WONDERLAND
A NOVEL ABSTRACTION-BASED OUT-OF-CORE GRAPH PROCESSING SYSTEM

Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, Kang Chen

Presented by Divya Gopinath

6.886 Spring 2019

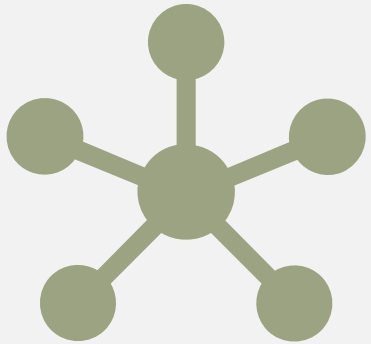
AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- System implementation
- Case study & benchmarking
- Conclusion
- Discussion

AGENDA

- **Motivation**
- **Background: out-of-core processing & graph abstraction**
- System implementation
- Case study & benchmarking
- Conclusion
- Discussion

MOTIVATION



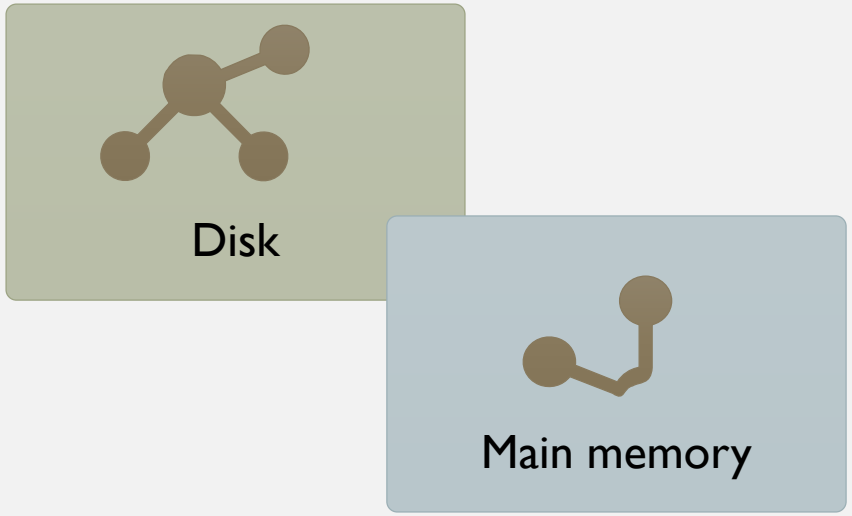
Graphs are everywhere, and require iterative algorithms to process input graph.

- How can we design an algorithm that isn't fully in-memory or distributed? ***Out-of-core***
- How can we reduce graph size? ***Abstraction***

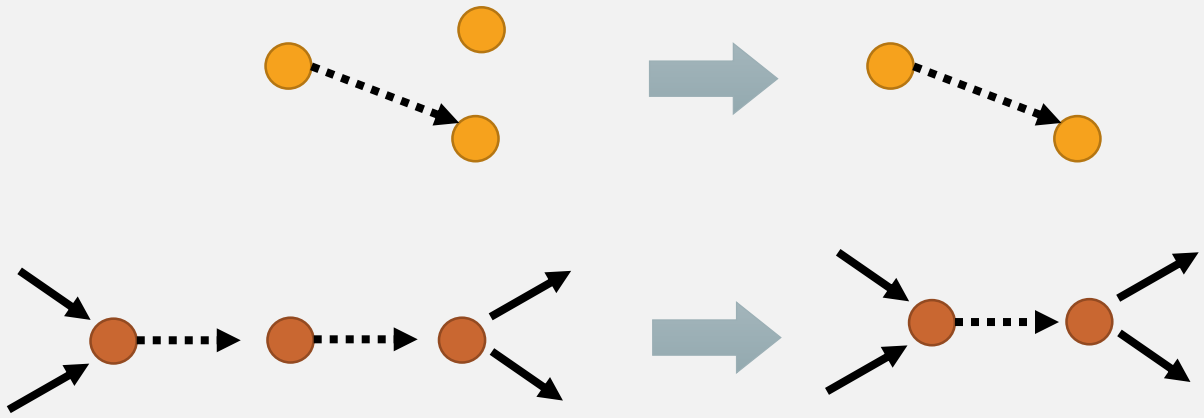
BACKGROUND

OUT-OF-CORE PROCESSING

- Equivalent to **disk-based single-machine** systems (external memory model)
  - Small portion of graph in main memory
  - Spill remainder to disk
- Shift from reducing slow random disk I/O to improving convergence speed



BACKGROUND



GRAPH ABSTRACTION

- Improve **execution efficiency** of algorithm
- Concise lossy representation of original graph
  - Bootstrap initial result to accelerate convergence
  - Fit abstraction in memory

BACKGROUND

How do we apply the **out-of-core model** to a **graph processing** framework that uses **abstraction** in some way?

BACKGROUND

How do we apply the **out-of-core model** to a **graph processing** framework that uses **abstraction** in some way?

Challenge 1: Performance

Most existing techniques assume original graph can be contained in memory– how do we adapt this for out-of-core systems? Transformations can be complex of need multiple passes through input data.

BACKGROUND

How do we apply the **out-of-core model** to a **graph processing** framework that uses **abstraction** in some way?

Challenge I: Performance

Most existing techniques assume original graph can be contained in memory– how do we adapt this for out-of-core systems? Transformations can be complex of need multiple passes through input data.

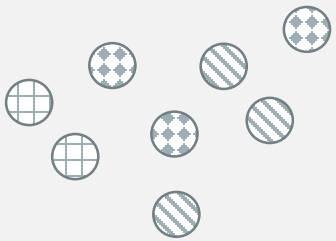
Challenge II: Programmability

Adding vertices/edges not in the original graph to create the abstraction makes programming harder– may require application-specific algorithms to recover the initial result.

