Graph Processing in NVRAM and Streaming Settings

Laxman Dhulipala
MIT (Postdoc)
https://ldhulipala.github.io/

Based on joint work with

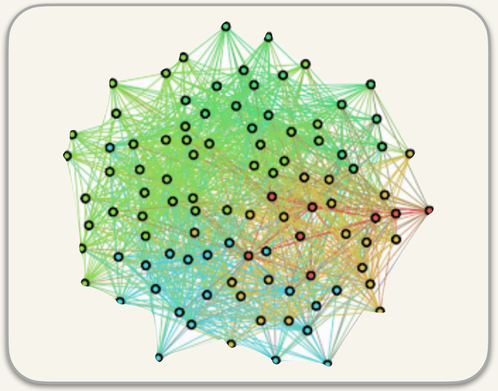
Guy Blelloch and Julian Shun (PLDI'19)

Charles McGuffey, Hong Kang, Yan Gu, Guy Blelloch, Phil Gibbons, and Julian Shun (VLDB'20)

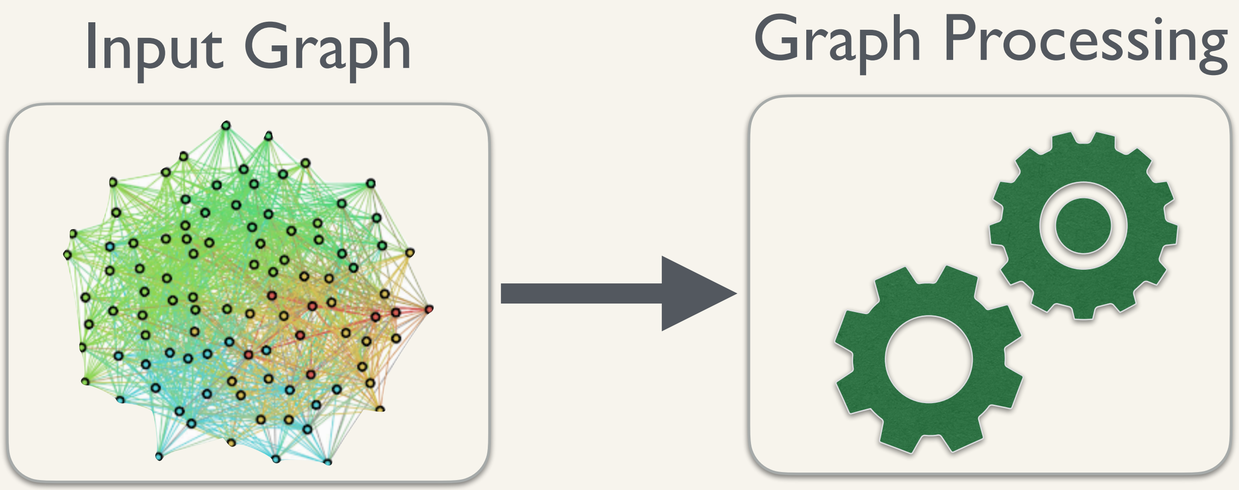
Graph Processing: algorithms and systems that enable us to analyze and understand graphs

Graph Processing: algorithms and systems that enable us to analyze and understand graphs

Input Graph



Graph Processing: algorithms and systems that enable us to analyze and understand graphs

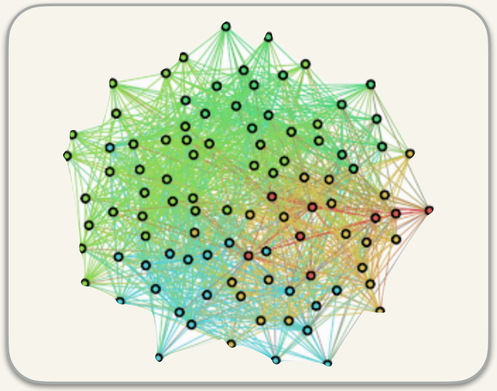


Algorithms

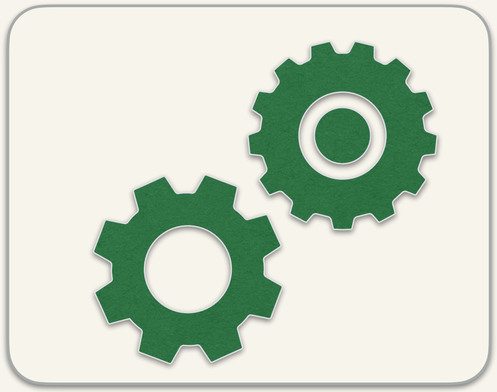
- Connectivity
- Graph Clustering
- ...
- Distance Computations
- Dense Subgraphs

Graph Processing: algorithms and systems that enable us to analyze and understand graphs

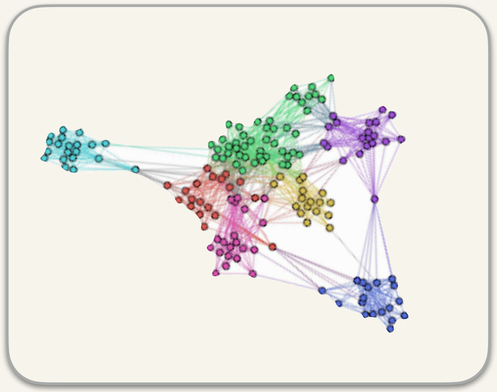
Input Graph



Graph Processing



Output



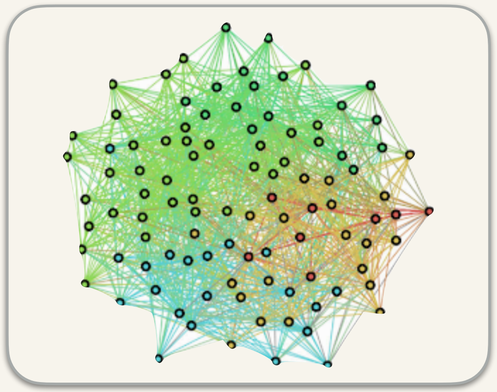
Algorithms

- Connectivity
- Graph Clustering
- ...
- Distance Computations
- Dense Subgraphs

- Understanding
- Visualizations
- Graph-based features
- System-optimization

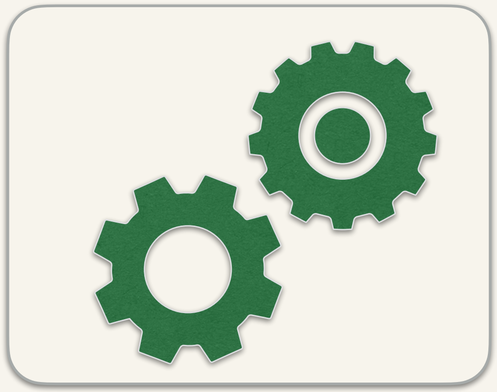
Graph Processing: algorithms and systems that enable us to analyze and understand graphs

Input Graph



- Static
- Dynamic

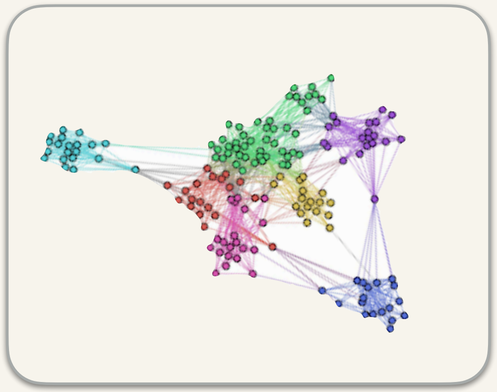
Graph Processing



Algorithms

- Connectivity
- Graph Clustering
- ...
- Distance Computations
- Dense Subgraphs

Output



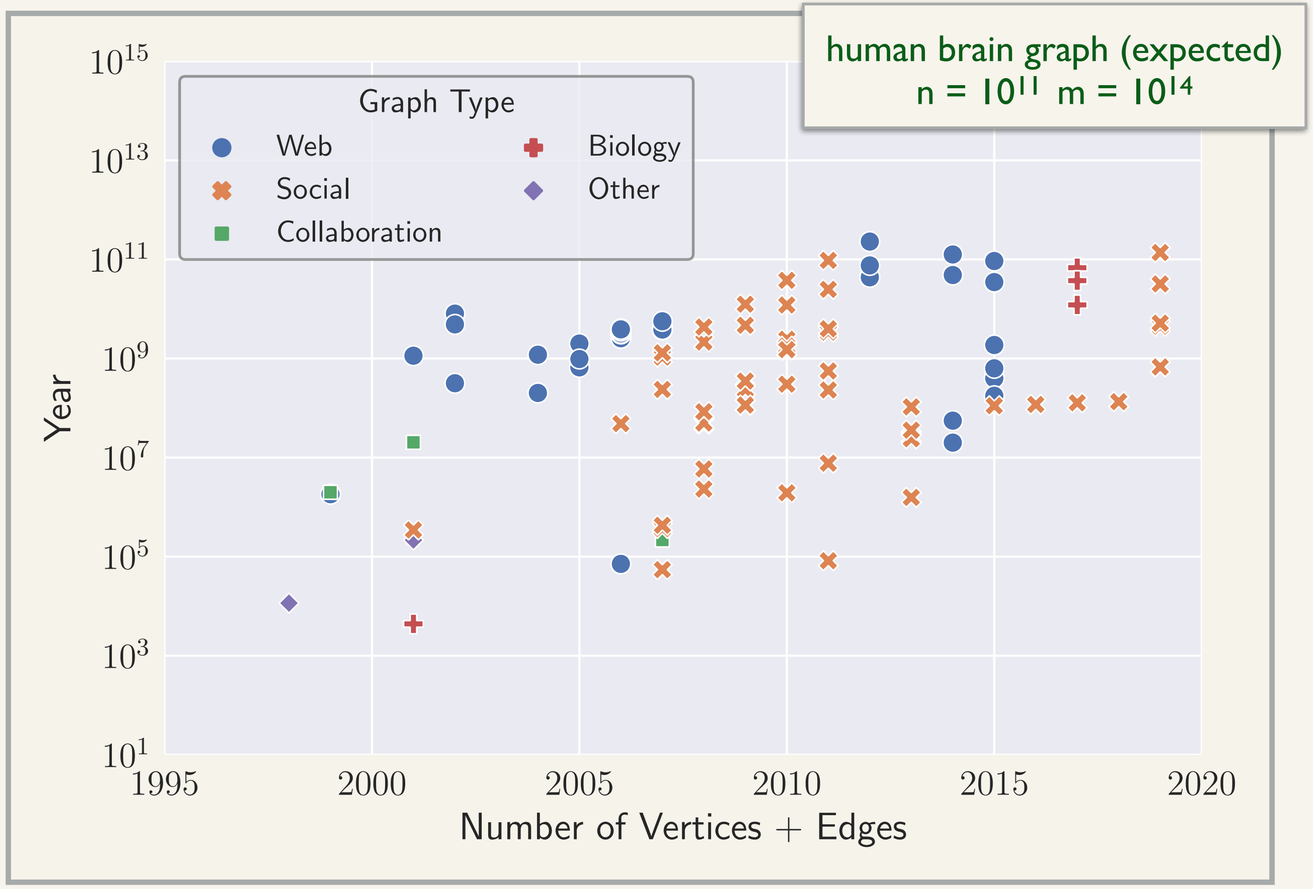
- Understanding
- Visualizations
- Graph-based features
- System-optimization

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

"…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft."



Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Sources: https://snap.stanford.edu/data/, http://law.di.unimi.it/datasets.php, http://webdatacommons.org/hyperlinkgraph/

Shared-Memory Parallelism

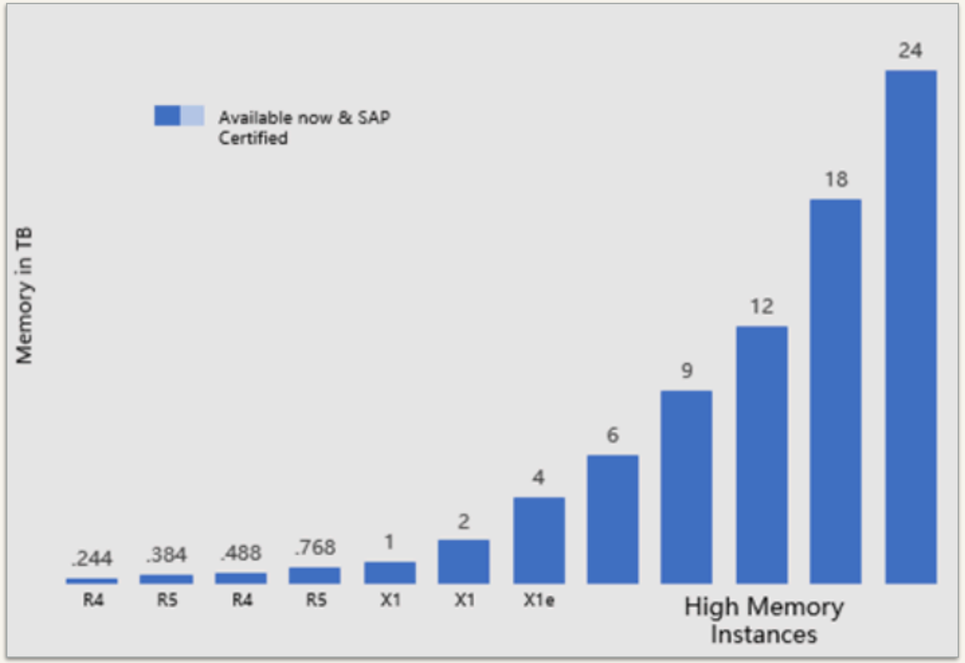
Shared-Memory Machines

- Cost for a 1TB memory machine with 72 processors is about $20,000.
- Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud

WebDataCommons Graph

- 3.5 billion vertices and 128 billion edges

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare



Shared-Memory Parallelism

Shared-Memory Machines

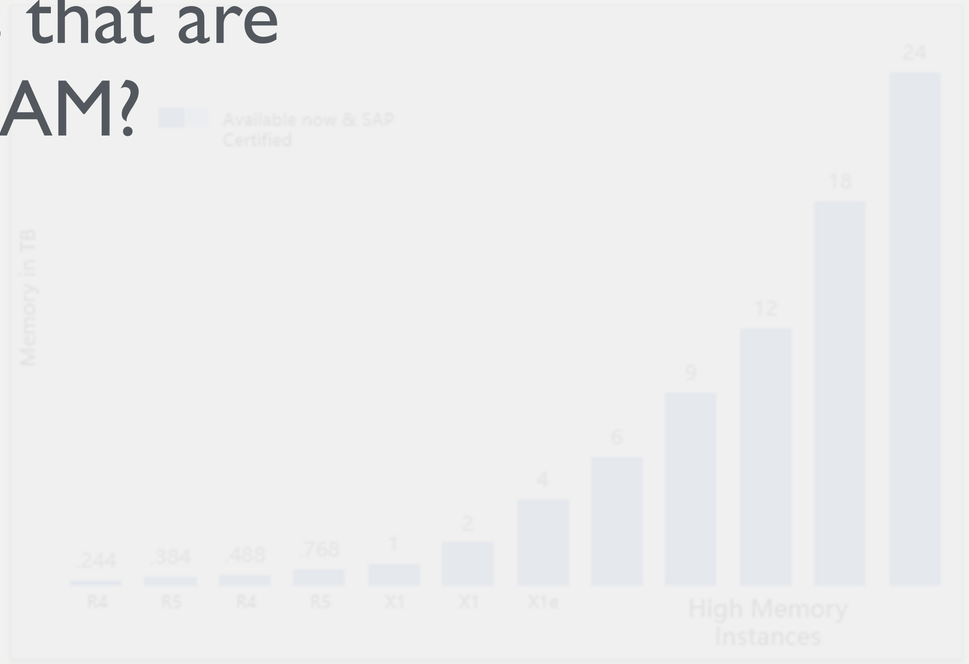
- Cost for a 1TB memory machine with 72 processors is about $20,000.
- Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud

WebDataCommons Graph

- 3.5 billion vertices and 128 billion edges

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare

What about graphs that are larger-than-DRAM?

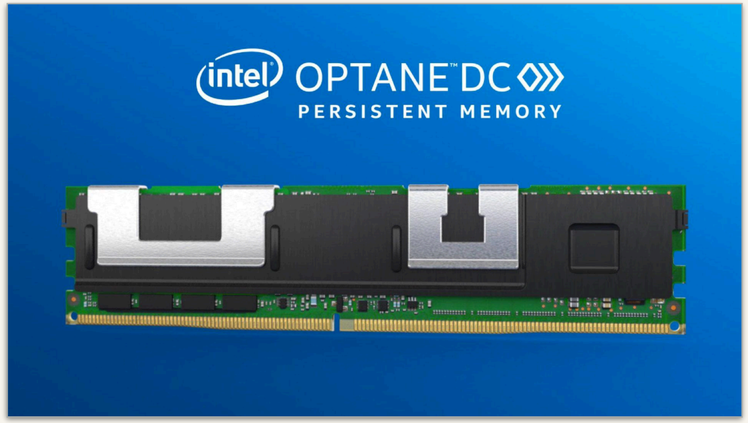


NVRAM Graph Processing

Non-Volatile Memory (NVRAM)

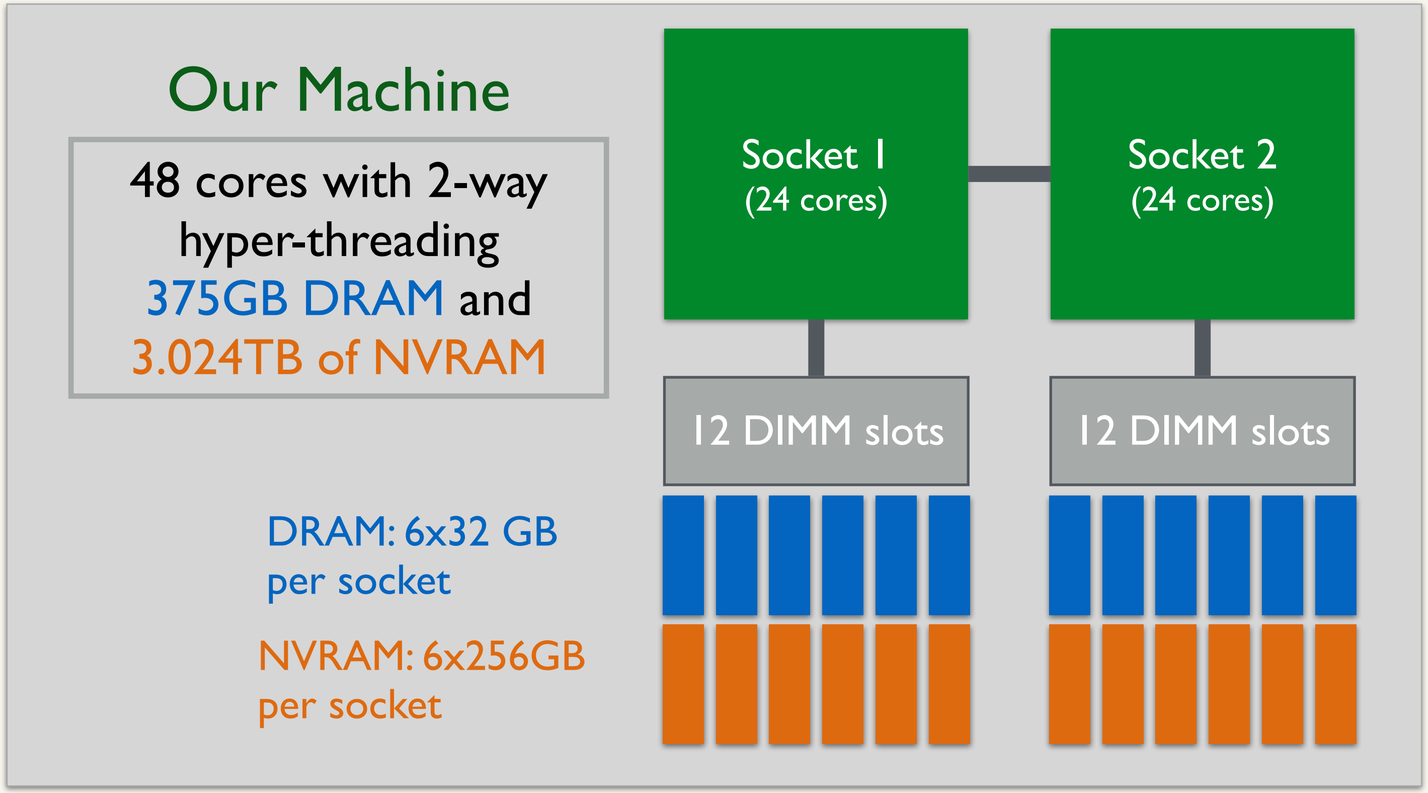
Intel Optane DC Memory

- Cheaper than DRAM on a per-byte basis
- Order of magnitude more capacity
- Memory is *persistent* and *byte-addressable*



Can we design algorithms that effectively use NVRAM as a higher-capacity memory while achieving DRAM-competitive performance?

