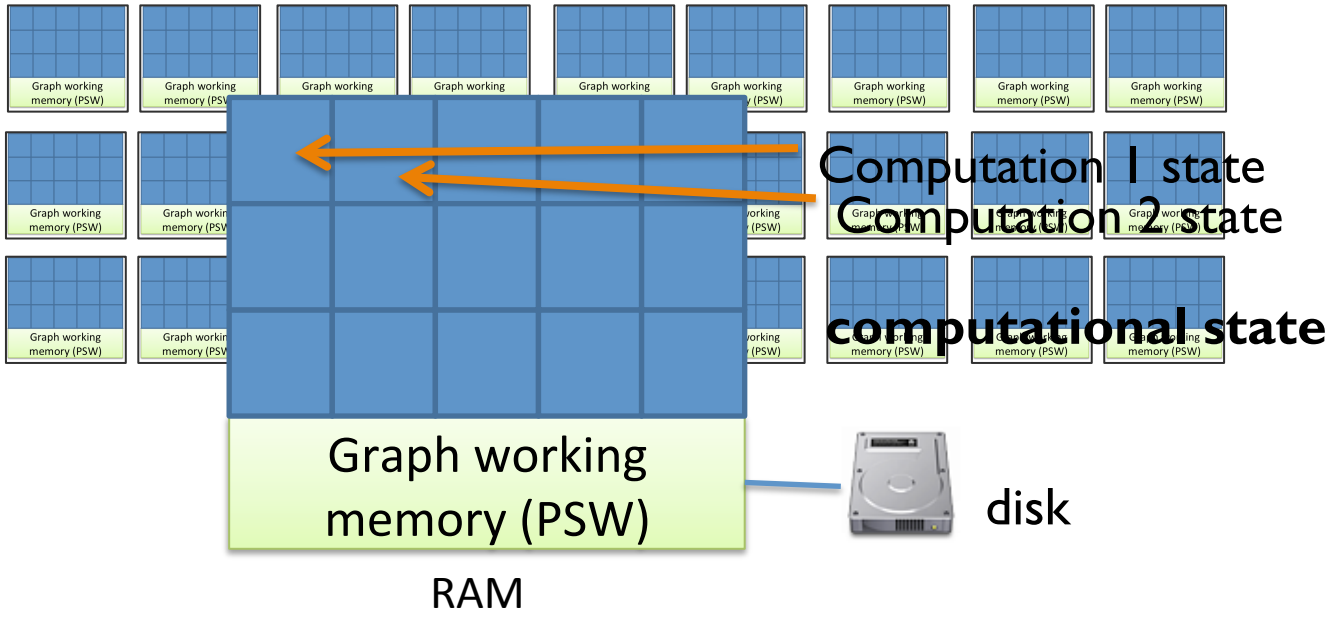


ID-mapping

- Vertex IDs mapped to internal IDs to balance shards:
  - Interval length constant **a**



What if we have a Cluster?



Trade latency for throughput!

Graph Computation: Research Challenges

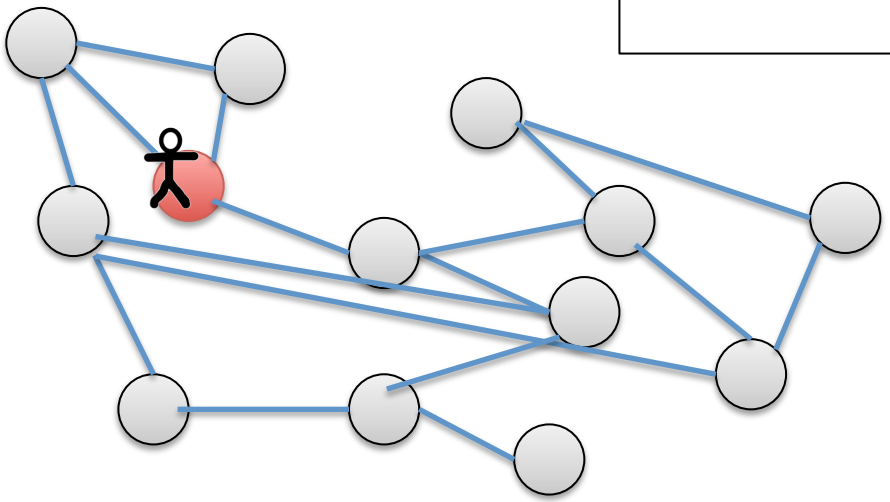
1. Lack of truly challenging (benchmark) applications
2. … which is caused by lack of good data available for the academics: big graphs with metadata
   - Industry co-operation → But problem with reproducibility
   - Also: it is hard to ask good questions about graphs (especially with just structure)
3. Too much focus on performance → More important to enable "extracting value"