BACKGROUND

OUT-OF-CORE GRAPH PROCESSING

GraphChi Vertex-centric



X-Stream Edge-centric

`Shuffle()` – external sort to provide updates

`Scatter()` – read all edges and output updates as stream

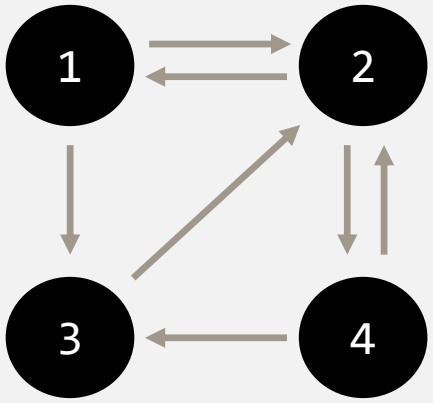
`Gather()` – apply updates

BACKGROUND

OUT-OF-CORE GRAPH PROCESSING

Grid graph Partition based

- Partition edges into `P x P` grids
- Each partition contains vertices within contiguous range
- Define edge function that updates `dst` from `src`



P = 2	Vertex 1 Vertex 2	Vertex 3 Vertex 4
Vertex 1 Vertex 2	(1, 2) (2, 1)	(1, 3) (2, 4)
Vertex 3 Vertex 4	(3, 2) (4, 2)	(4, 3)

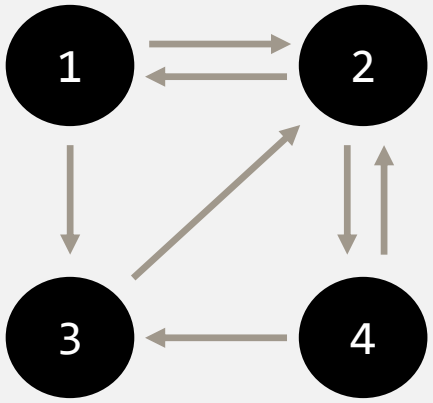
BACKGROUND

OUT-OF-CORE GRAPH PROCESSING

Grid graph Partition based

Common patterns:
Disjoint partitions
Fixed processing order

- Partition edges into `P` x `P` grids
- Each partition contains vertices within contiguous range
- Define edge function that updates `dst` from `src`



P = 2	Vertex 1 Vertex 2	Vertex 3 Vertex 4
Vertex 1 Vertex 2	(1, 2) (2, 1)	(1, 3) (2, 4)
Vertex 3 Vertex 4	(3, 2) (4, 2)	(4, 3)

BACKGROUND

OUT-OF-CORE GRAPH PROCESSING WITH ABSTRACTION

Challenge I: Performance

Challenge II: Programmability

- Create abstractions that consist **only of vertices and edges existing in original graph** (select subset of edge set)
- Enforces **application-independent** abstractions
- Possible to generate an abstraction with a **single-pass read** of original graph

AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- **System implementation**
- Case study & benchmarking
- Conclusion
- Discussion

WONDERLAND: OVERVIEW

Data model same as Gridgraph:

Mutable vertex property

Read-only edges

Two-phase workflow:

Abstraction generation

Query processing

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10       in_memory_graph = {abstract}
11       while not converge
12           Process(query, in_memory_graph)
13           load = Choose(on-disk)
14           in_memory_graph = {abstract, load}
```

Initialize empty abstraction

Expand abstraction with edges from graph

Select set of *X* edges to remain in abstraction

Edges not selected dumped/streamed to disk

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10       in_memory_graph = {abstract}
11       while not converge
12           Process(query, in_memory_graph)
13           load = Choose(on-disk)
14           in_memory_graph = {abstract, load}
```

Initialize empty abstraction →

Expand abstraction with edges from graph →

Select set of *X* edges to remain in abstraction →

Edges not selected dumped/streamed to disk →

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10       in_memory_graph = {abstract}
11       while not converge
12           Process(query, in_memory_graph)
13           load = Choose(on-disk)
14           in_memory_graph = {abstract, load}
```

Suitable for out-of-core systems!

Only expanded abstract loaded into memory in each iteration; limited edges selected

Original graph read only once

Generated abstraction only has edges from original graph

Initialize empty abstraction

Expand abstraction with edges from graph

Select set of *X* edges to remain in abstraction

Edges not selected dumped/streamed to disk

Compute on entire graph, or a round

Only store abstracted graph in memory

Compute on a subgraph, or an iteration

Part of on-disk graph chosen to load into memory

New in-memory subgraph created

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10       in_memory_graph = {abstract}
11       while not converge
12           Process(query, in_memory_graph)
13           load = Choose(on-disk)
14           in_memory_graph = {abstract, load}
```

GridGraph – information propagation is limited per round (each edge processed once), so slower convergence

Wonderland – Abstraction is a "bridge", and common abstractions are shared among iterations

Compute on entire graph, or a round

Only store abstracted graph in memory

Compute on a subgraph, or an iteration

Part of on-disk graph chosen to load into memory

New in-memory subgraph created

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10      in_memory_graph = {abstract}
11      while not converge
12          Process(query, in_memory_graph)
13          load = Choose(on-disk)
14          in_memory_graph = {abstract, load}
```

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10       in_memory_graph = {abstract}
11       while not converge
12           Process(query, in_memory_graph)
13           load = Choose(on-disk)
14           in_memory_graph = {abstract, load}
```

1

How do we select edges for our abstraction that will be suitable for a variety of applications?

2

How do we reduce the number of random disk accesses?

3

How do we implement the choose function that decides which part of the on-disk graph we load next?

4

What is a user-friendly, intuitive way of programming the process function?

```
1   // Abstraction Generation
2   abstraction = on-disk = {}
3   while not graph.empty()
4       abstraction = {abstract, graph.PopN(...)}
5       abstract, deleted = Select(abstract, X)
6       on-disk = {on-disk, deleted}
7
8   // Query Processing
9   foreach query
10      in_memory_graph = {abstract}
11      while not converge
12          Process(query, in_memory_graph)
13          load = Choose(on-disk)
14          in_memory_graph = {abstract, load}
```

1 SELECTING EDGES

What do users need to provide?

```
1  // Input
2  X = size of abstraction
3  B = the number of edges loaded per iteration
4  S = maximum size of each edge grid
5  W = width of grid
```

In practice, setting `B/X = 0.25` is a reasonable choice.
Picking X a bit harder– we'll come back to this.