NVRAM Characteristics

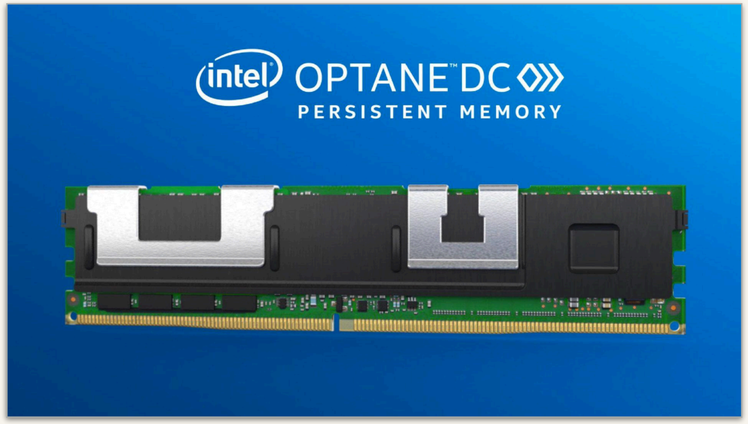


- 8x more NVRAM than DRAM
- NVRAM read throughput ~3x lower than DRAM read
- NVRAM write throughput further 4x lower

Recent work on Asymmetry

Benchmarking

- Two recent studies by Izraelevitz et al. [0] and van Renen et al. [1] perform careful benchmarking of Optane memory, and report similar asymmetries



Algorithms and Systems for Asymmetric Settings

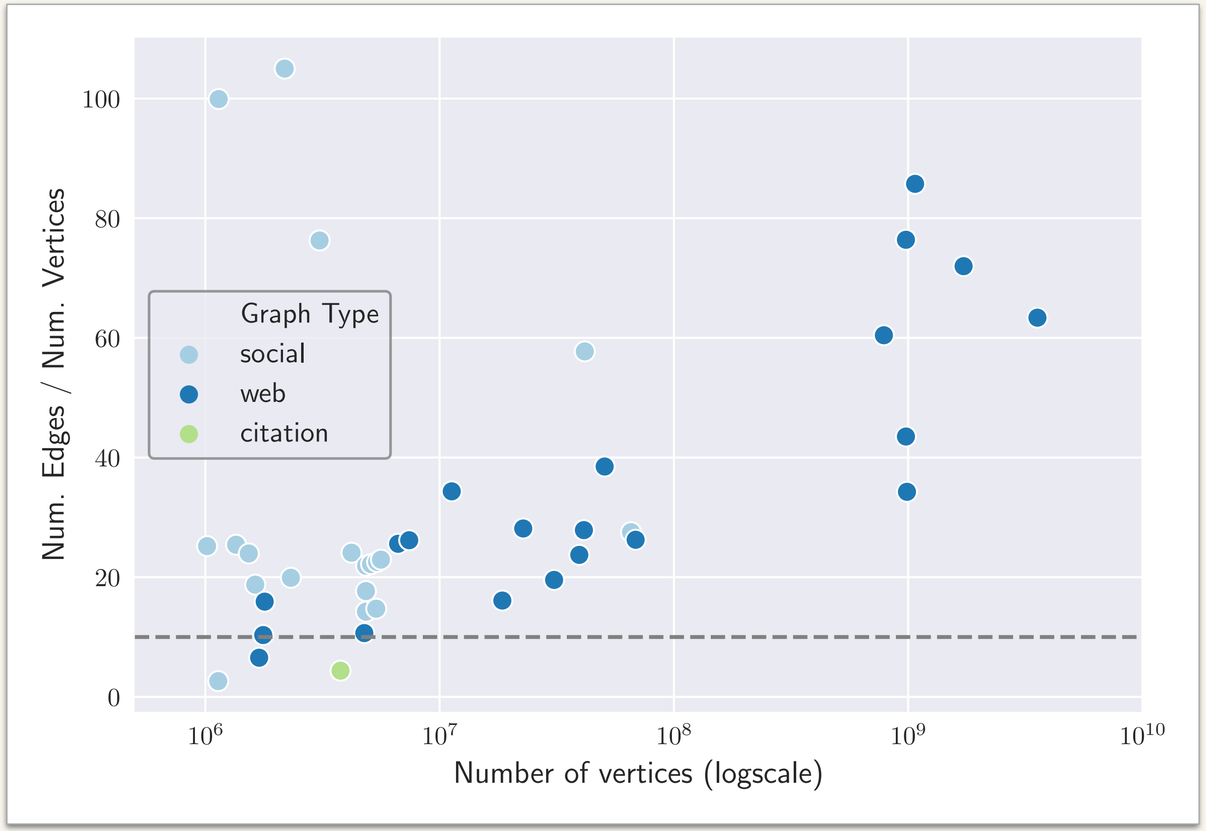
- Recent work explores how to minimize the number of NVRAM writes, e.g., [2 – 4], including many other papers
- Also significant work from systems, architecture, and database communities, e.g., [5 – 7], amongst many other papers

Sources:
[0] Izraelevitz et al. Basic performance measurements of the Intel Optane DC persistent memory module. (2019)
[1] van Renen et al. Persistent Memory I/O Primitives (2019)
[2] Ben-David et al. Parallel algorithms for asymmetric read-write costs (2016)
[3] Blelloch et al. Efficient algorithms with asymmetric read and write costs (2016)
[4] Carson et al. Write-avoiding algorithms (2016)
[5] Peng et al. System Evaluation of the Intel Optane byte-addressable NVM (2019)
[6] Ni et al. SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging (2019)
[7] Yang et al. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory (2020)

Semi-Asymmetric Parallel Graph Algorithms for NVRAMs [DMKGBGS'20]

Can we design practical and theoretically-sound techniques to overcome read/write asymmetry for graph problems on NVRAMs?

Real World Graphs are not Ultra-Sparse



Over 90% of graphs with > 1M vertices from SNAP and LAW datasets have $m/n \geq 10$

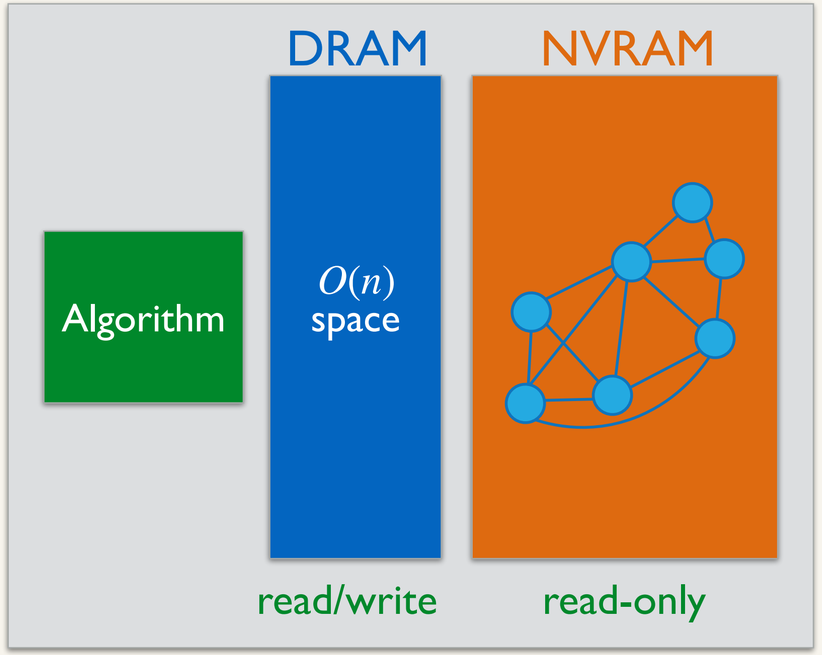
We expect that ratio of NVRAM/DRAM in future systems will be similar (our ratio is 8x)

Sources:
https://snap.stanford.edu/data/
http://law.di.unimi.it/datasets.php

Our Approach

Semi-Asymmetric Approach

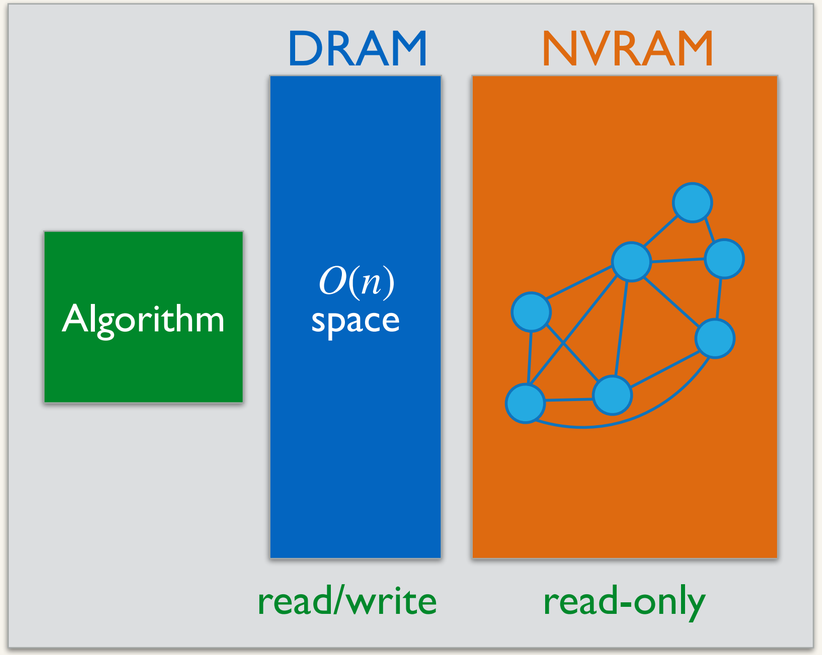
- Graph stored in NVRAM and accessed in a *read-only mode*
- Amount of DRAM is proportional to the number of vertices



Our Approach

Semi-Asymmetric Approach

- Graph stored in NVRAM and accessed in a *read-only mode*
- Amount of DRAM is proportional to the number of vertices



Benefits

- Algorithms avoid costly NVRAM writes, and algorithm design is independent of this cost
- Algorithms do not contribute to NVRAM wear-out

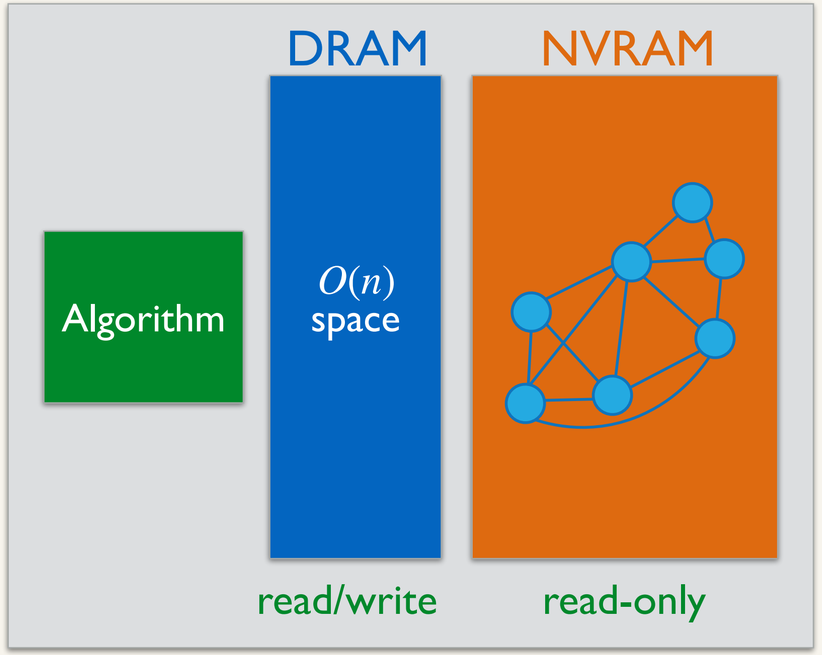
Our Approach

Semi-Asymmetric Approach

- Graph stored in NVRAM and accessed in a *read-only mode*
- Amount of DRAM is proportional to the number of vertices

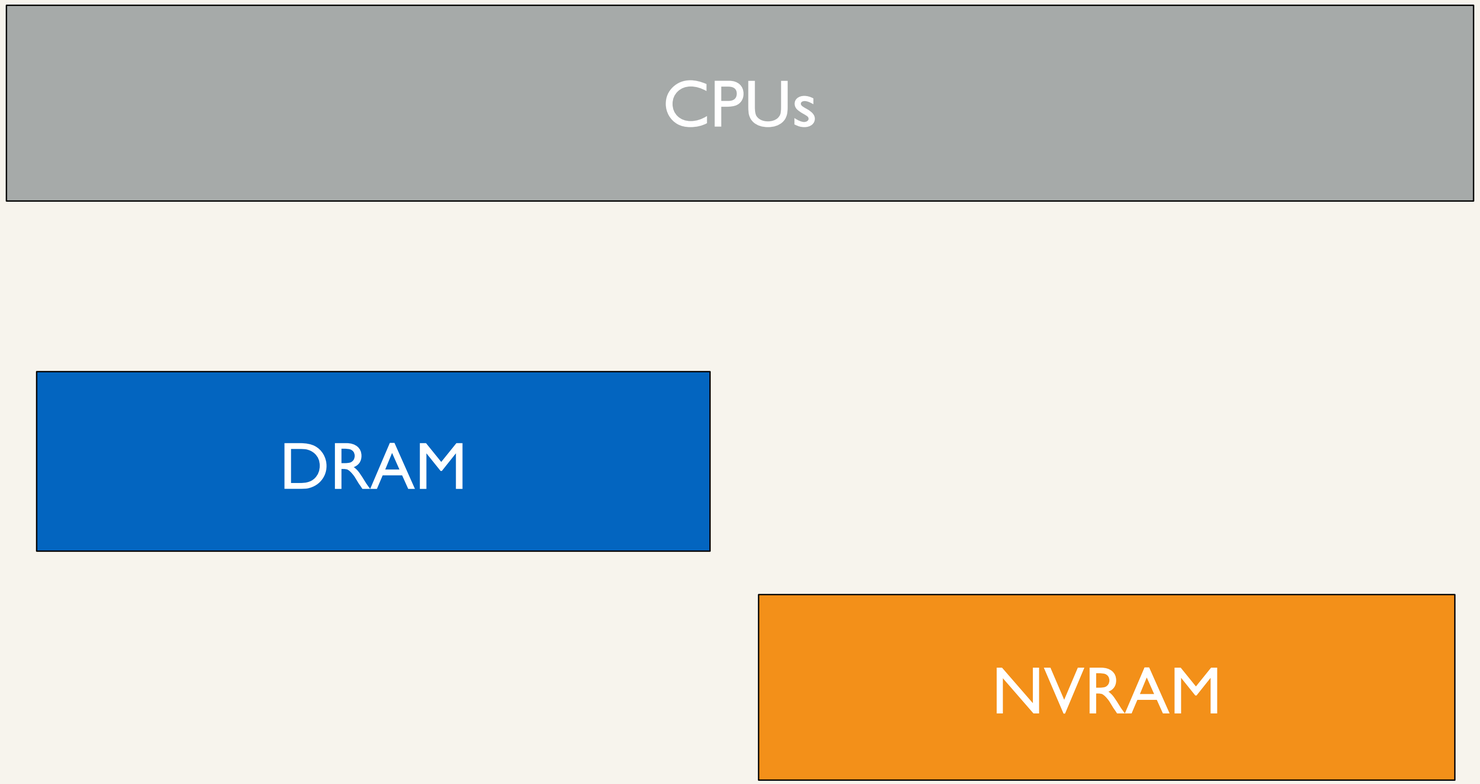
Benefits

- Algorithms avoid costly NVRAM writes, and algorithm design is independent of this cost
- Algorithms do not contribute to NVRAM wear-out