Random walk in an in-memory graph

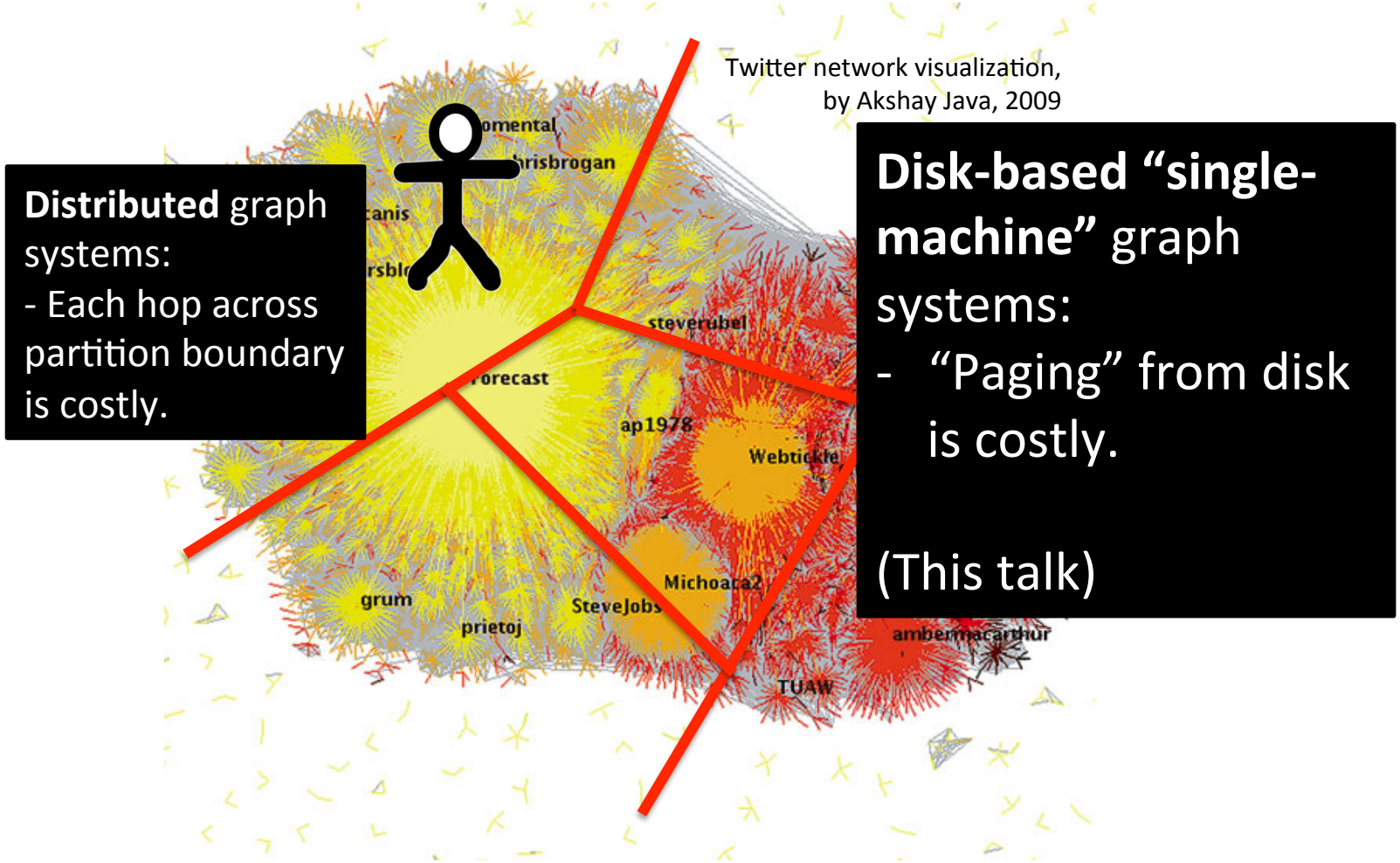
- Compute one walk a time (multiple in parallel, of course):

```
parfor walk in walks:
   for i=1 to numsteps:
       vertex = walk.atVertex()

walk.takeStep(vertex.randomNeighbor())
```



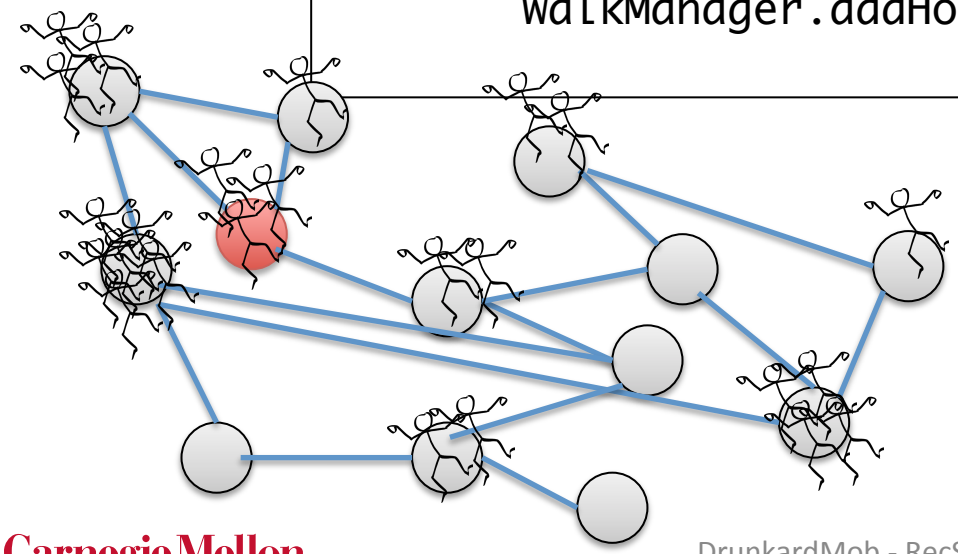
Problem: What if Graph does not fit in memory?



Random walks in GraphChi

- **DrunkardMob** –algorithm
  - Reverse thinking

```
ForEach interval p:
   walkSnapshot = getWalksForInterval(p)
   ForEach vertex in interval(p):
     mywalks = walkSnapshot.getWalksAtVertex(vertex.id)
     ForEach walk in mywalks:
        walkManager.addHop(walk, vertex.randomNeighbor())
```



Note: Need to store only current position of each walk!