1 SELECTING EDGES

What do users need to provide?

What level of abstraction do we give to the user?

```
1  // Input
2  X = size of abstraction
3  B = the number of edges loaded per iteration
4  S = maximum size of each edge grid
5  W = width of grid
```

In practice, setting `B/X = 0.25` is a reasonable choice. Picking X a bit harder– we'll come back to this.

Define low-level and high-level APIs for selecting edges:

Low-level API

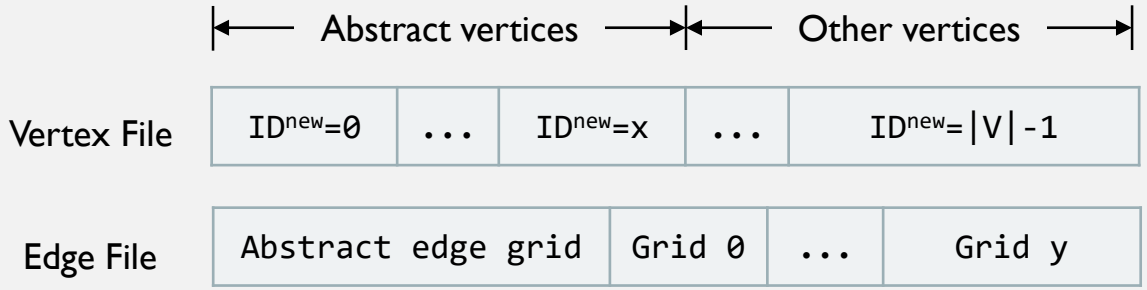
- `Select(vector<Edge>& abstract, size_t X)`
- Arbitrary method to sort edges

High-level API

- Edge-priority selection: `EdgePriority(Edge e)`
- Generate abstraction containing as few weakly connected components as possible with disjoint-set DS– first, join disconnected components, then add edges by priority

2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?



Reduce random edge accesses:

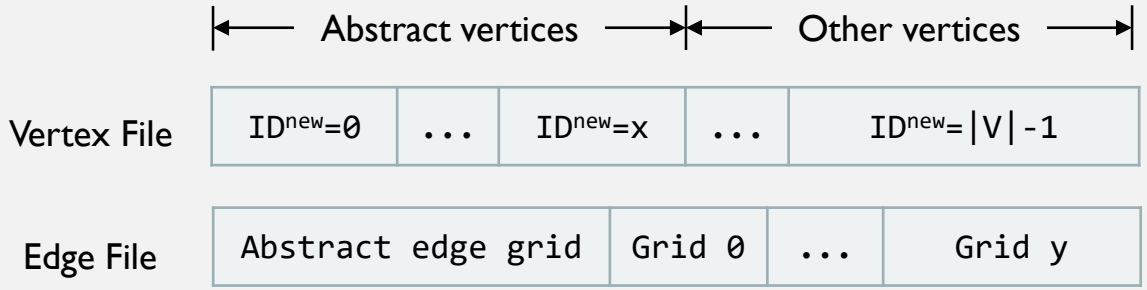
- Abstraction edges at header because used in tandem with all other grids
- Typically edges of abstraction are cached

Reduce random vertex accesses:

- Initially, we have a vertex file anyway!
- Possible that non-contiguous vertex IDs are accesses…

2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?

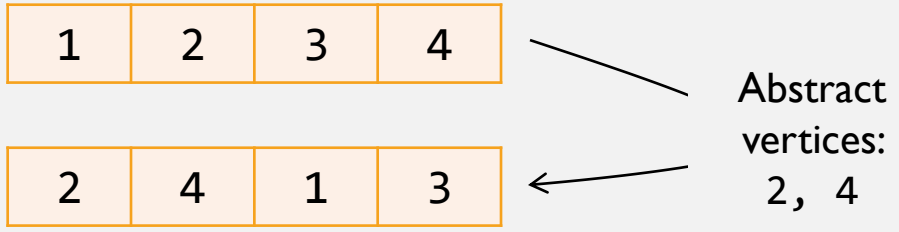


Reduce random edge accesses:

- Abstraction edges at header because used in tandem with all other grids
- Typically edges of abstraction are cached

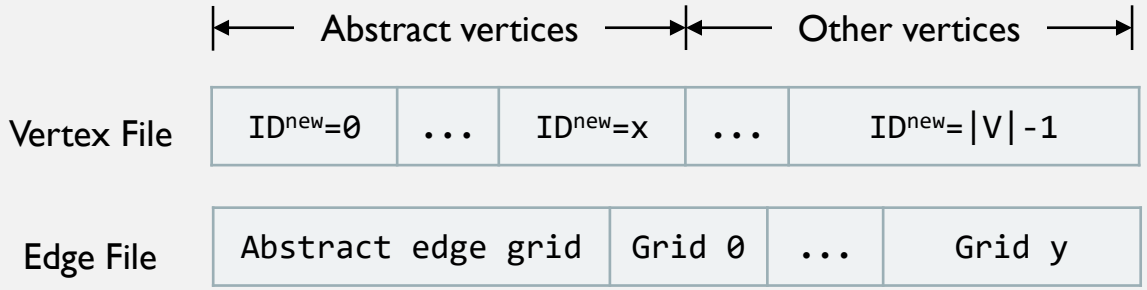
Reduce random vertex accesses:

- Initially, we have a vertex file anyway!
- Possible that non-contiguous vertex IDs are accesses…
- Remap so abstract vertices are at header (but order preserved as much as possible)

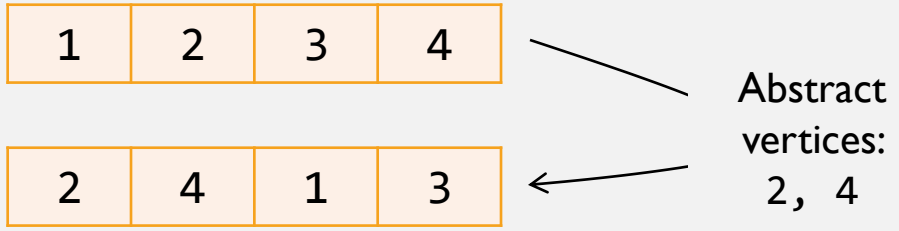


2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?