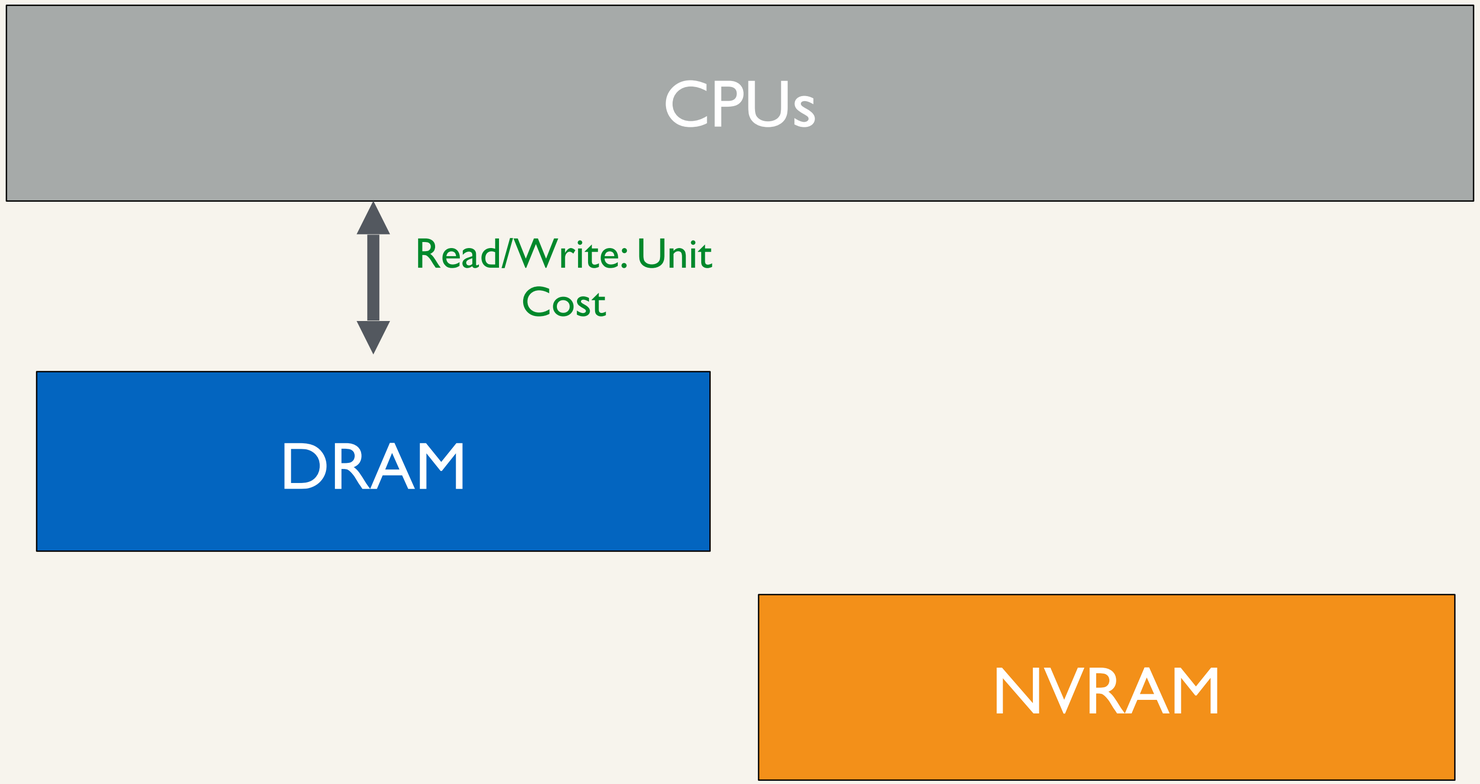


Our contribution:
This (restrictive) semi-asymmetric approach is effective for designing fast parallel graph algorithms

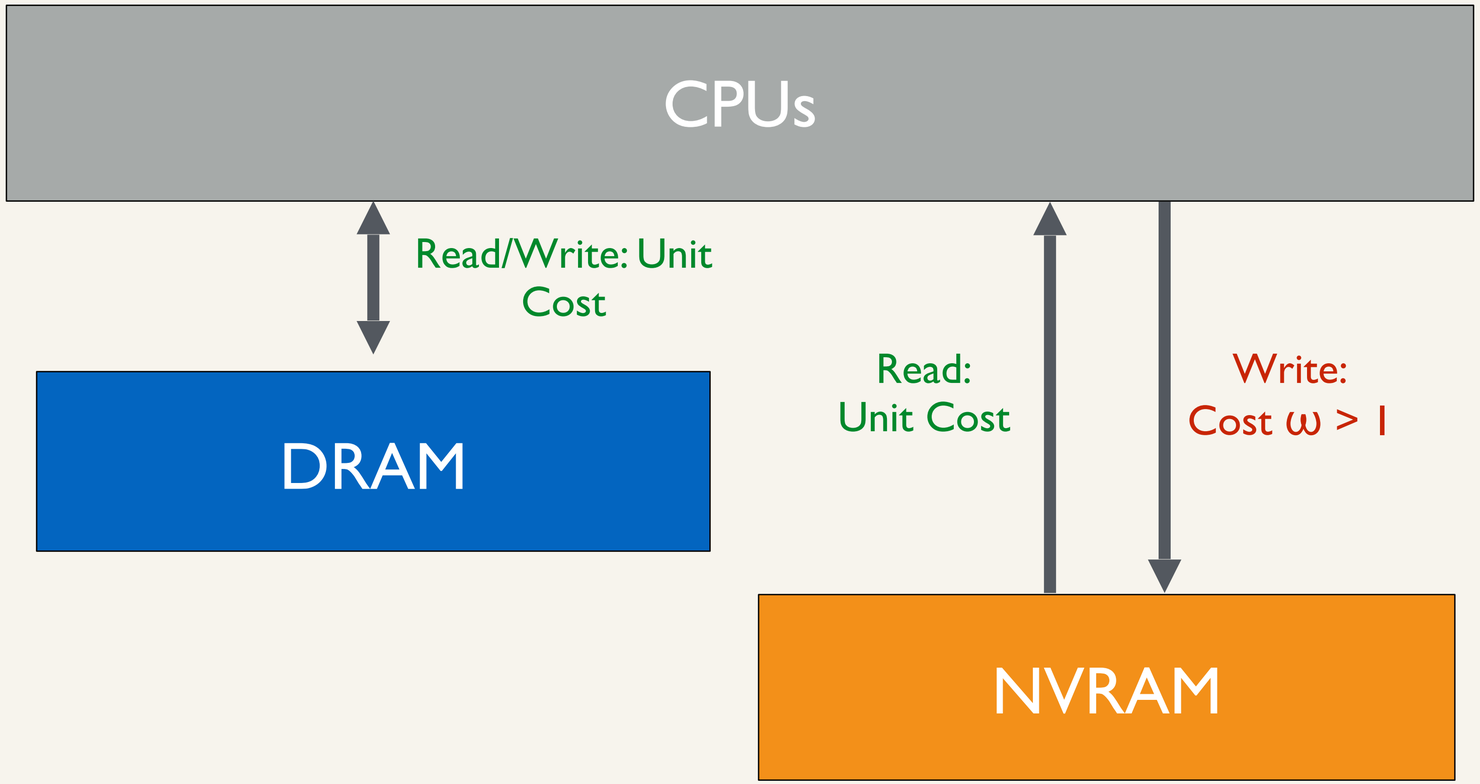
Parallel Semi-Asymmetric Model (PSAM)



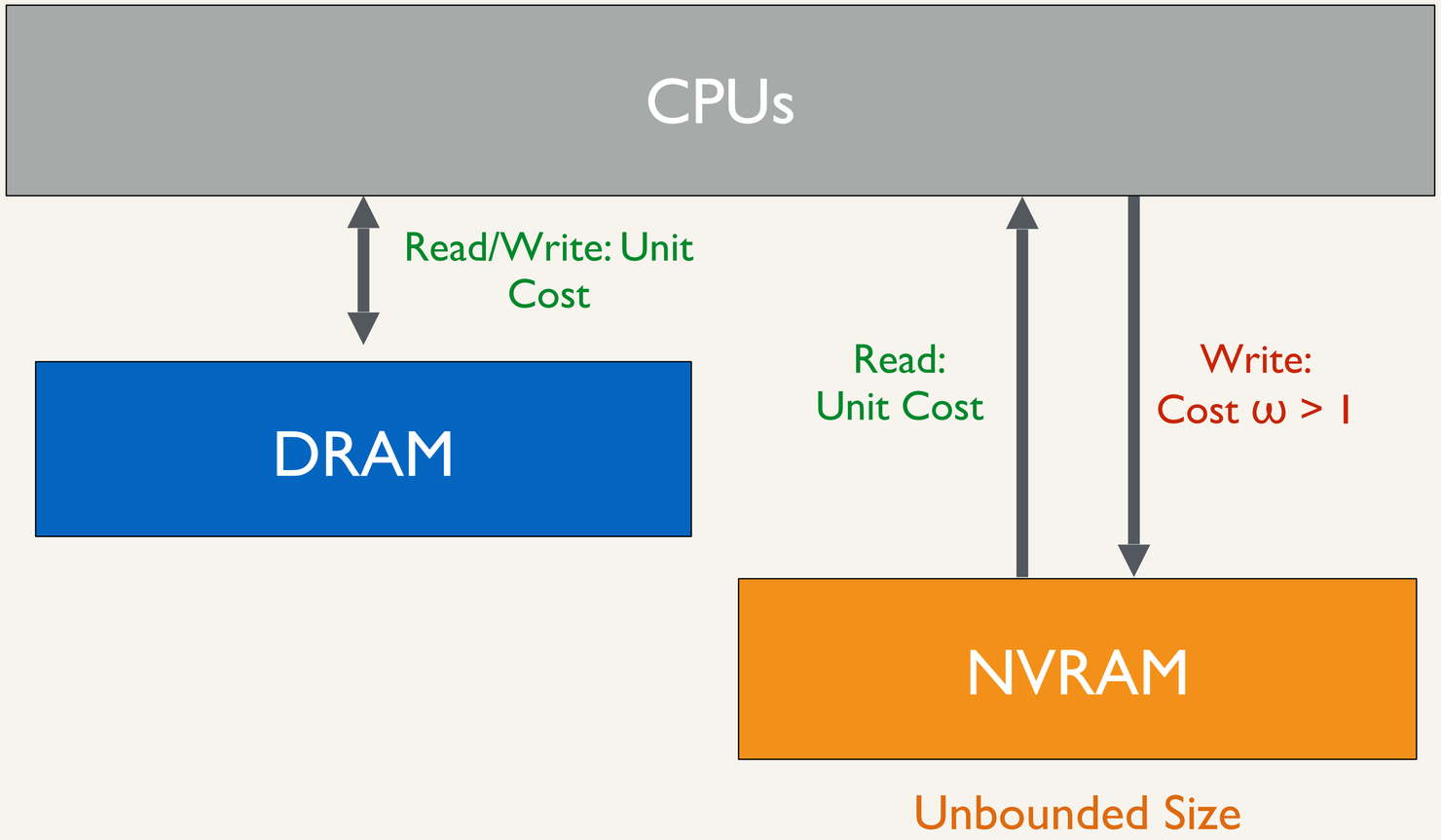
Parallel Semi-Asymmetric Model (PSAM)



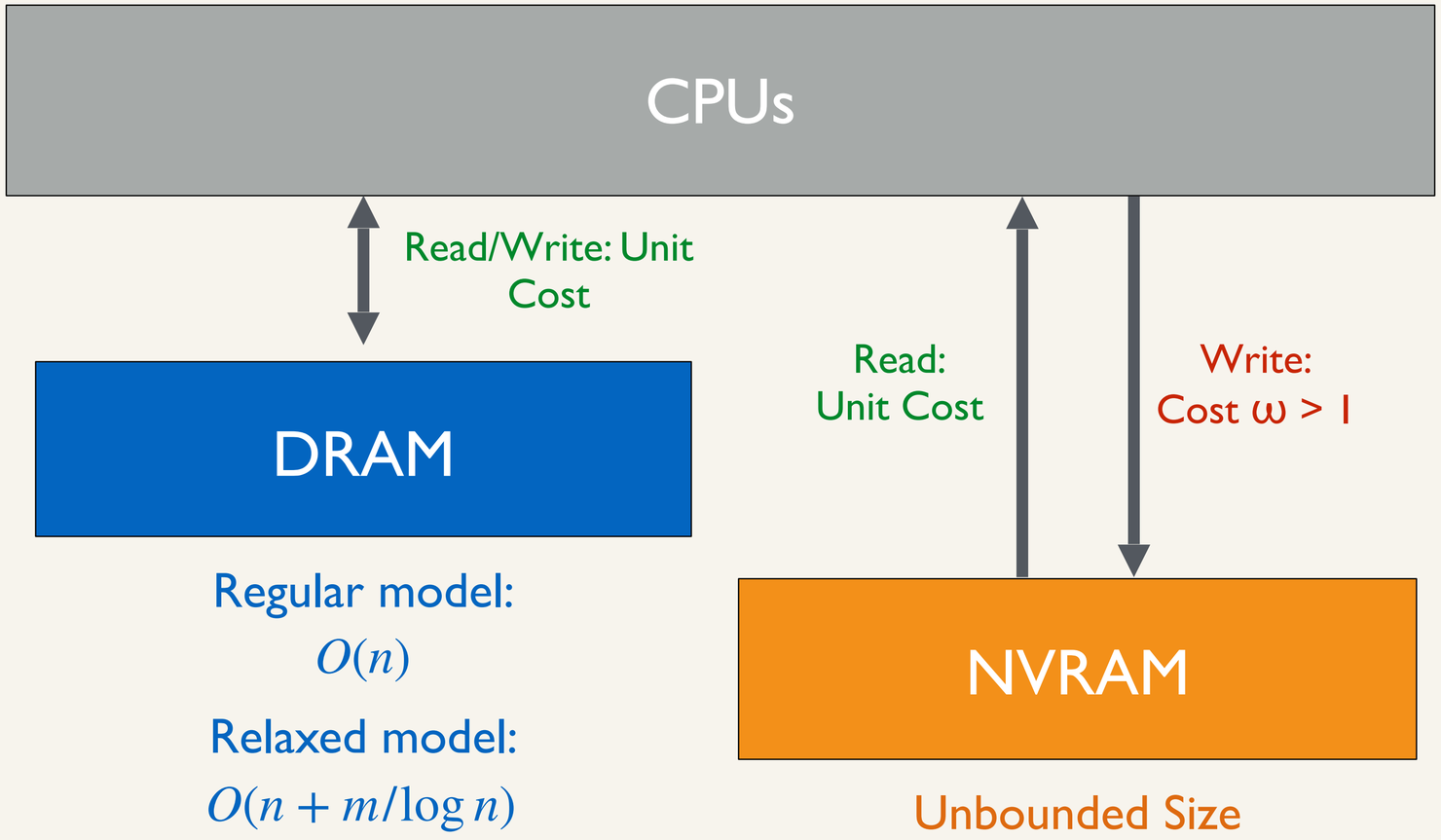
Parallel Semi-Asymmetric Model (PSAM)



Parallel Semi-Asymmetric Model (PSAM)

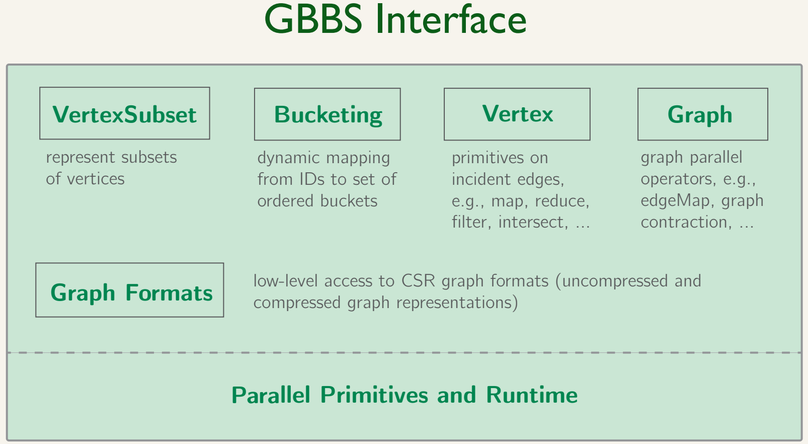


Parallel Semi-Asymmetric Model (PSAM)



Overview of Semi-Asymmetric Algorithms

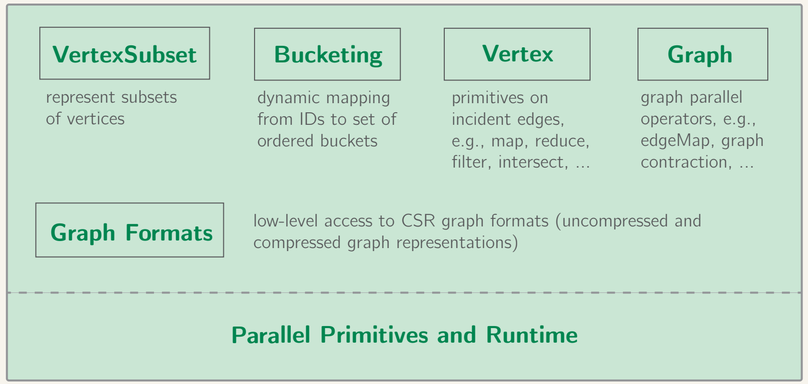
- Start with work-efficient shared-memory algorithms from the Graph Based Benchmark Suite (GBBS)
- Implement interface primitives used by GBBS algorithms (*edgeMap* and *filtering*) efficiently in the PSAM



Overview of Semi-Asymmetric Algorithms

- Start with work-efficient shared-memory algorithms from the Graph Based Benchmark Suite (GBBS)
- Implement interface primitives used by GBBS algorithms (*edgeMap* and *filtering*) efficiently in the PSAM

GBBS Interface



edgeMap

	Problem	GBBS Work	Sage Work	Sage Depth
EDGEMAPCHUNKED	**Breadth-First Search**	$O(\omega m)$	$O(m)$	$O(\mathsf{d}_G \log n)$
	Weighted BFS	$O(\omega m)^*$	$O(m)^*$	$O(\mathsf{d}_G \log n)^\ddagger$
	Bellman-Ford	$O(\omega \mathsf{d}_G m)$	$O(\mathsf{d}_G m)$	$O(\mathsf{d}_G \log n)$
	Single-Source Widest Path	$O(\omega \mathsf{d}_G m)$	$O(\mathsf{d}_G m)$	$O(\mathsf{d}_G \log n)$
	Single-Source Betweenness	$O(\omega m)$	$O(m)$	$O(\mathsf{d}_G \log n)$
	$O(k)$-Spanner	$O(\omega m)^*$	$O(m)^*$	$O(k \log n)^\ddagger$
	LDD	$O(\omega m)^*$	$O(m)^*$	$O(\log^2 n)^\ddagger$
	Connectivity	$O(\omega m)^*$	$O(m)^*$	$O(\log^3 n)^\ddagger$
	Spanning Forest	$O(\omega m)^*$	$O(m)^*$	$O(\log^3 n)^\ddagger$
	Graph Coloring	$O(\omega m)^*$	$O(m)^*$	$O(\log n + L \log \Delta)^*$
	Maxmial Independent Set	$O(\omega m)^*$	$O(m)^*$	$O(\log^2 n)^\ddagger$

GBBS work indicates the work of naively converting exisitng shared-memory algorithms from GBBS to NVRAM algorithms

Filtering (relaxed model)

	Problem	GBBS Work	Sage Work	Sage Depth
Both	**Biconnectivity**¶	$O(\omega m)^*$	$O(m)^*$	$O(\mathsf{d}_G \log n + \log^3 n)^\ddagger$
	Apx. Set Cover†	$O(\omega m)^*$	$O(m)^*$	$O(\log^3 n)^\ddagger$
Filter	**Triangle Counting**†	$O(\omega(m+n)+m^{3/2})$	$O(m^{3/2})$	$O(\log n)$
	Maximal Matching†	$O(\omega m)^*$	$O(m)^*$	$O(\log^3 m)^\ddagger$