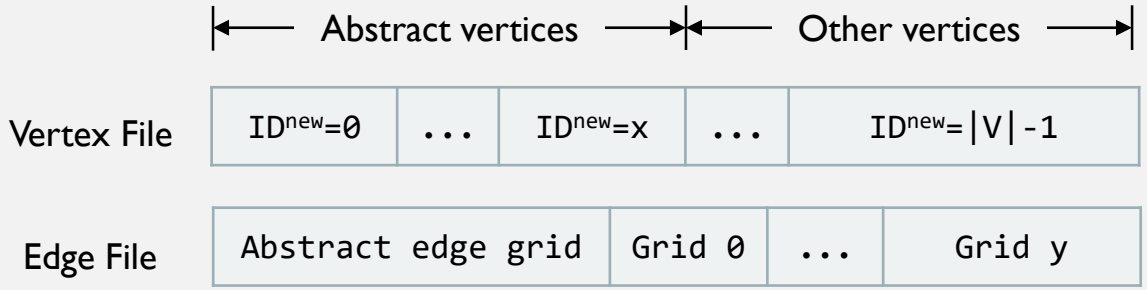
How is the remapping done in practice?



- Keep in-memory vector with abstract vertices
- Binary search on this. For a given vertex X, we know: 1) is it in the abstraction and 2) the number of vertices Y remaining in the abstract with smaller ID than X
  - If it remained, remap to Y
  - If it was removed, remap to X – Y + (number of vertices that remained in abstract)

2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?

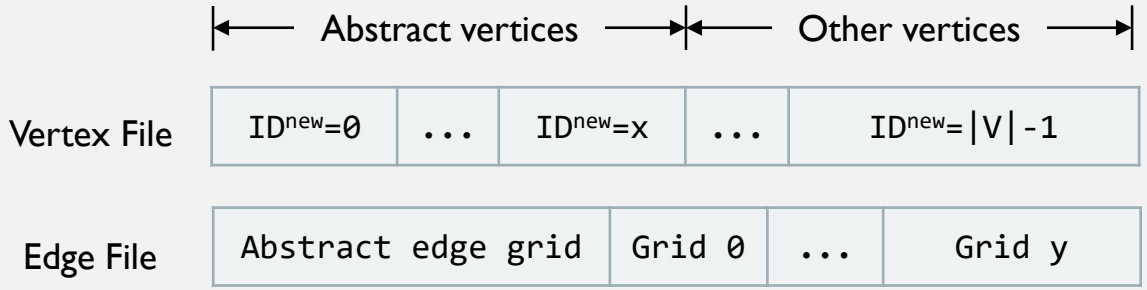


How are edges stored?

- If a grid contains X edges and is above the memory limit, partitioned into $\lceil X/S \rceil$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole

2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?



How are edges stored?

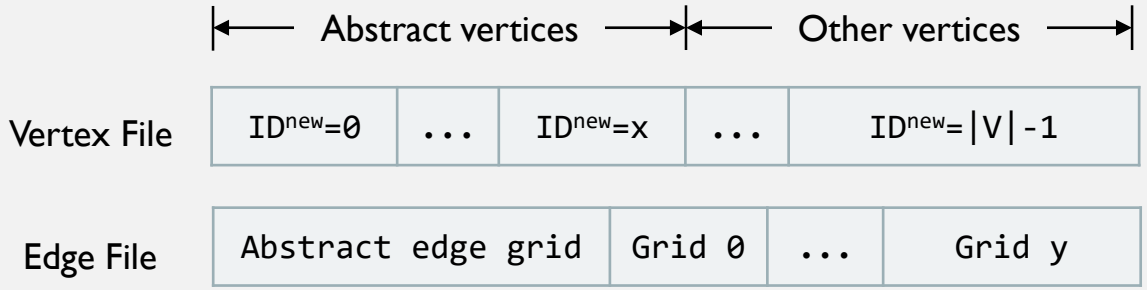
- If a grid contains X edges and is above the memory limit, partitioned into $\lceil X/S \rceil$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole

How many passes through the edge list is required for grid partitioning?

Counting edges per grid (1)
Write edges to locations (1)
Total: 2 passes

2 REDUCING DISK ACCESSES

How do we store the graph in memory, noting that we process in grids?



How are edges stored?

- If a grid contains X edges and is above the memory limit, partitioned into $\lceil X/S \rceil$ edges (either randomly or by priority)
- Edges contiguously stored in CSR format (locating outgoing edges of a vertex in constant time)
- Grids are always loaded as a whole

How many passes through the edge list is required for grid partitioning?

Counting edges per grid (1)
Write edges to locations (1)
Total: 2 passes

Combine first pass with remapping procedure.
Preprocessing total: 3 passes of edges (selecting abstraction, remapping, partitioning)

3 CHOOSING EDGES FROM DISK

How do we select what edges we load from disk to get our in-memory graph?

Problem: this is very application-specific, and we wanted to develop an **application-agnostic** framework.

First, enforce users can only choose **grid partitions.**

Have users specify the order in which they evaluate grids to enforce a scheduling.

4 EXECUTION OF QUERY

How do we actually express an algorithm, or process, to run a query on the graph?

Low-level API — Do-it-yourself

- Define `Process(Graph& g)` function
- Given a vector of loaded grids, including abstraction

High-level API — "Think like a vertex"

- Define function to process on a per-vertex basis
- Provide multi-threaded executor to parallelize
- Reuse execution engine already present in literature

4 EXECUTION OF QUERY

How do we actually express an algorithm, or process, to run a query on the graph?

Low-level API

```
Process(Graph& g)
```

Given a list of grids (including abstraction, which is a special grid), user has access to:

- List of edges in the grid
- Smallest and largest ID of source vertices
- Access to endpoints for each vertex

4 EXECUTION OF QUERY

How do we actually express an algorithm, or process, to run a query on the graph?

Low-level API

```
Process(Graph& g)
```

Given a list of grids (including abstraction, which is a special grid), user has access to:

- List of edges in the grid
- Smallest and largest ID of source vertices
- Access to endpoints for each vertex

High-level API

```
func VertexProgram(Graph& g, Index u)
  // Iterating loaded edges
  foreach grid in g.loaded_grids
    foreach edge in [grid.StartEdge(u),
                     grid.EndEdge(u)]
      ProcessEdge(g.Vertex(u), edge,
                  g.Vertex(edge.destination))

  // Update priority of grids
  foreach grid in g.all_grids
    if u >= grid.EndVertex(): continue
    if u < grid.StartVertex(): continue
    UpdatePriority(g.Vertex(u), g.Priority(grid))
```

FINAL IMPLEMENTATION SKETCH

```
1  // Input
2  X = size of abstraction
3  B = the number of edges loaded per iteration
4  S = maximum size of each edge grid
5  W = width of grid
6
7  // Generating Abstract
8  abstract = vector<Edge>()
9  on-disk = fstream(...)
10 while not graph.empty()
11   abstract = {abstract, graph.PopN(B)}
12   abstract, deleted = Select(abstract, X)
13
14 // Remapping and Partitioning
15 abstract, grids = Partition(abstract, on-disk)
16
17 // Processing Querys
18 foreach query
19   in_memory_graph = {abstract}
20   worklist = BootstrapWorklist(in_memory_graph, query)
21   while not converge
22     // Worklist-based processing
23     while not worklist.empty()
24       u = worklist.pop()
25       for e in in_memory_graph.loaded_edges(u)
26         ProcessEdge(u, e)
27         Append worklist accordingly
28       Update priority of grids accordingly
29     // Bootstrap next iteration
30     {grid1, grid2, ...} = Choose(grids, B)
31     in_memory_graph = {abstract, grid1, grid2, ...}
32     worklist = BootstrapWorklist(in_memory_graph, query)
```

AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- System implementation
- **Case study & benchmarking**
- Conclusion
- Discussion

CASE STUDY: SHORTEST-PATH

- Given a weighted graph with a source (`src`) and a destination (`dst`).
- Negative edge weights allowed, but no negative weight cycles.
- Attach a dist property to each vertex and run Dijkstra's.
- Relaxation: `dist[u] + w[u, v] < dist[v]`

First, we define the necessary steps for an edge relaxation…

Perform relaxation if necessary →
"Activate" edge destination →
Push to worklist or process all loaded vertices once →

```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
7        float new_dist = src_dist + edge.weight
8        float& dst_dist = g.Vertex(edge.desination).dist
9        if new_dist < dst_dist
10         dst_dist = new_dist
11         g.Active(edge.destination)
12         Worklist.push(edge.destination)
13
14   // Update priority of grids
15   foreach i in [0, g.all_grids.size())
16     Grid& grid = g.all_grids[i]
17     if u >= grid.EndVertex(): continue
18     if u < grid.StartVertex(): continue
19     float new_priority = -(src_dist + min_expect[i, dst])
20     if grid.priority < new_priority
21       grid.priority = new_priority
```

Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
7        float new_dist = src_dist + edge.weight
8        float& dst_dist = g.Vertex(edge.desination).dist
9        if new_dist < dst_dist
10         dst_dist = new_dist
11         g.Active(edge.destination)
12         Worklist.push(edge.destination)
13
14   // Update priority of grids
15   foreach i in [0, g.all_grids.size())
16     Grid& grid = g.all_grids[i]
17     if u >= grid.EndVertex(): continue
18     if u < grid.StartVertex(): continue
19     float new_priority = -(src_dist + min_expect[i, dst])
20     if grid.priority < new_priority
21       grid.priority = new_priority
```

Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

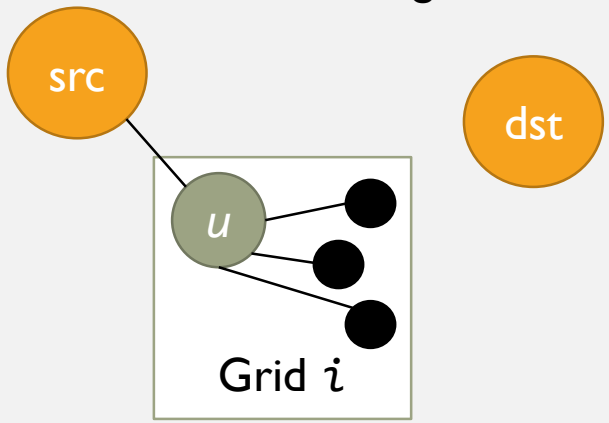
Define `min_expect[i, dst]` for each grid and each `dst` to be the precalculated value of the lower bound of a path starting from an edge in each grid to `dst`.

```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
7        float new_dist = src_dist + edge.weight
8        float& dst_dist = g.Vertex(edge.desination).dist
9        if new_dist < dst_dist
10         dst_dist = new_dist
11         g.Active(edge.destination)
12         Worklist.push(edge.destination)
13
14   // Update priority of grids
15   foreach i in [0, g.all_grids.size())
16     Grid& grid = g.all_grids[i]
17     if u >= grid.EndVertex(): continue
18     if u < grid.StartVertex(): continue
19     float new_priority = -(src_dist + min_expect[i, dst])
20     if grid.priority < new_priority
21       grid.priority = new_priority
```

Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

Define `min_expect[i, dst]` for each grid and each `dst` to be the precalculated value of the lower bound of a path starting from an edge in each grid to `dst`.