Other Techniques

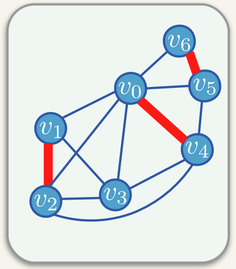
PageRank Iteration	$O(m+\omega n)$	$O(m)$	$O(\log n)$
PageRank	$O(P_{it}(m+\omega n))$	$O(P_{it} m)$	$O(P_{it} \log n)$
k-core	$O(\omega m)^*$	$O(m)^*$	$O(\rho \log n)^\ddagger$
Apx. Densest Subgraph	$O(\omega m)$	$O(m)$	$O(\log^2 n)$

Semi-Asymmetric Filtering

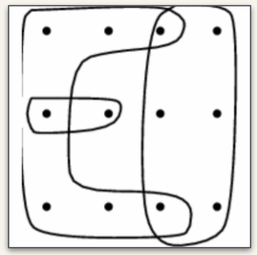
Motivation

- Some algorithms *remove*, or *batch-delete edges* over the course of their operation for work-efficiency
- Modifying the graph directly requires writing to NVRAM

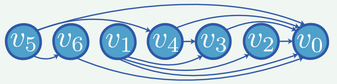
Maximal Matching



Parallel Approximate Set Cover



Triangle Counting



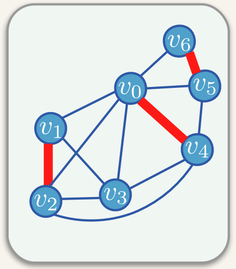
Orient edges based on a given order

Semi-Asymmetric Filtering

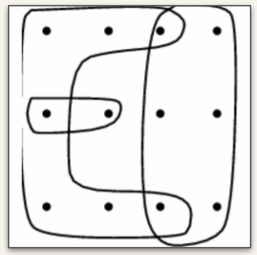
Motivation

- Some algorithms *remove*, or *batch-delete edges* over the course of their operation for work-efficiency
- Modifying the graph directly requires writing to NVRAM

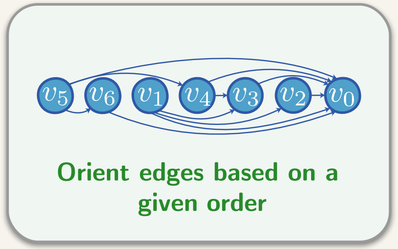
Maximal Matching



Parallel Approximate Set Cover

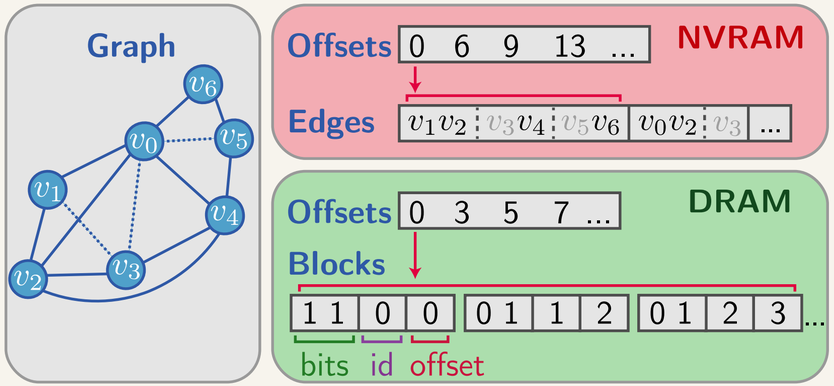


Triangle Counting

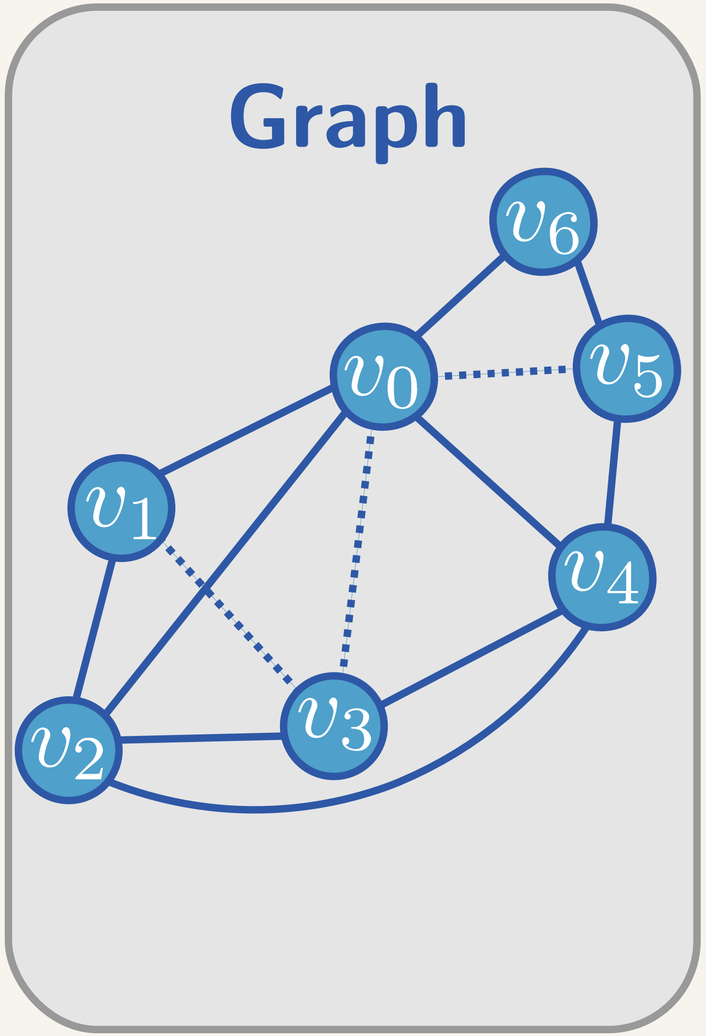


Semi-Asymmetric Filtering

- Work in the relaxed model
- Use one bit per edge and mirror the CSR structure (in NVRAM) using a blocked approach in DRAM



Semi-Asymmetric Filtering



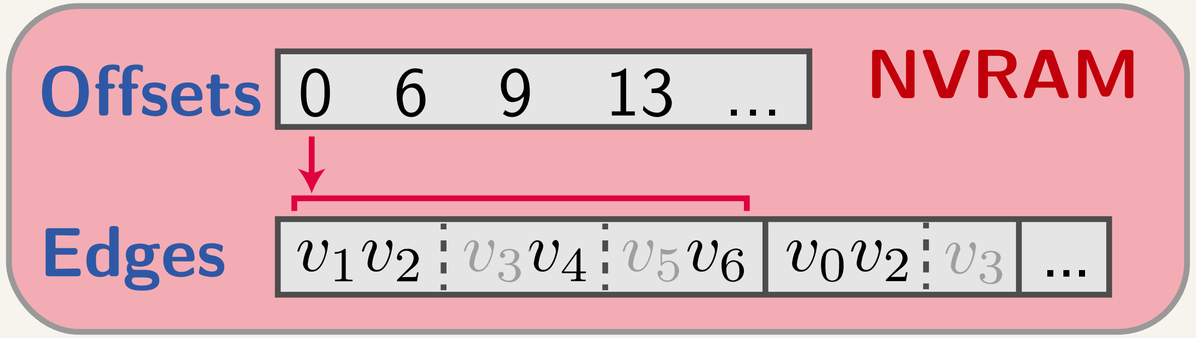
logically deleted

present in graph

Semi-Asymmetric Filtering

High-level Approach

(i) Set a filter block size, and logically chunk the CSR structure into chunks of this size

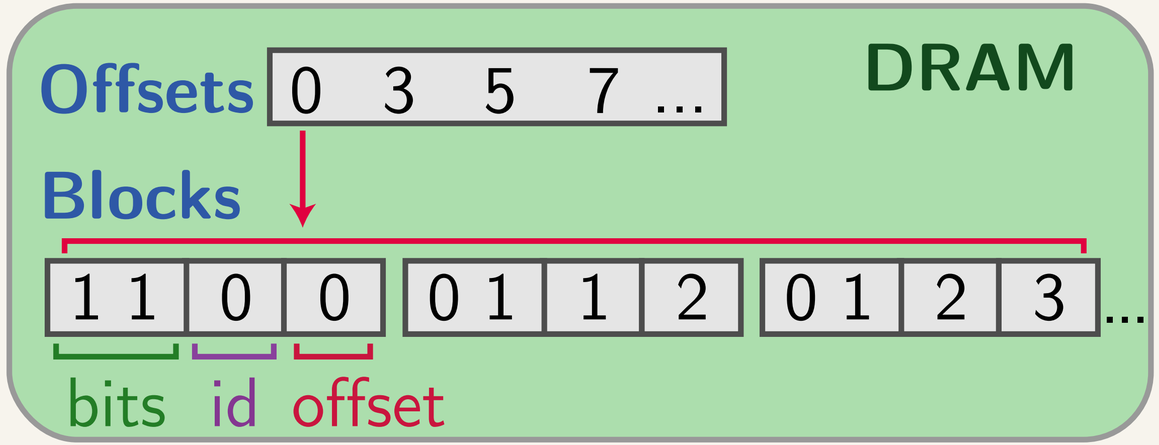


Graph in CSR format, stored in NVRAM $(\mathscr{F}_B = 2)$

Semi-Asymmetric Filtering

High-level Approach

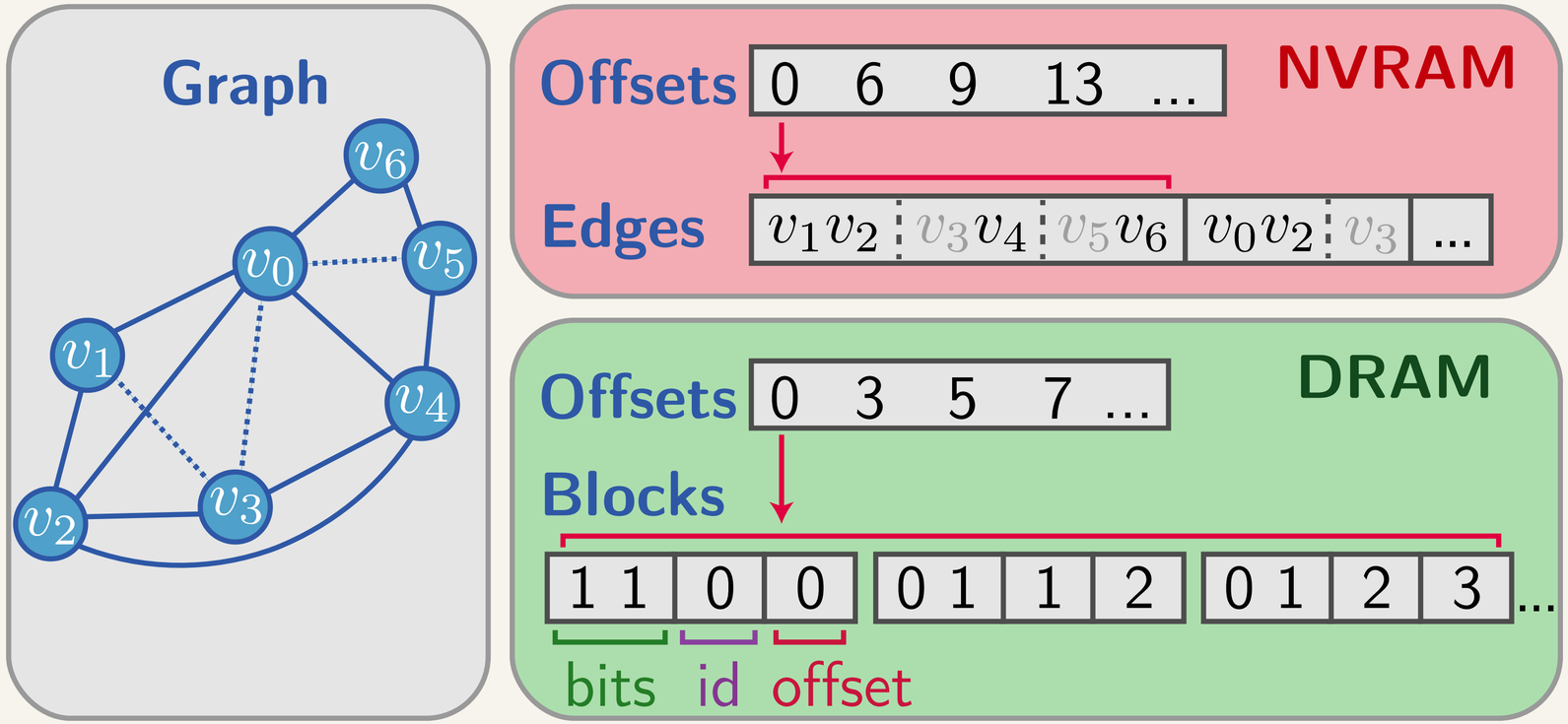
(ii) Create a "mirrored" filter structure in DRAM, storing 1 bit per edge in NVRAM



GraphFilter in CSR format, stored in DRAM ($\mathscr{F}_B = 2$)

Semi-Asymmetric Filtering

Structure Overview



Note: Blocks with no "1" bits remaining are deleted

Relationship to Other Models

Semi-External Memory (SE) Model

- SE model performs block-transfers, with a focus on I/O cost [0, 1]
- Both PSAM and SE models provide the same amount of DRAM, but SE does not account for DRAM reads and writes

Sources:
[0] Abello et al. A Functional Approach to External Graph Algorithms (2002)
[1] Zheng et al. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs (2015)
[2] Blelloch et al. Efficient algorithms with asymmetric read and write costs (2016)
[3] Ben-David et al. Parallel algorithms for asymmetric read-write costs (2016)

Relationship to Other Models

Semi-External Memory (SE) Model

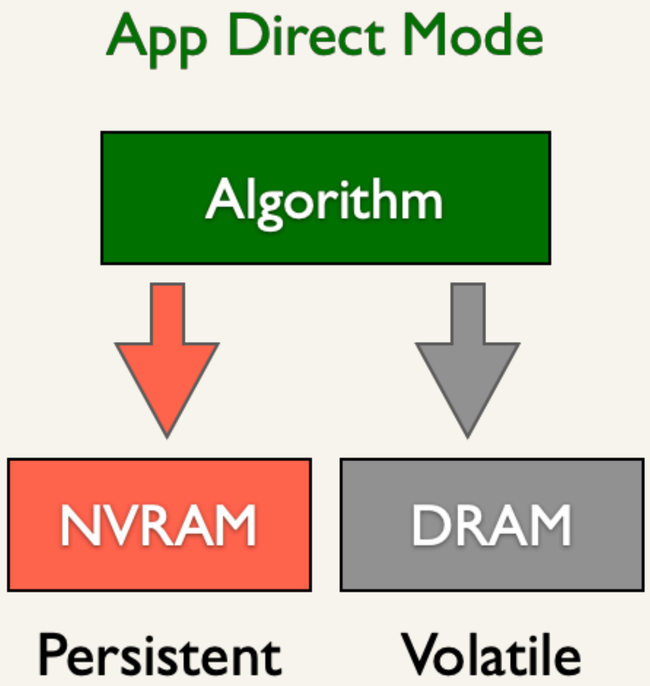
- SE model performs block-transfers, with a focus on I/O cost [0, 1]
- Both PSAM and SE models provide the same amount of DRAM, but SE does not account for DRAM reads and writes

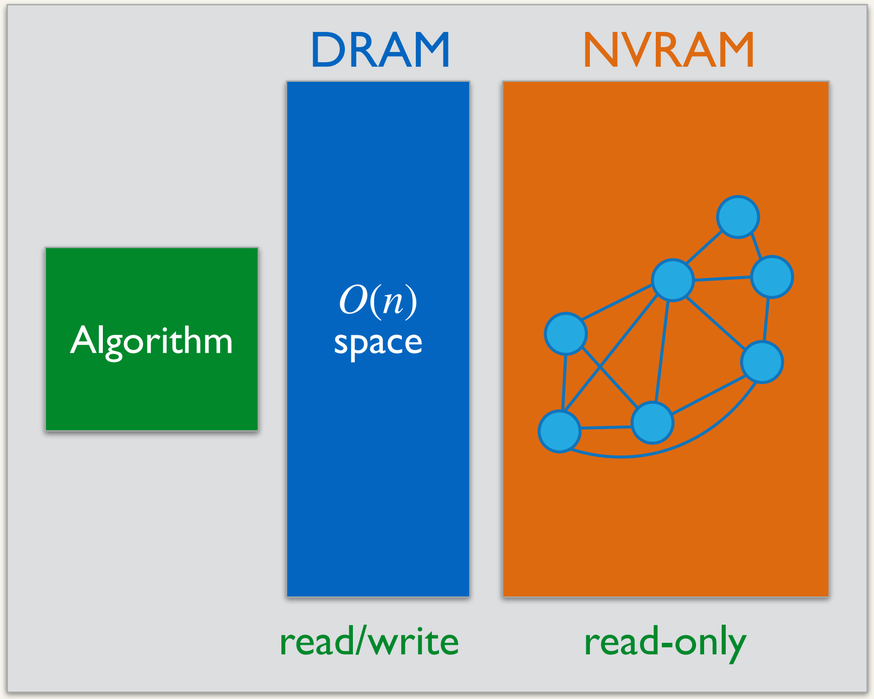
Asymmetric RAM and Asymmetric Nested Parallel Models

- Both ARAM [2] and ANP [3] models capture asymmetry of writing to NVRAM
- Unlike ARAM/ANP models, the PSAM includes a fast memory, and is specialized for graph problems

Sources:
[0] Abello et al. A Functional Approach to External Graph Algorithms (2002)
[1] Zheng et al. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs (2015)
[2] Blelloch et al. Efficient algorithms with asymmetric read and write costs (2016)
[3] Ben-David et al. Parallel algorithms for asymmetric read-write costs (2016)