```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
7        float new_dist = src_dist + edge.weight
8        float& dst_dist = g.Vertex(edge.desination).dist
9        if new_dist < dst_dist
10         dst_dist = new_dist
11         g.Active(edge.destination)
12         Worklist.push(edge.destination)
13
14   // Update priority of grids
15   foreach i in [0, g.all_grids.size())
16     Grid& grid = g.all_grids[i]
17     if u >= grid.EndVertex(): continue
18     if u < grid.StartVertex(): continue
19     float new_priority = -(src_dist + min_expect[i, dst])
20     if grid.priority < new_priority
21       grid.priority = new_priority
```

Current priority: p

But, if:

`Vertex(u).dist + min_expect[i, dst] < p`

Update priority with its negation

Then, we need to update the priority of the given grid.

Intuition: estimate lower-bound on distance. The smaller this is, the higher the priority.

Define `min_expect[i, dst]` for each grid and each `dst` to be the precalculated value of the lower bound of a path starting from an edge in each grid to `dst`.

Computing this could be hard, so instead, we let each grid keep one `min_expect`, which is the minimum edge weight in the grid.

```
1  func VertexProgram(Graph& g, Index u)
2    float src_dist = g.Vertex(u).dist
3
4    // Iterating loaded edges
5    foreach grid in g.loaded_grids
6      foreach edge in [grid.StartEdge(u), grid.EndEdge(u))
7        float new_dist = src_dist + edge.weight
8        float& dst_dist = g.Vertex(edge.desination).dist
9        if new_dist < dst_dist
10         dst_dist = new_dist
11         g.Active(edge.destination)
12         Worklist.push(edge.destination)
13
14   // Update priority of grids
15   foreach i in [0, g.all_grids.size())
16     Grid& grid = g.all_grids[i]
17     if u >= grid.EndVertex(): continue
18     if u < grid.StartVertex(): continue
19     float new_priority = -(src_dist + min_expect[i, dst])
20     if grid.priority < new_priority
21       grid.priority = new_priority
```

SHORTEST-PATH OPTIMIZATIONS

Graph abstraction	Edge priority is negative of its weight
	Lightest X edges always in abstraction
Upper-bound	Processing a sequence of SP queries (not SSSP)
	Can use current dist priorty of dst vertex as upper-bound of relaxation
Selecting Loading	If all edges in grid are not activated, skip over

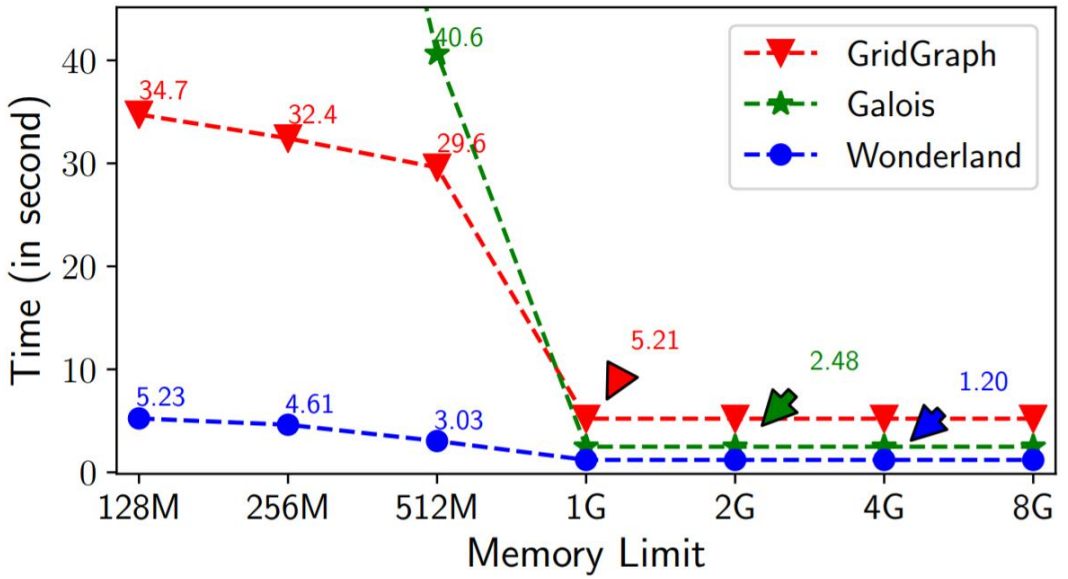
EVALUATION ENVIRONMENT

- Two 8-core Intel Xeon CPUs
- Set memory limits per query
- Compare against GridGraph and Galois, as well as out-of-core Galois (named LigraChi-g)
- Average results from 30 query pairs
- Reported results on LiveJournal (790 MB) and Twitter (17 GB) graphs

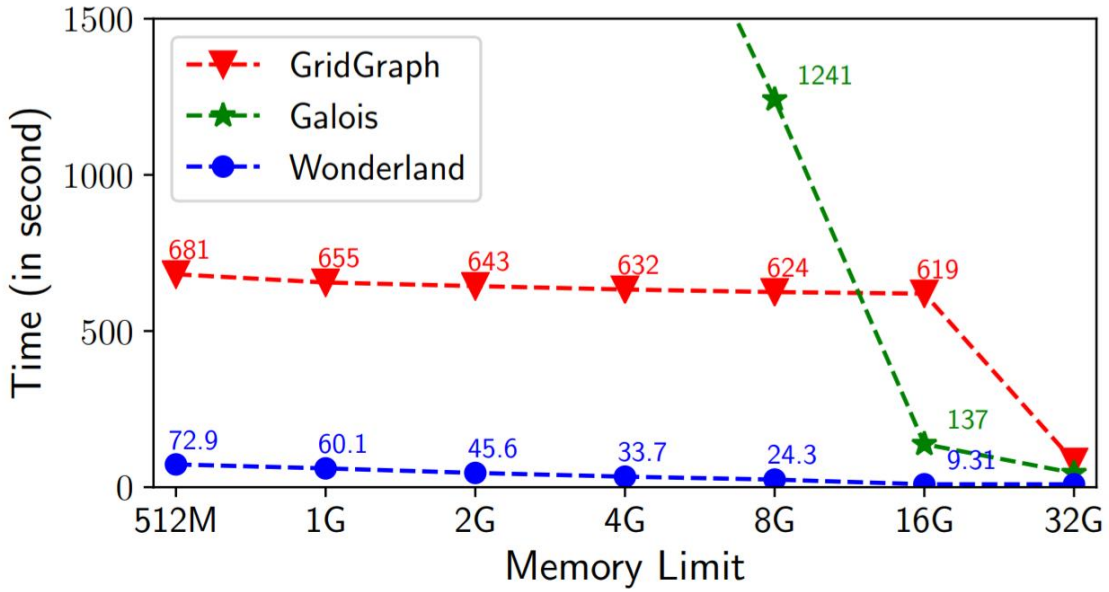
SHORTEST-PATH PERFORMANCE: BENCHMARKS

For high memory limits, devolves into fully in-memory setting, and Galois and Wonderland outperform GridGraph significantly.

For out-of-core system, Wonderland outperforms GridGraph because of abstraction (always keep certain things in memory).



(a) LiveJournal.



(b) Twitter.

SHORTEST PATH CASE STUDY
PIECEWISE BREAKDOWN

Three sources of speedup:

1. Bootstrapping an initial result
2. Abstraction-enabled information propagation
3. Abstraction-guided priority scheduling

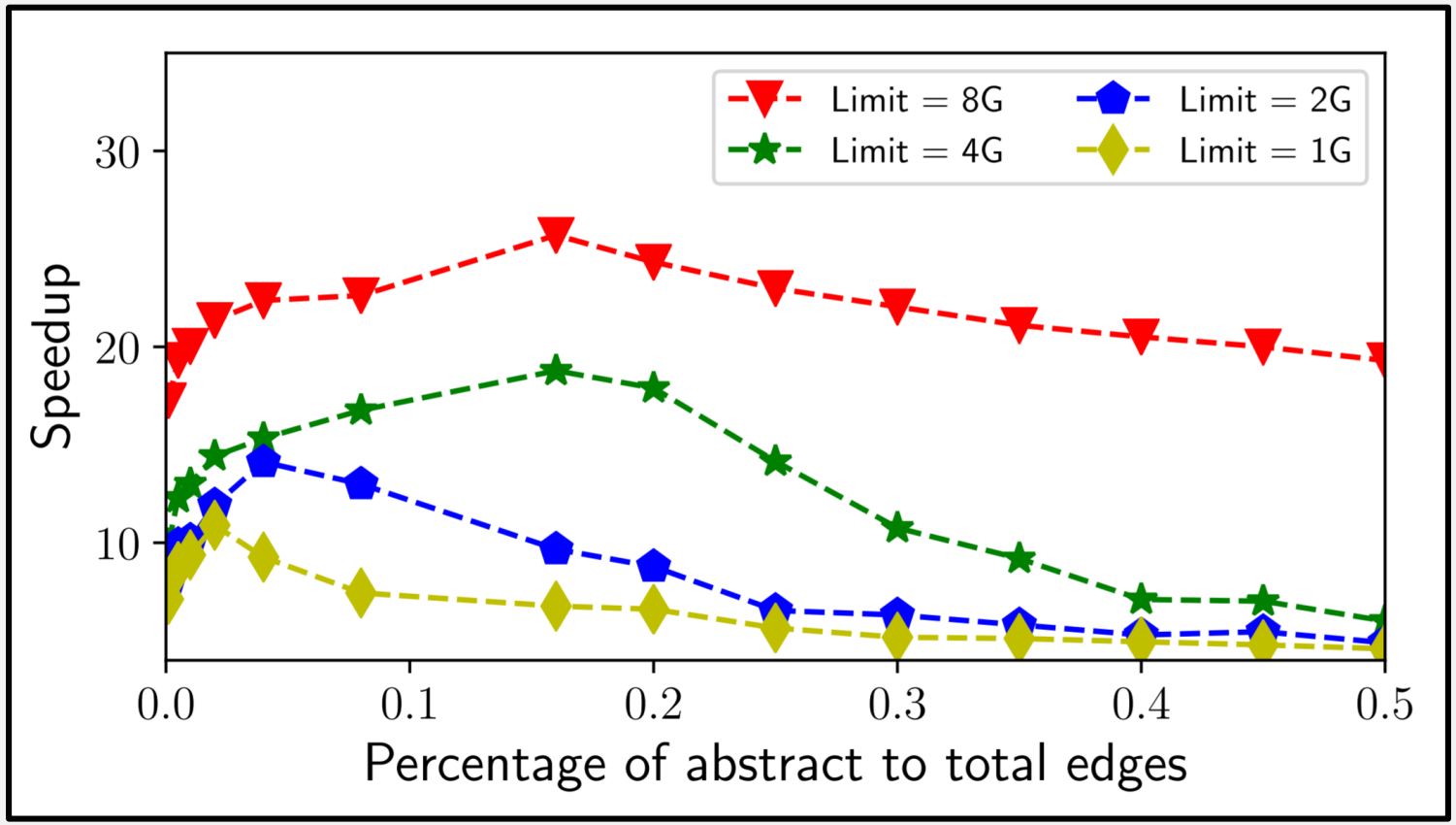
If only (1), speedups are limited as memory limit goes down

Adding (2), extra speedup across the board

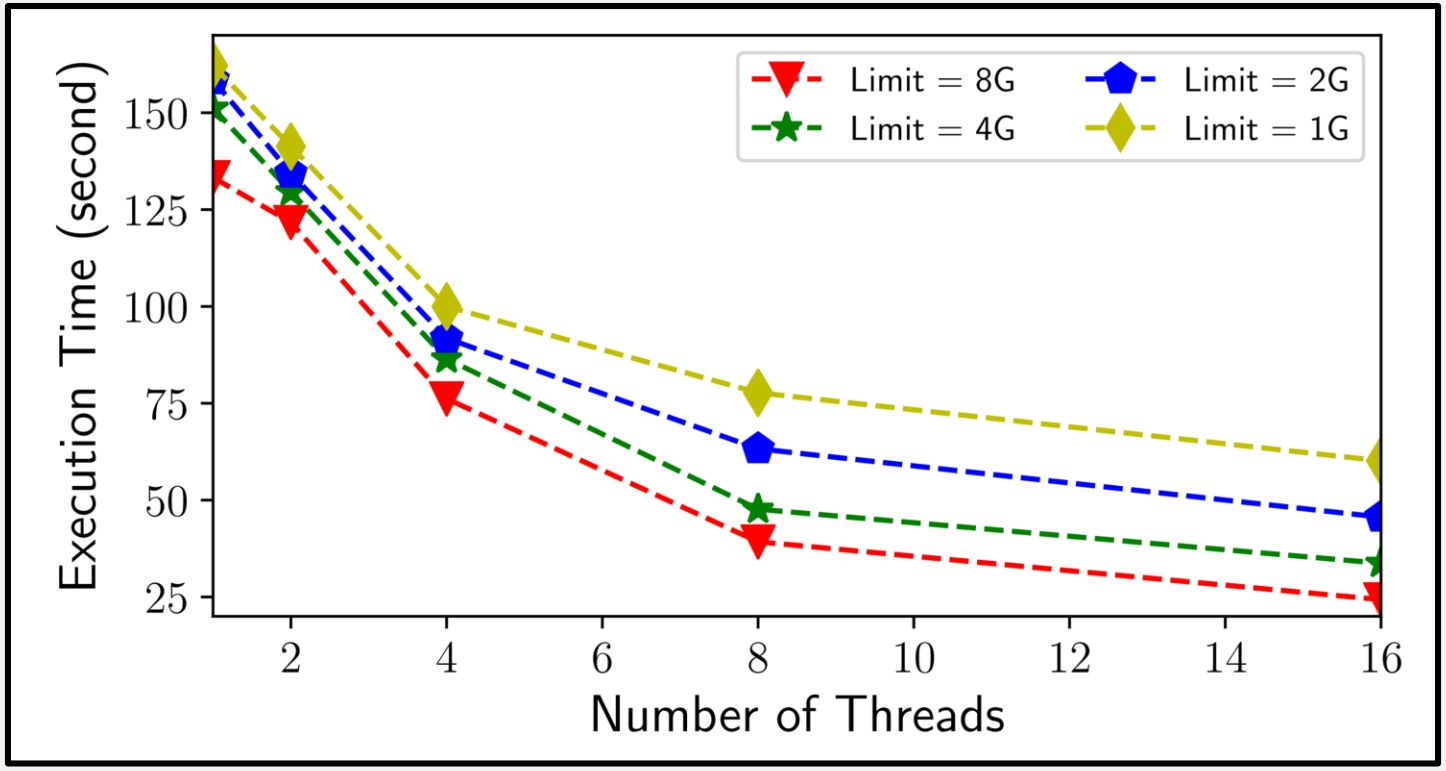
Adding (3), extra speedup across the board

But, (1) most important if memory limit is relatively high

SHORTEST PATH CASE STUDY
ABSTRACTION SIZE SENSITIVITY



SHORTEST PATH CASE STUDY
MULTI-THREAD SPEEDUP



SHORTEST PATH CASE STUDY
SELECTIVITY SENSITIVITY

We define the **selectivity** of the query to denote how much of the graph structure the query necessitates (# total paths)/(# paths we check):

- ***Any path queries***: reachability, weakly-connected components (WCC)
- ***All path queries:*** shortest path (can be pruned from an exhaustive search, so high-selectivity), widest path (maximize weight of min-weight edge)