Semi-Asymmetric Graph Engine (Sage) Approach

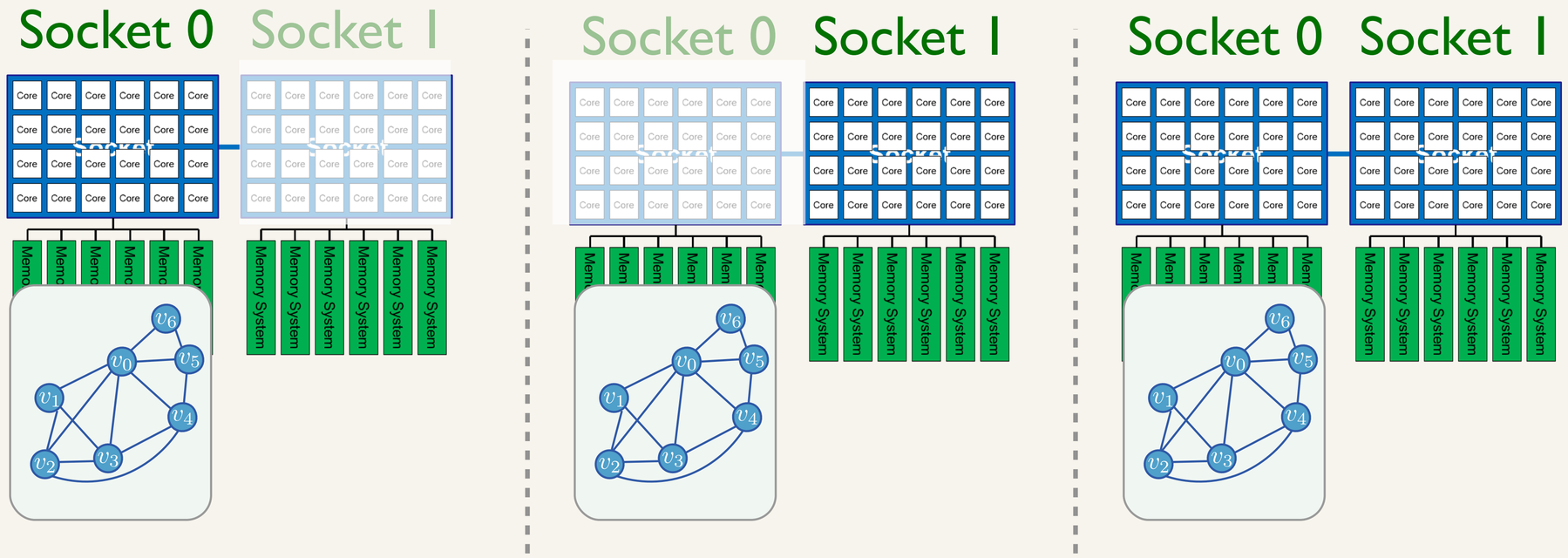




AppDirect Mode enables a direct implementation of PSAM algorithms

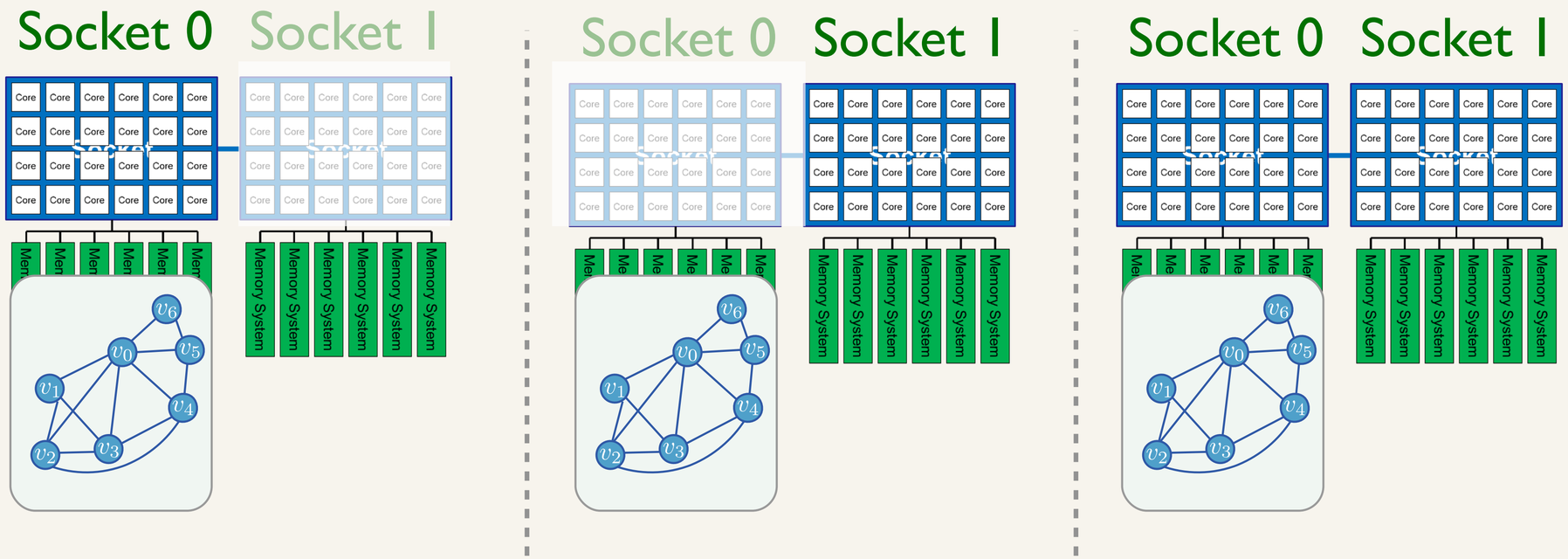
NUMA Optimization in Sage

Consider an algorithm that maps over all vertices, and for each vertex performs a reduction over the neighbors of the vertex



Three experiments based on (threads, storage)

NUMA Optimization in Sage



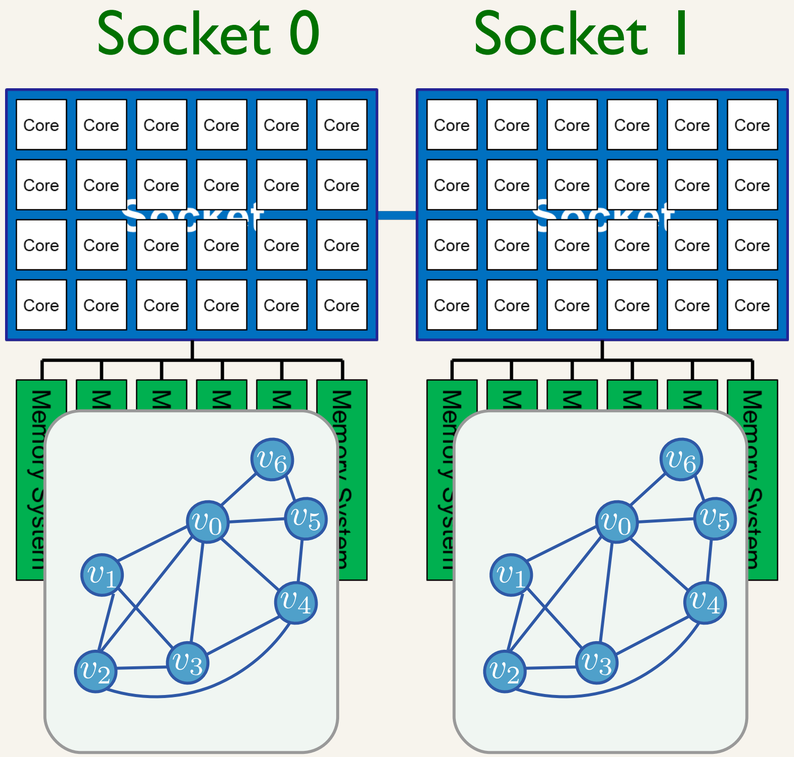
7 s

> 4x slower
first run
~7s subsequently

26 s

Cross-socket NVM reads should be avoided

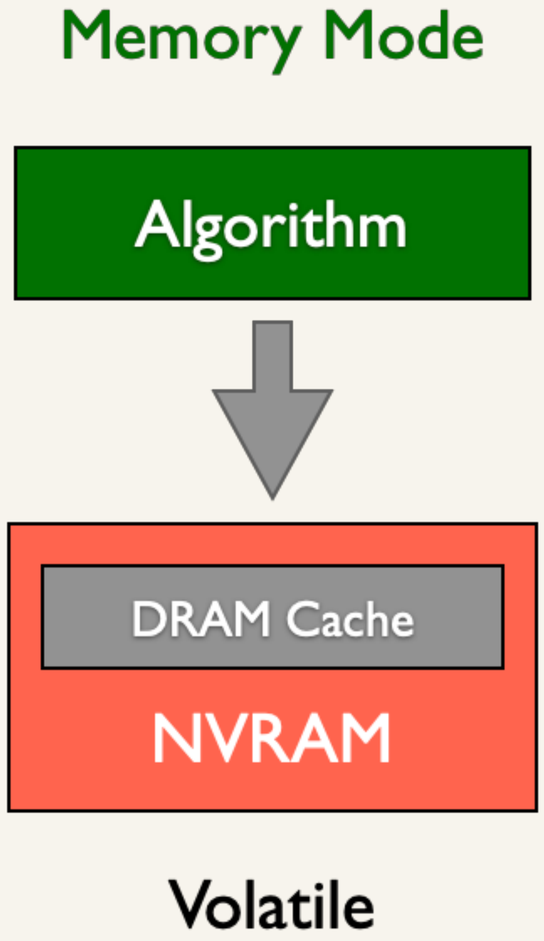
NUMA Optimization in Sage



4.3 s for microbenchmark

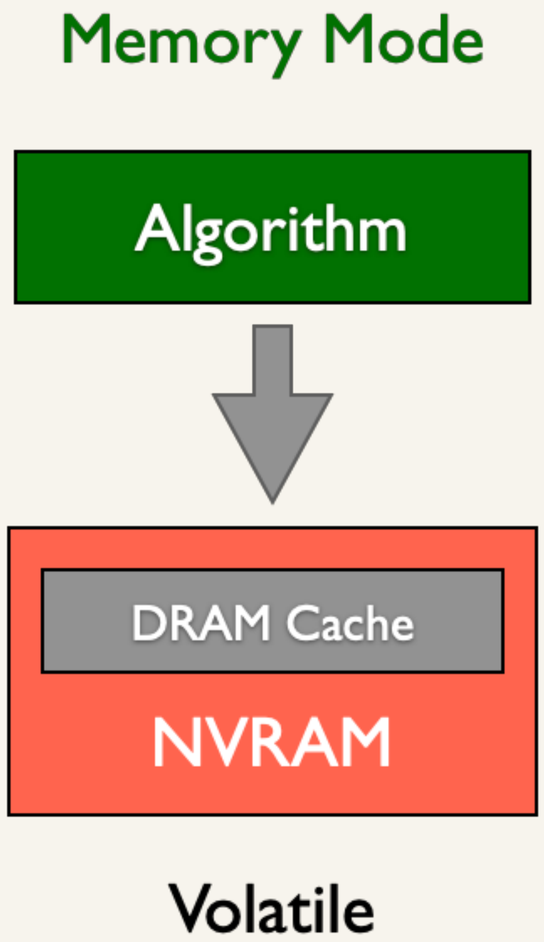
Both graphs stored in compressed CSR format

Existing Approaches: DRAM as a Cache



- Applications do not distinguish between DRAM and NVRAM
- Existing shared-memory software does not require modification
- Workloads that are larger than DRAM can involve costly NVRAM writes

Existing Approaches: DRAM as a Cache

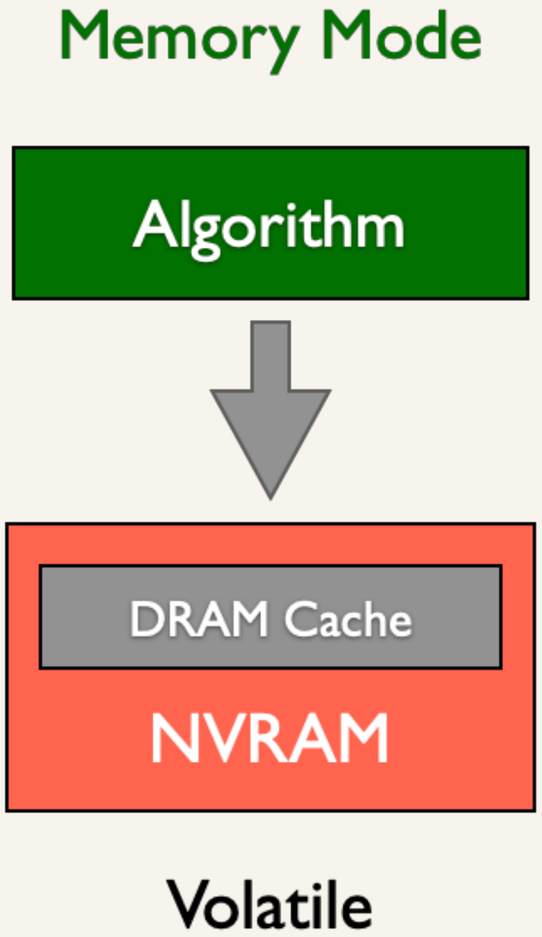


- Applications do not distinguish between DRAM and NVRAM
- Existing shared-memory software does not require modification
- Workloads that are larger than DRAM can involve costly NVRAM writes

Galois (Gill et al.)

- Gill et al. study the performance of the Galois engine using MemMode
- They show promising results for scaling to larger than DRAM sizes

Existing Approaches: DRAM as a Cache



- Applications do not distinguish between DRAM and NVRAM
- Existing shared-memory software does not require modification
- Workloads that are larger than DRAM can involve costly NVRAM writes

Galois (Gill et al.)

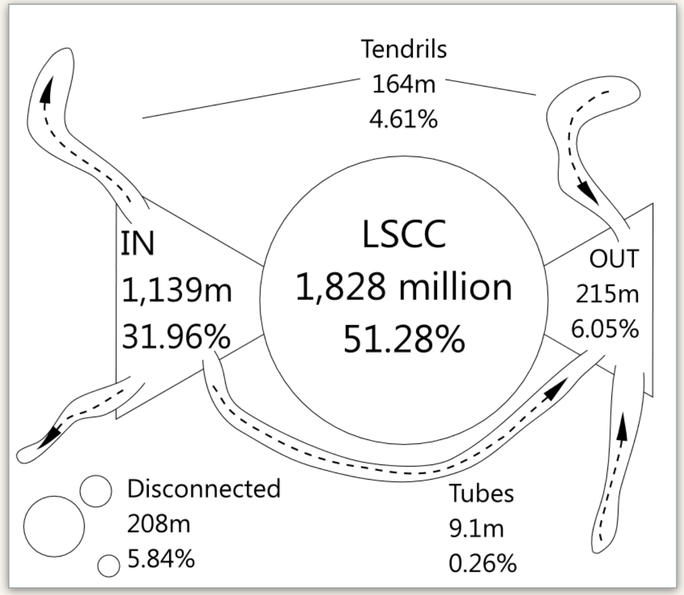
- Gill et al. study the performance of the Galois engine using MemMode
- They show promising results for scaling to larger than DRAM sizes

How does our approach compare?

Results for Larger-than-DRAM Graphs

WebDataCommons Graph

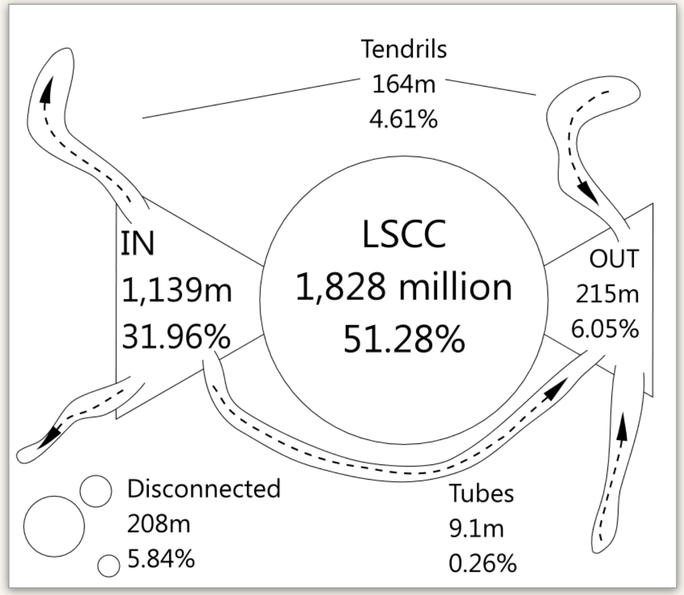
- Largest publicly available graph today
- 3.5B vertices connected by 128B edges (225B symmetrized)



Results for Larger-than-DRAM Graphs

WebDataCommons Graph

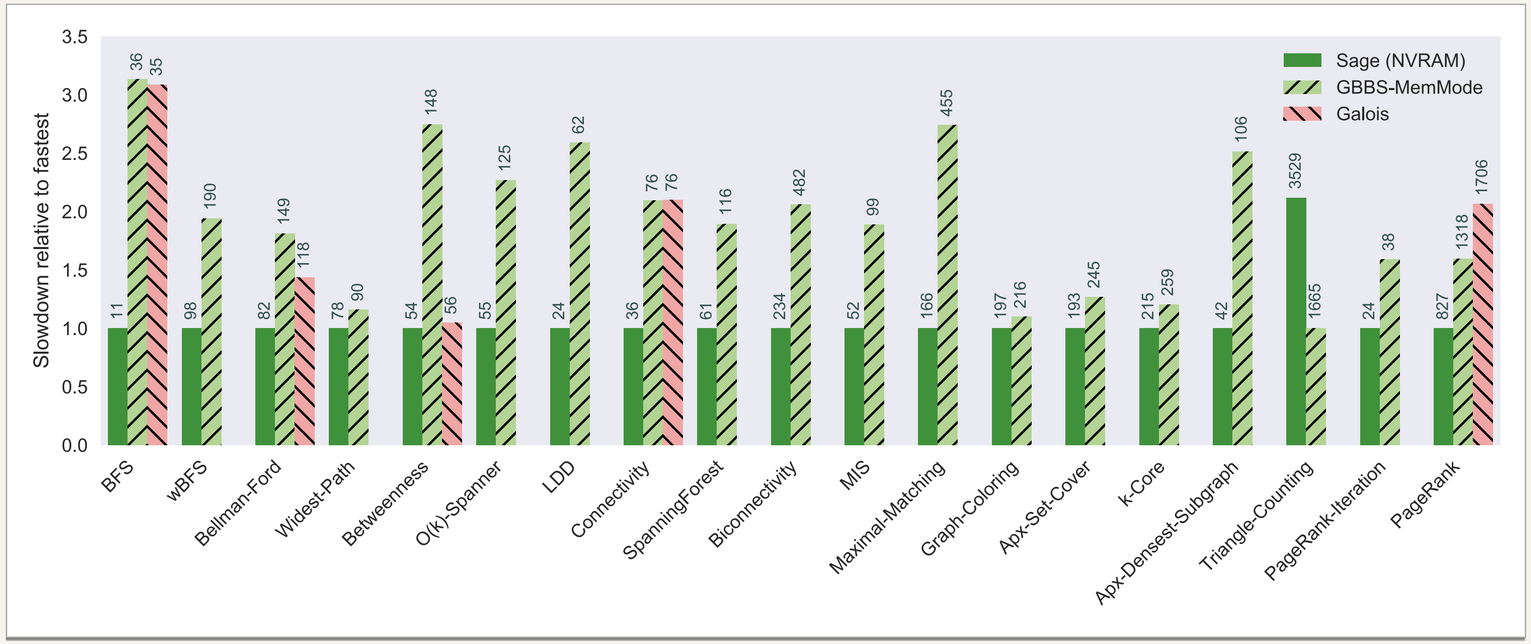
- Largest publicly available graph today
- 3.5B vertices connected by 128B edges (225B symmetrized)