Memory Limit		1/2	1/4	1/8
Reachbility	LiveJournal	19.8×	16.1×	13.5×
	Twitter	86.6×	67.0×	45.7×
	Frindster	376×	247×	201×
	Dimacs	55.3×	39.1×	10.7×
WCC	LiveJournal	4.85×	3.75×	1.82×
	Twitter	6.18×	5.90×	3.78×
	Frindster	7.11×	6.20×	4.89×
	Dimacs	9.03×	4.32×	1.49×
Shortest Path	LiveJournal	9.92×	8.05×	3.53×
	Twitter	24.7×	20.9×	15.4×
	Frindster	36.2×	24.2×	17.3×
	Dimacs	17.8×	5.91×	2.13×
Widest Path	LiveJournal	11.3×	6.18×	3.03×
	Twitter	9.68×	6.91×	6.04×
	Frindster	20.1×	17.5×	11.2×
	Dimacs	63.2×	41.3×	2.65×

PREPROCESSING TIME

Dataset	GridGraph	Wonderland		
		Random	Order	Connectivity
LiveJournal	5.53	7.09	9.15	18.6
Twitter	92.8	101	161	317
Frindster	235	295	398	912
Dimacs	3.79	6.29	6.35	11.6

Even for one query, Wonderland is faster than GridGraph.

SCOPE OF APPLICATION

- Higher speedup if problem has a higher selectivity (BFS, MST)
- Basic graph operations; could be extended to clustering, matching, flow, etc.
- Wonderland can also be faster in computing sparse matrix-vector multiplication algos like PageRank, but due to Galois engine, not abstraction

AGENDA

- Motivation
- Background: out-of-core processing & graph abstraction
- System implementation
- Case study & benchmarking
- **Conclusion**
- **Discussion**

CONCLUSION

- Wonderland: novel, out-of-core graph processing framework
  - Extract effective abstractions from original graph
  - Use this to enable effective information propagation
  - Use this to enable priority scheduling for faster convergence

- Drastic speedup over other state-of-the-art systems

DISCUSSION

- More information on tuning parameters during benchmarking
- Useful to see piecewise breakdown, but unclear how bootstrapping of initial result happens
- Theoretical work into deriving bounds on the best X
- Paper laid out well but should have done a more thorough review of literature– many of the concepts (besides abstraction) are not novel to this paper in particular