Experiment

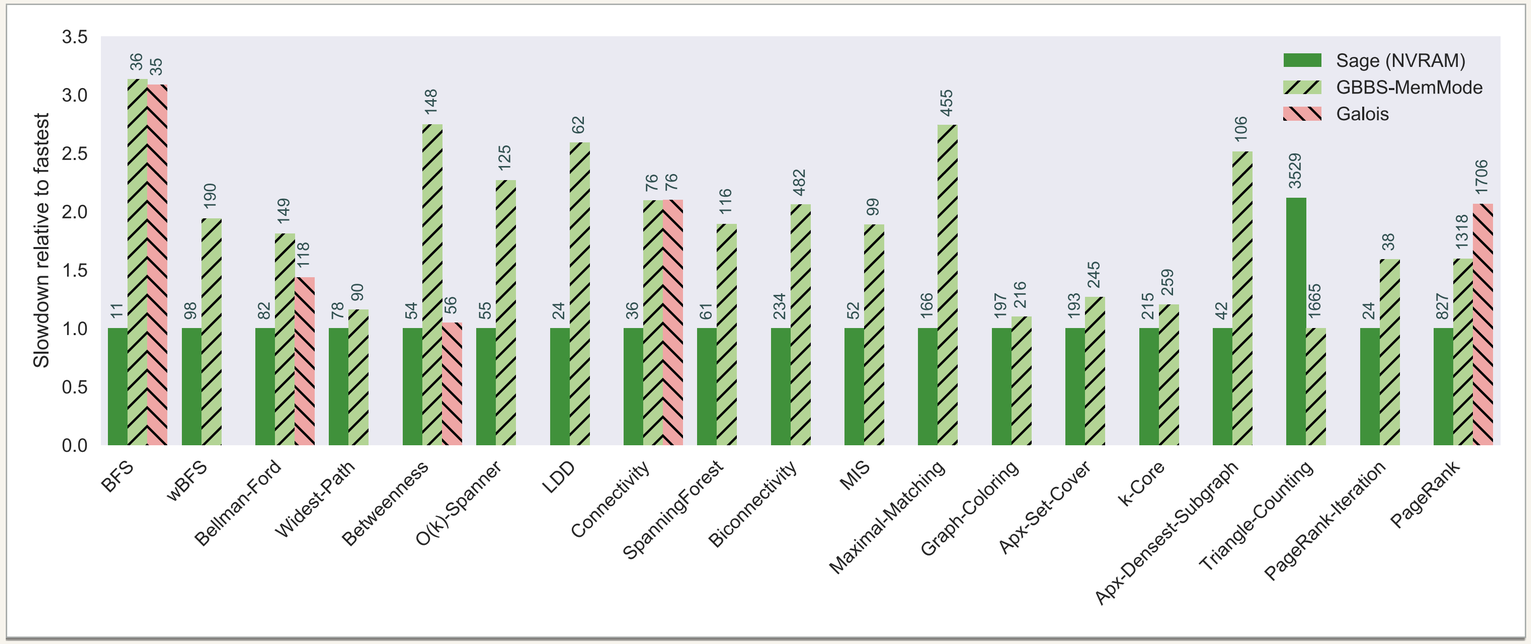
- Compare Sage results with
  - GBBS using MemMode (existing shared-memory codes)
  - Galois using MemMode (using numbers reported by authors on the same machine)

Results for Larger-than-DRAM Graphs



Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

Results for Larger-than-DRAM Graphs



Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

1.94x speedup on average over Galois (state-of-the-art existing approach to NVRAM graph processing), and 1.87x speedup over simply running GBBS codes using MemMode

Results for Graphs Stored in Main Memory

ClueWeb Graph

- Large web crawl with ~1B vertices connected by 42B edges (74B symmetrized)
- Graph fits entirely in the main memory of our machine

libvmmalloc: see https://pmem.io/pmdk/libvmmalloc/

Results for Graphs Stored in Main Memory

ClueWeb Graph

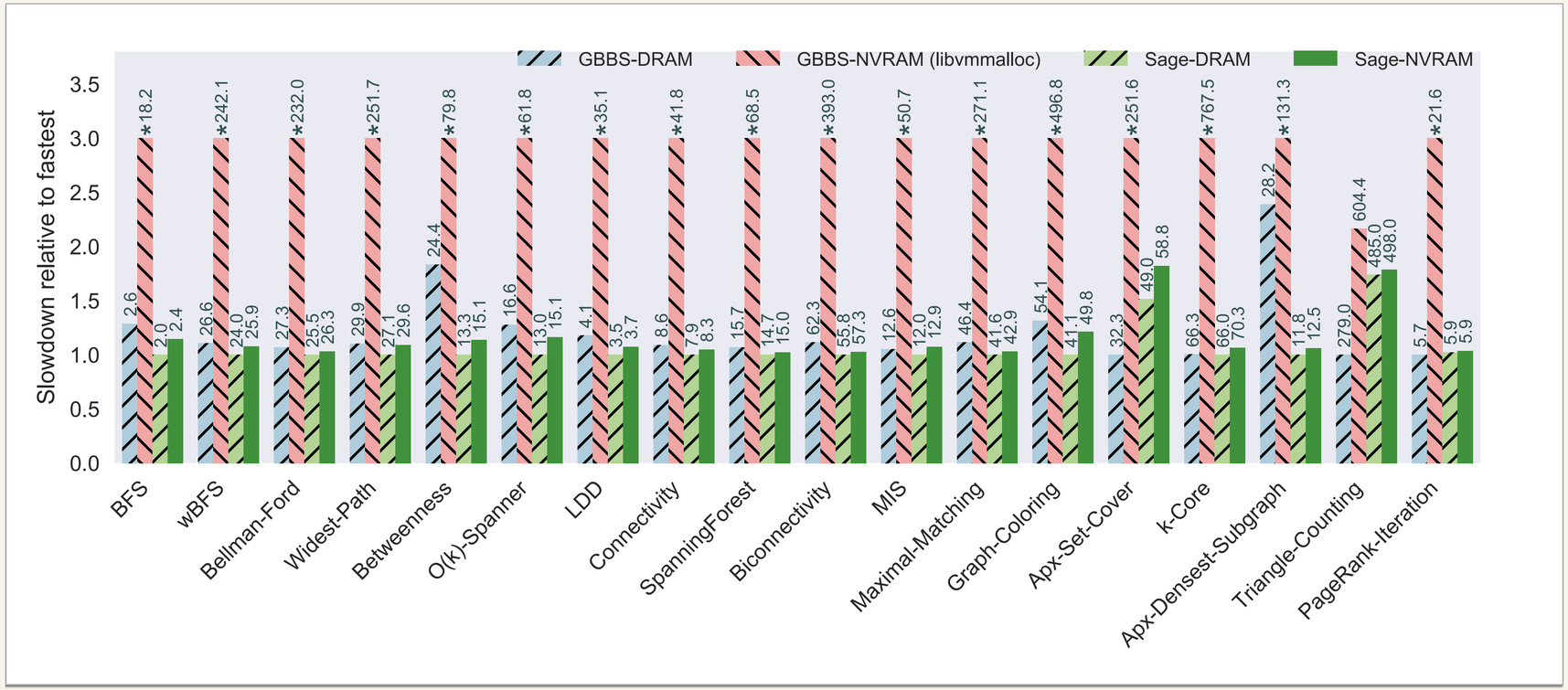
- Large web crawl with ~1B vertices connected by 42B edges (74B symmetrized)
- Graph fits entirely in the main memory of our machine

Experiment

- Compare Sage (graph stored on NVRAM) with
  - Sage (graph stored in DRAM)
  - GBBS (graph stored in DRAM)
  - GBBS with libvmmalloc (graph stored on NVRAM)

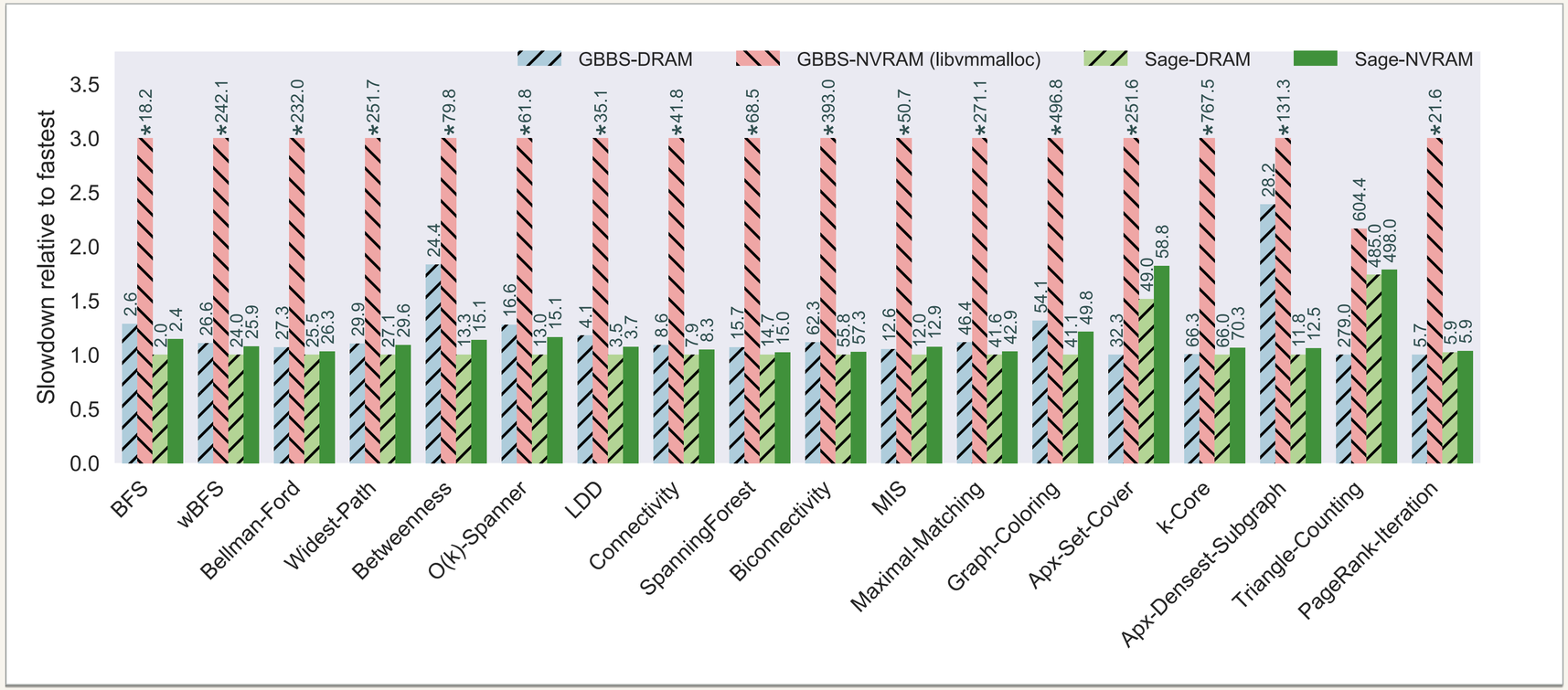
libvmmalloc: see https://pmem.io/pmdk/libvmmalloc/

Results for Graphs Stored in Main Memory



Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

Results for Graphs Stored in Main Memory



Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

Sage provides DRAM-competitive performance even when reading graph from NVRAM (only 5% slower on average)

Lessons and Directions for Future Work

Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

- NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach

Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

- NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach

Utilize App-Direct Mode

- Nearly 2x improvement for App-Direct based PSAM algorithms over two fast Memory Mode approaches

Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

- NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach

Utilize App-Direct Mode

- Nearly 2x improvement for App-Direct based PSAM algorithms over two fast Memory Mode approaches

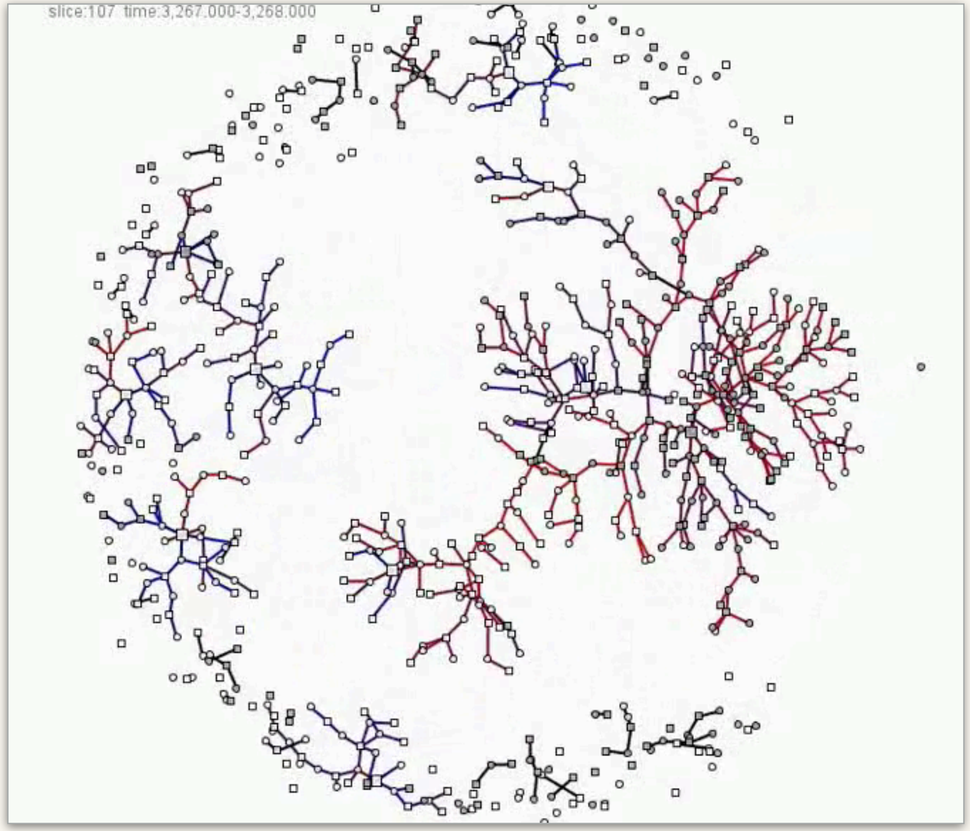
Avoid NVRAM Writes

- PSAM implementations which only read from NVRAM are over 6x faster than our algorithms which write to NVRAM (using libvmmalloc)

Streaming Graph Processing

Dynamic Graph Processing

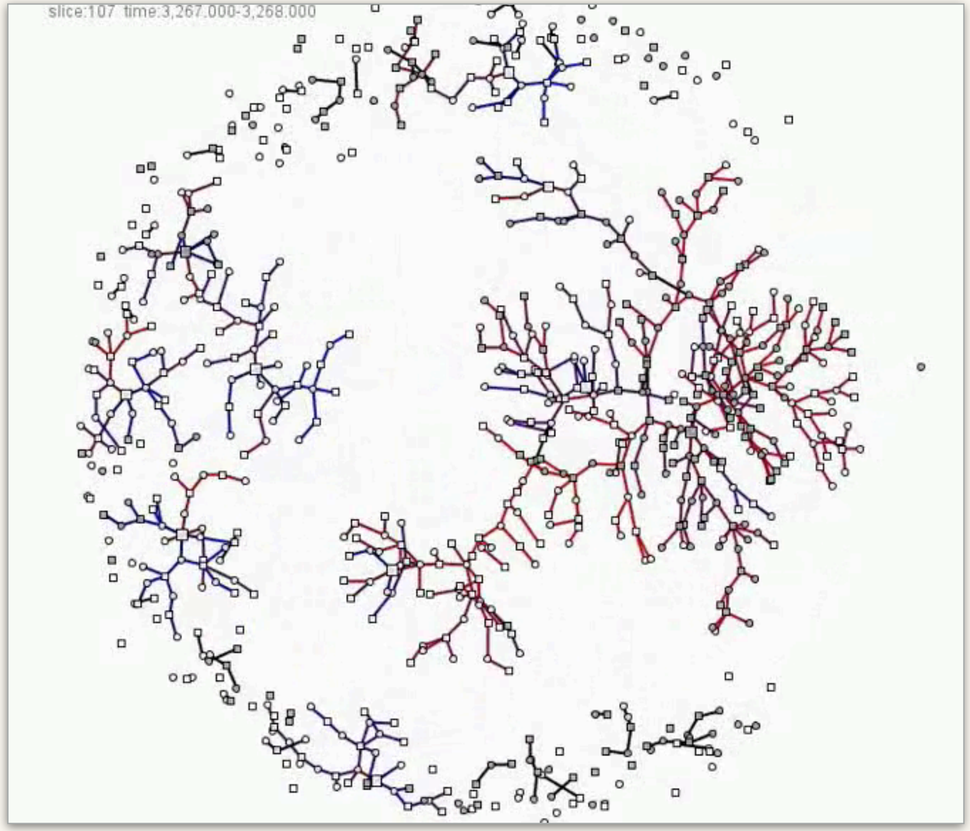
Measuring the spread of infections



Source: Infection transmission in a dynamic network

Dynamic Graph Processing

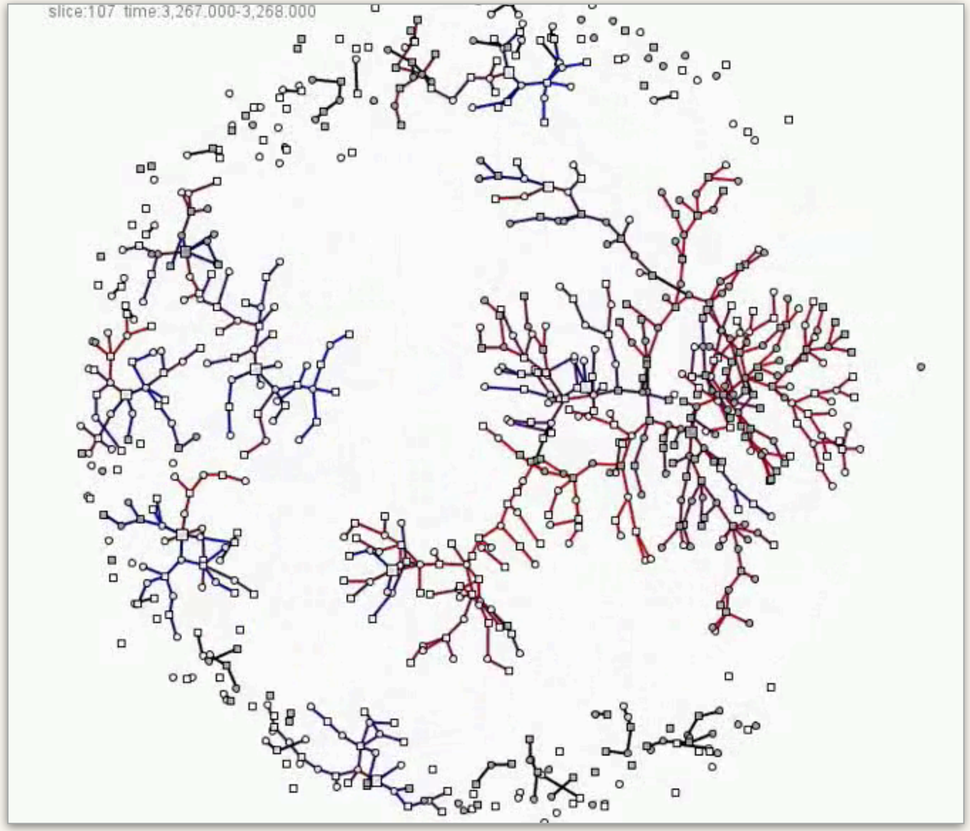
Measuring the spread of infections



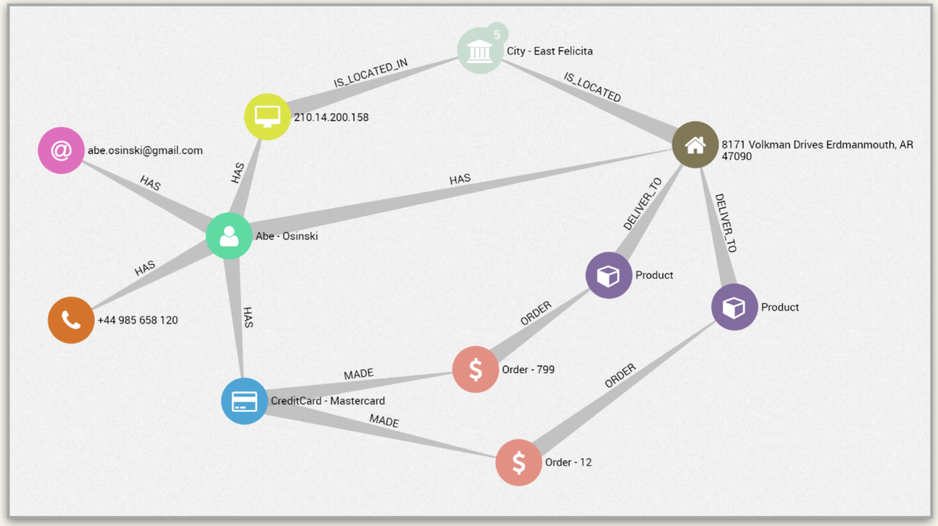
Source: Infection transmission in a dynamic network

Dynamic Graph Processing

Measuring the spread of infections



Preventing money laundering and fraud



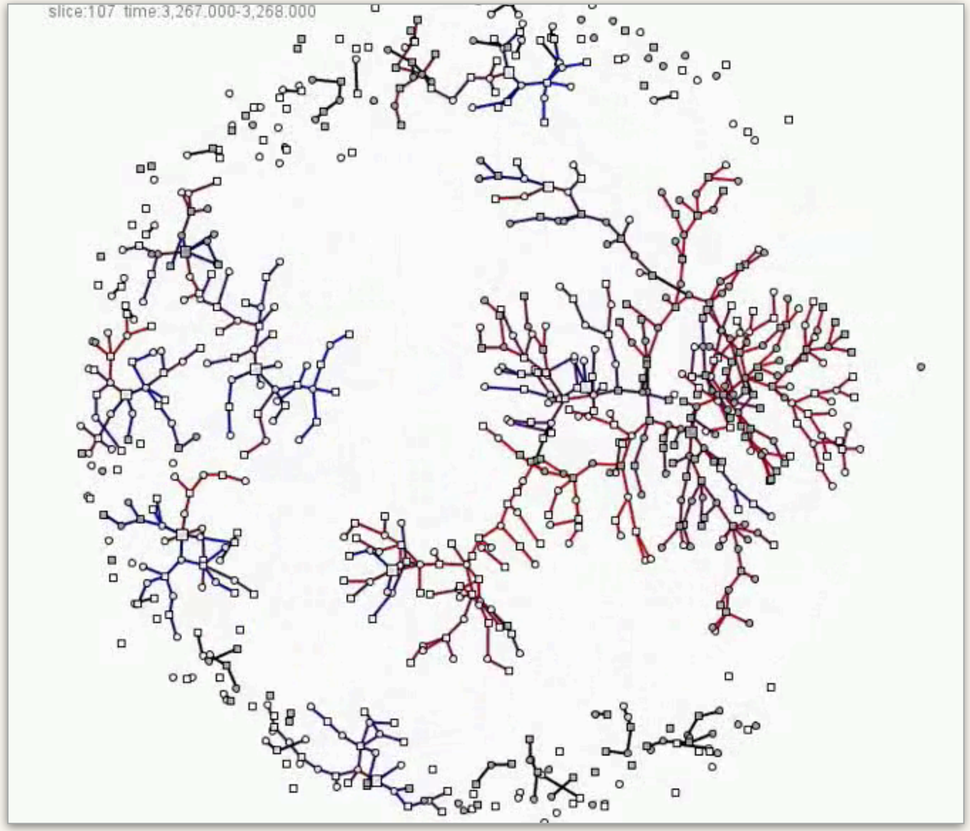
Other Applications

- Recommendation Systems
- Geospatial Systems

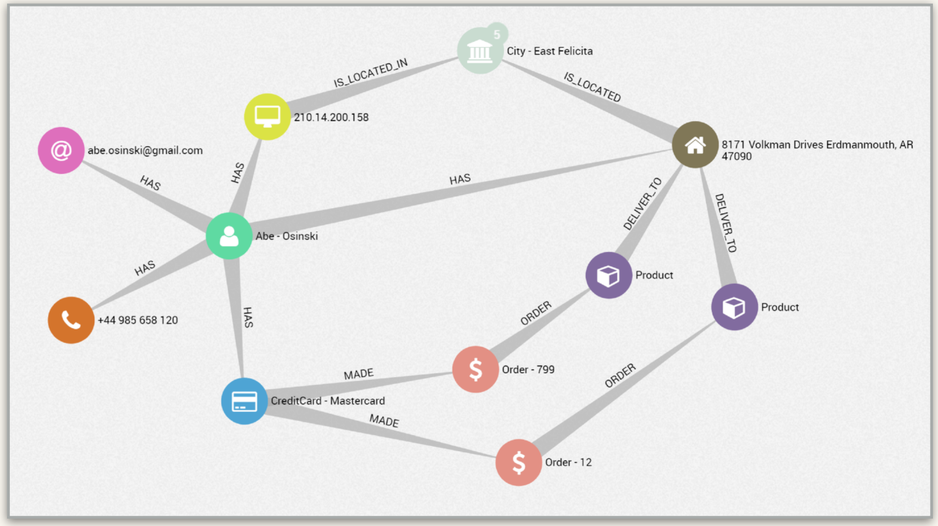
Source: Infection transmission in a dynamic network

Dynamic Graph Processing

Measuring the spread of infections



Preventing money laundering and fraud



Other Applications

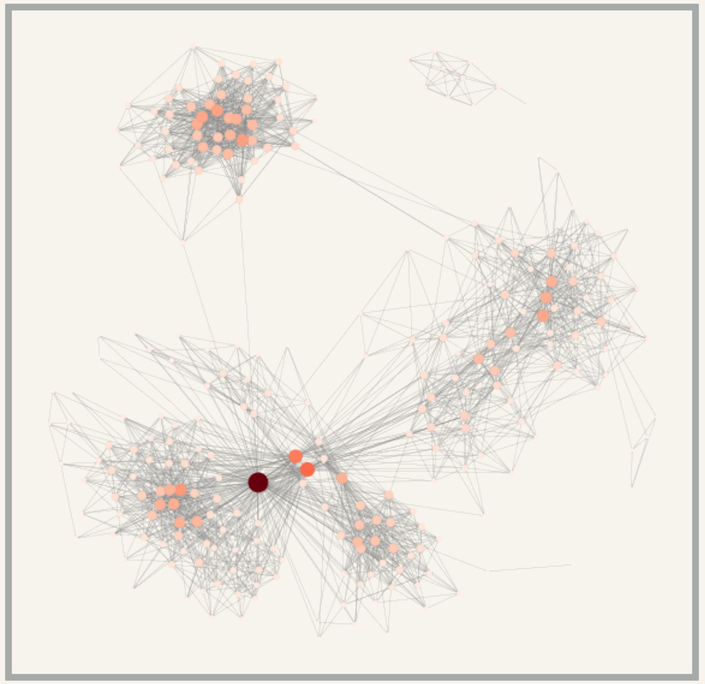
- Recommendation Systems
- Geospatial Systems

Many important applications must maintain information about evolving graphs!

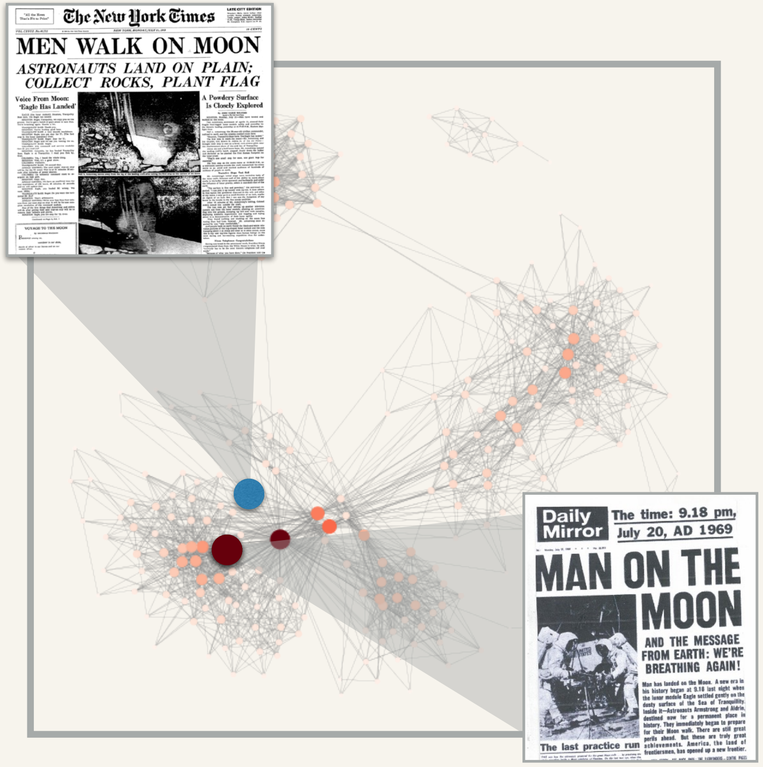
Source: Infection transmission in a dynamic network

Dynamic Graph Processing: Example

Dynamic Graph Processing: Example

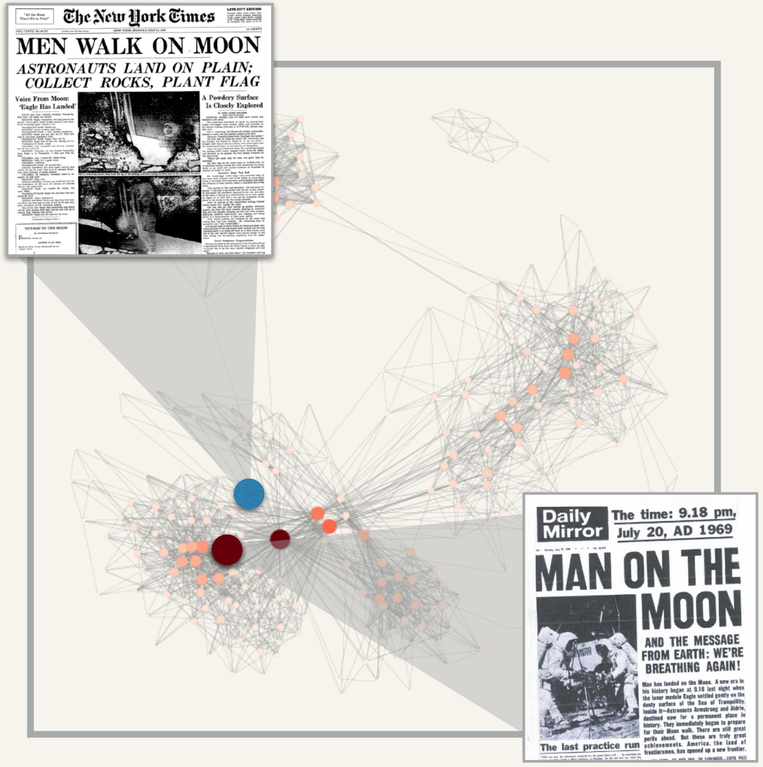


Dynamic Graph Processing: Example



Dynamic Graph Processing: Example

Update Stream



Dynamic Graph Processing: Example

Update Stream

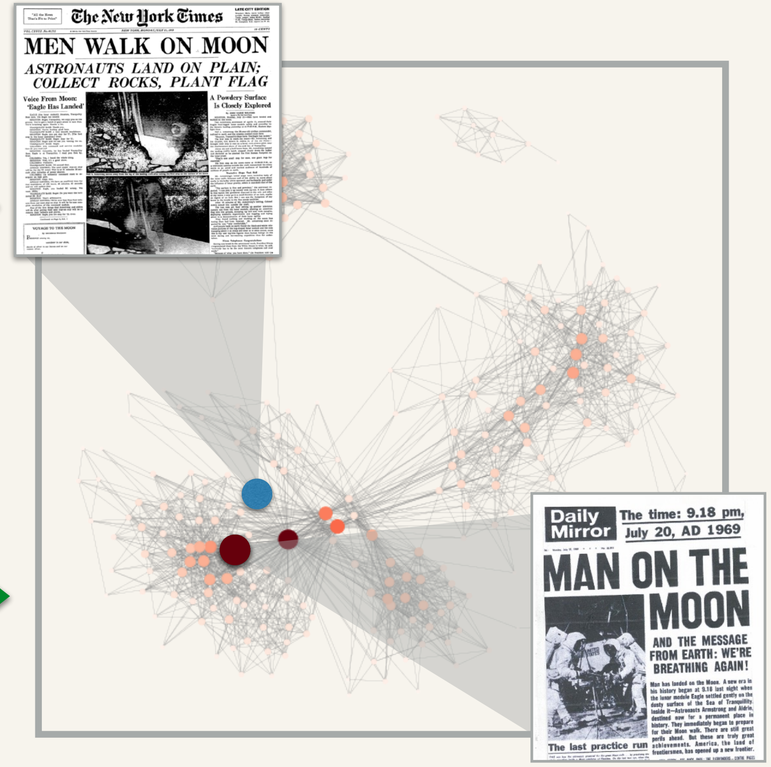
The Washington Post

NBC NEWS

REUTERS

Query Stream

- Fetch similar vertices
- Clustering Coefficients
- Fetch vertex u's neighbors
- Centrality Ranking



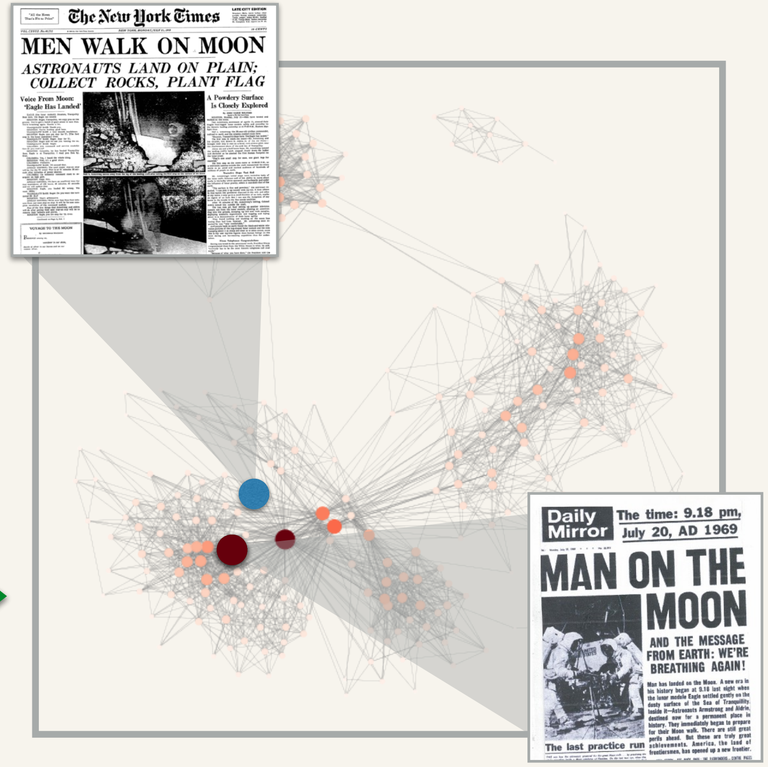
Dynamic Graph Processing: Example

Update Stream

The Washington Post

NBC NEWS

REUTERS



Query Stream

- Fetch similar vertices
- Clustering Coefficients
- Fetch vertex u's neighbors
- Centrality Ranking

Streaming Graph Processing

Update the graph (in parallel);
Execute arbitrary queries on snapshots.

Dynamic Graph Processing: Example

Update Stream

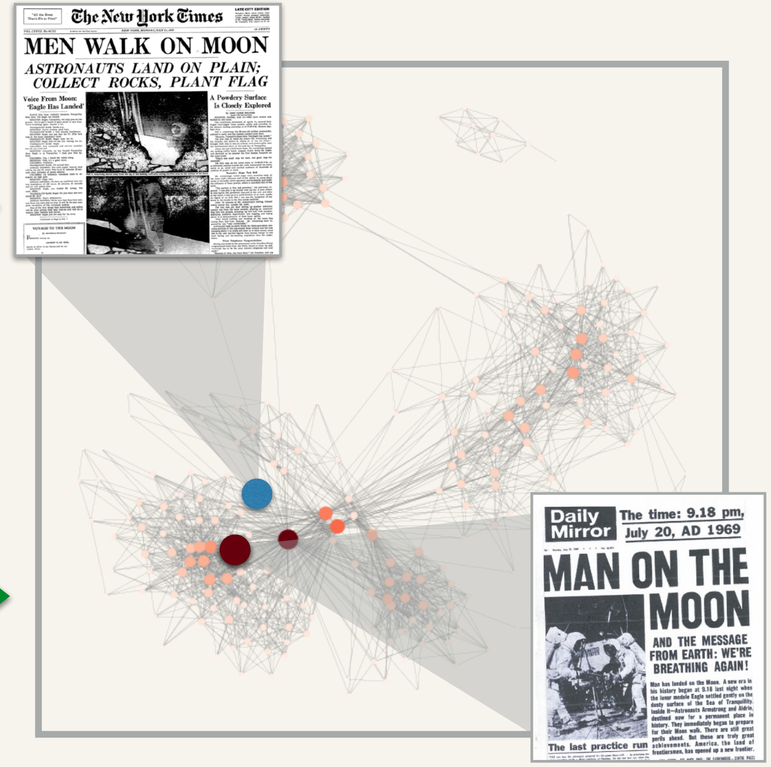
The Washington Post

NBC NEWS

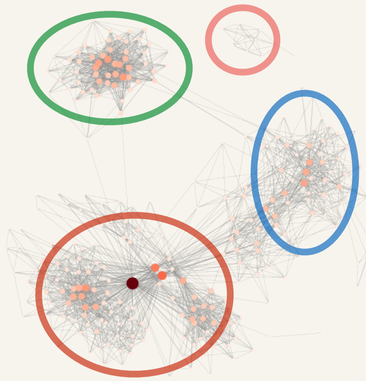
REUTERS

Query Stream

- Fetch similar vertices
- Clustering Coefficients
- Fetch vertex u's neighbors
- Centrality Ranking



Clustering



Triangle Counting

Connected Components

Streaming Graph Processing

Update the graph (in parallel);
Execute arbitrary queries on snapshots.

Dynamic Graph Processing: Example

Update Stream

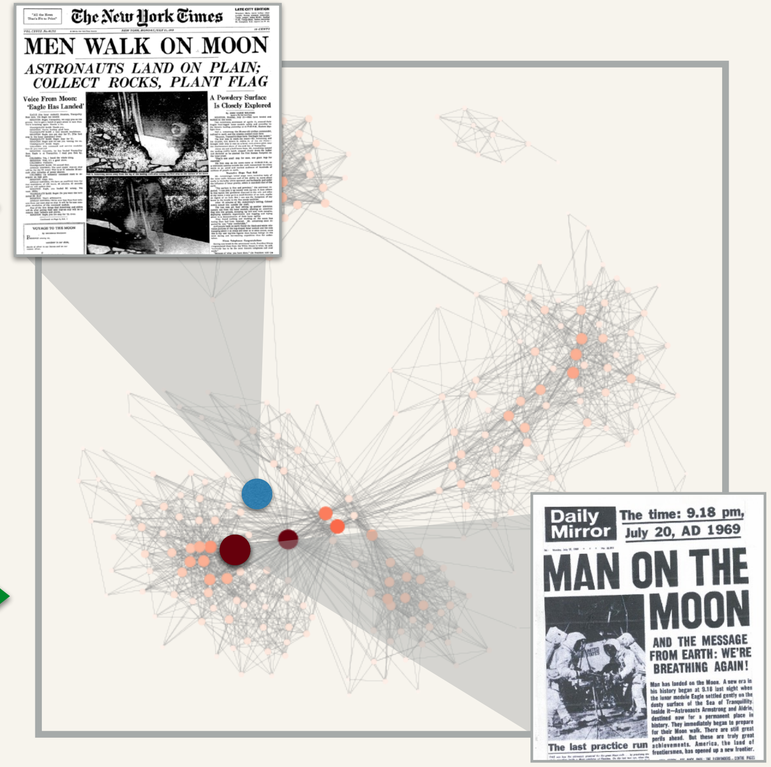
The Washington Post

NBC NEWS

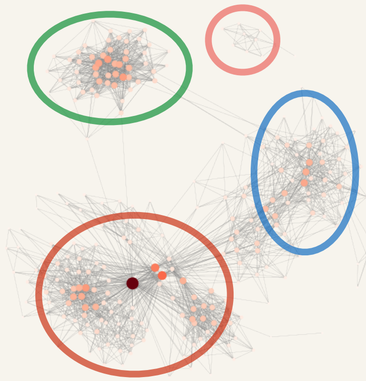
REUTERS

Query Stream

- Fetch similar vertices
- Clustering Coefficients
- Fetch vertex u's neighbors
- Centrality Ranking



Clustering



Triangle Counting

Connected Components

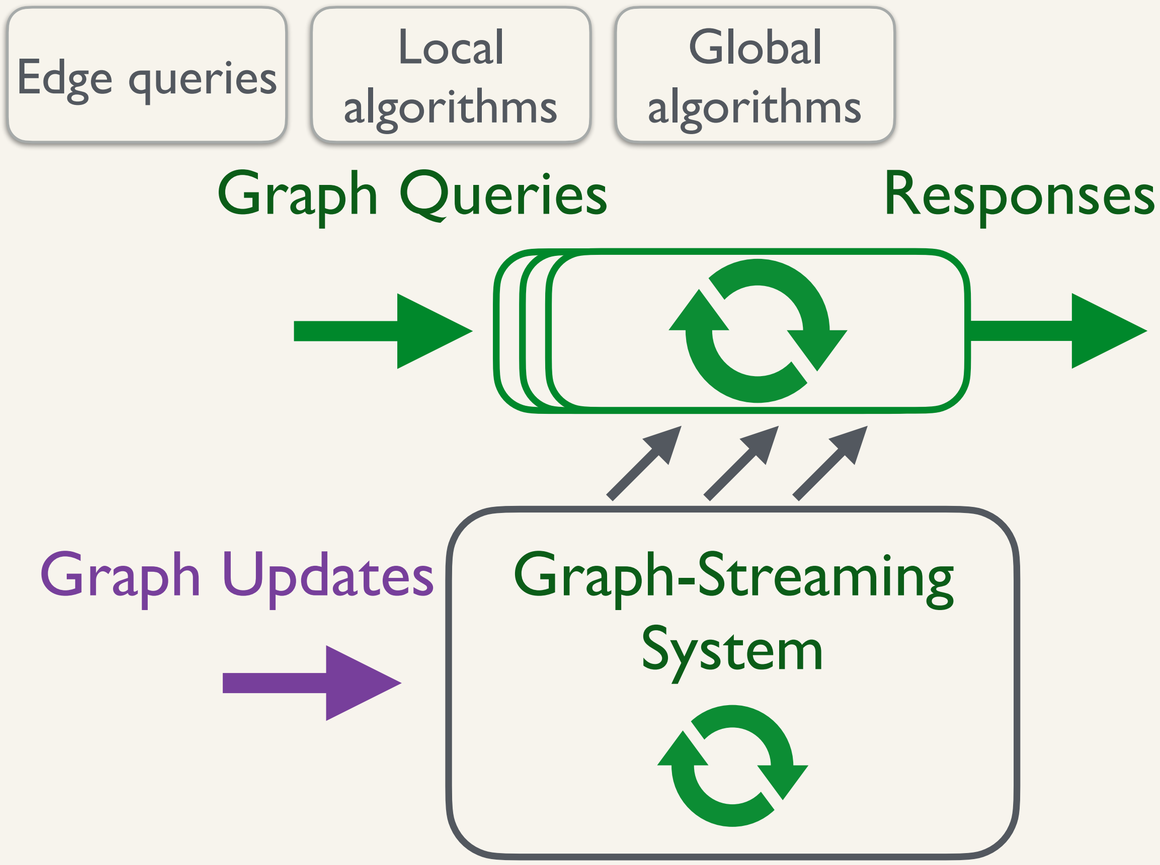
Streaming Graph Processing

Update the graph (in parallel);
Execute arbitrary queries on snapshots.

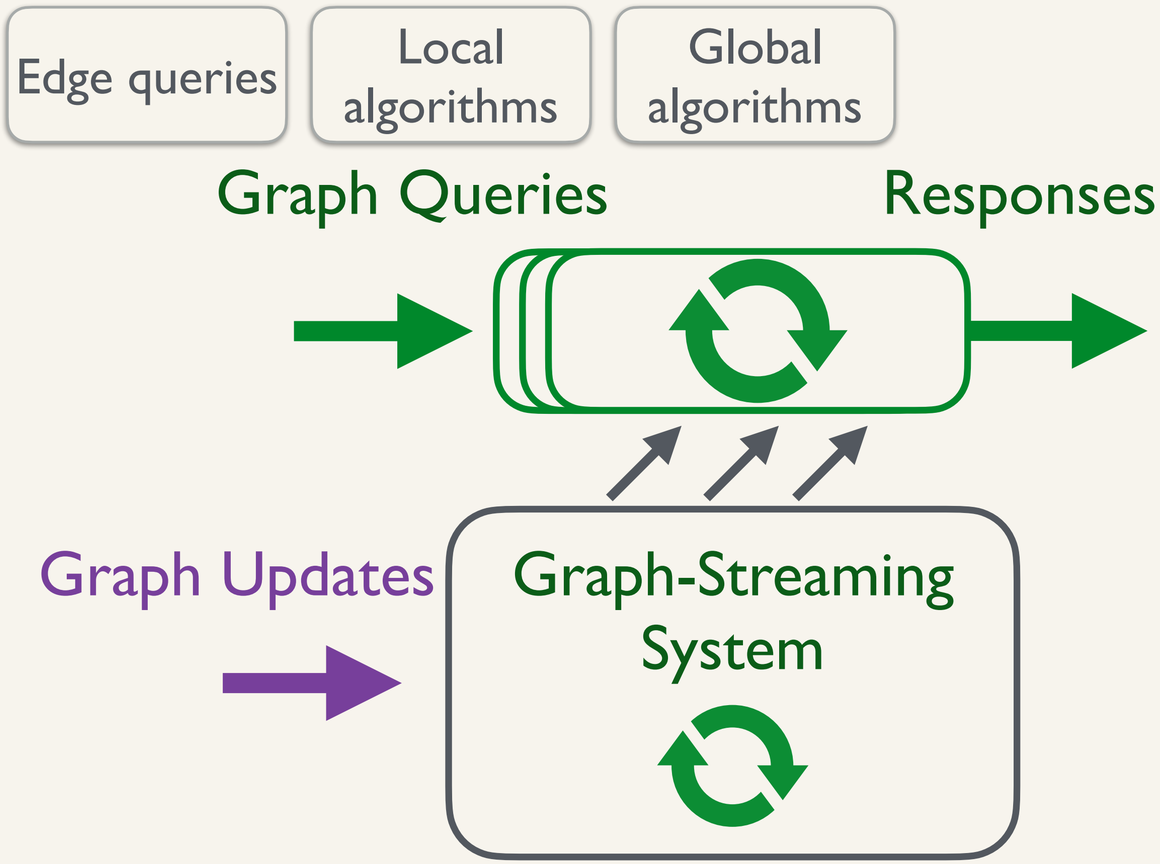
Batch-Dynamic Graph Processing

Pre-determined queries;
Process updates faster than recomputation.

Streaming Graph Processing

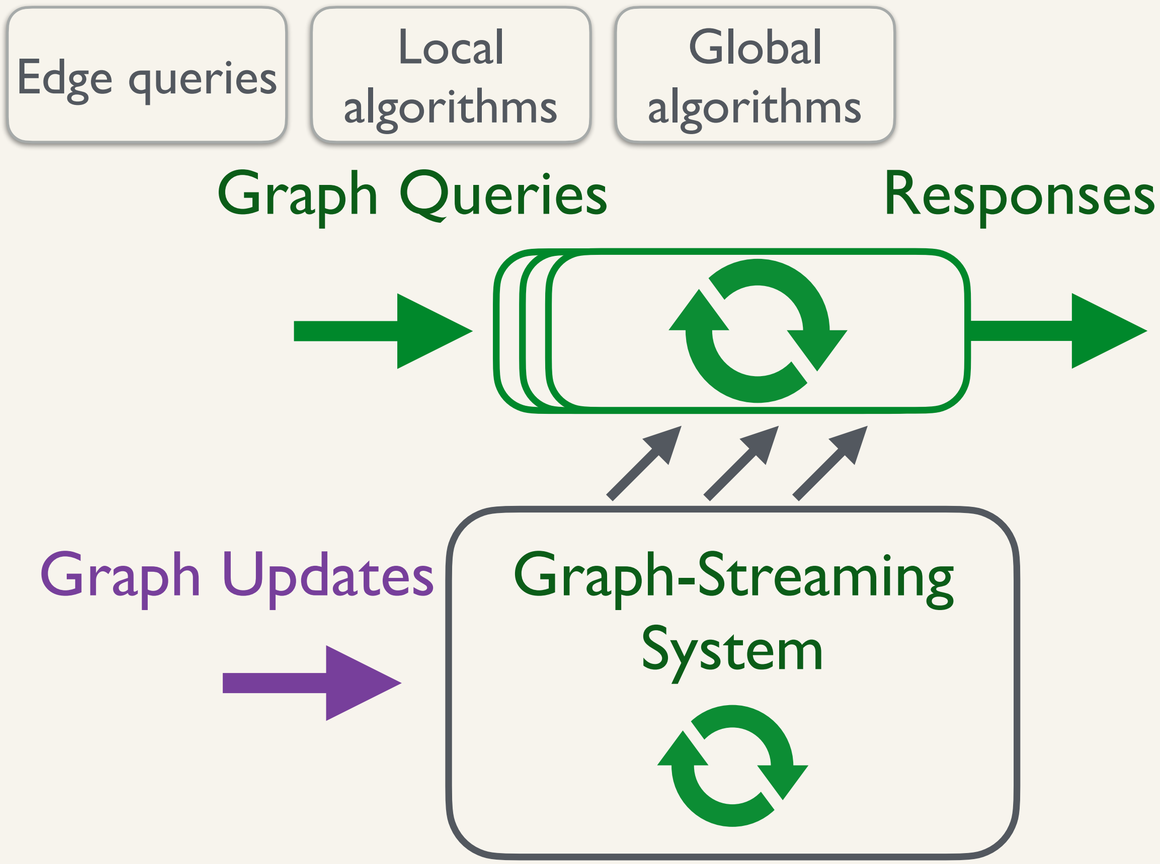


Streaming Graph Processing



Goal: low-latency for both updates and queries arriving concurrently to the system

Streaming Graph Processing



Single-version

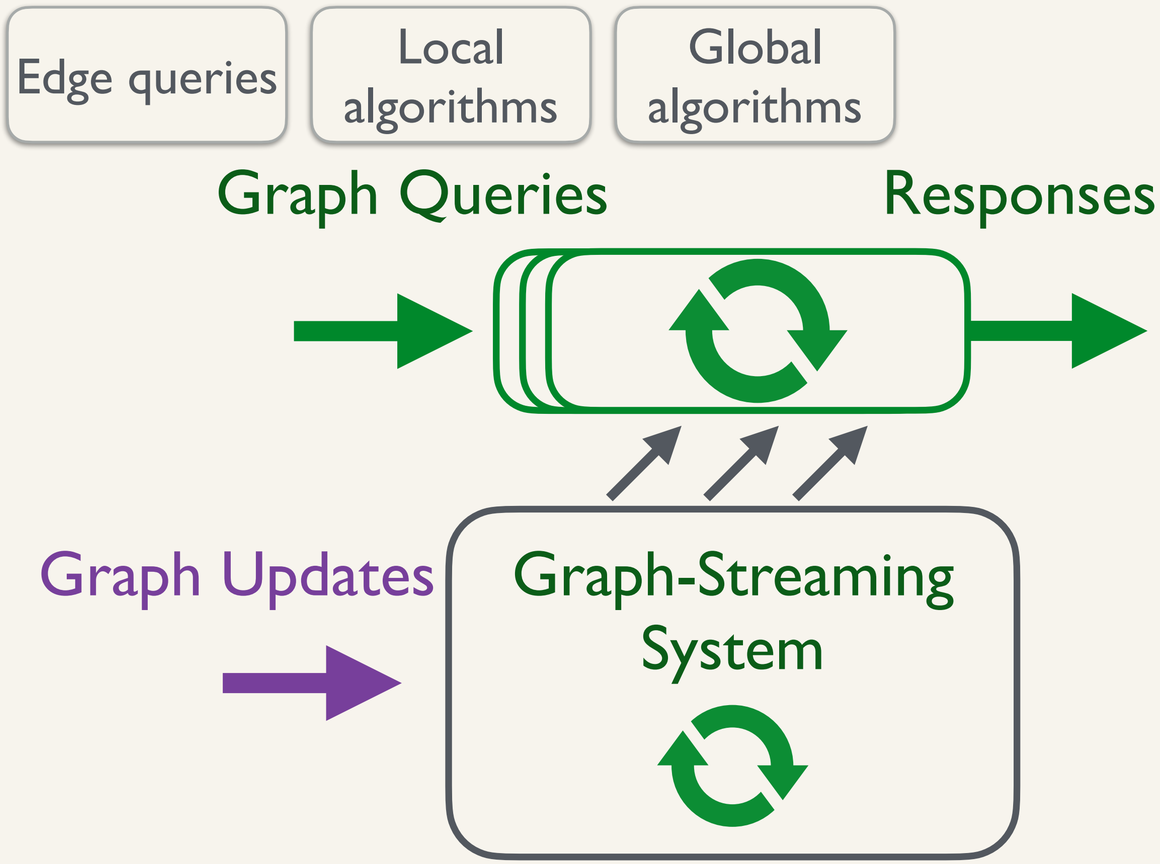
STINGER [EMRB'12]

cuSTINGER [GB'16]

Kickstarter [VGX'17]

Goal: low-latency for both updates and queries arriving concurrently to the system

Streaming Graph Processing



Single-version

STINGER [EMRB'12]

cuSTINGER [GB'16]

Kickstarter [VGX'17]

Multi-version (snapshot-based)

Kineograph [CHKMW+'12]

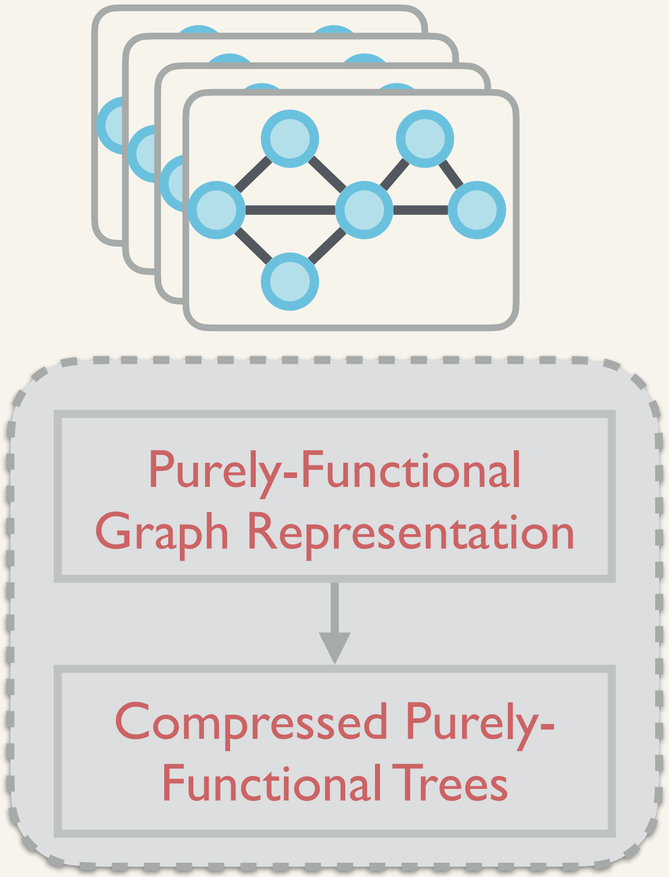
LLAMA [MMMS'15]

Goal: low-latency for both updates and queries arriving concurrently to the system

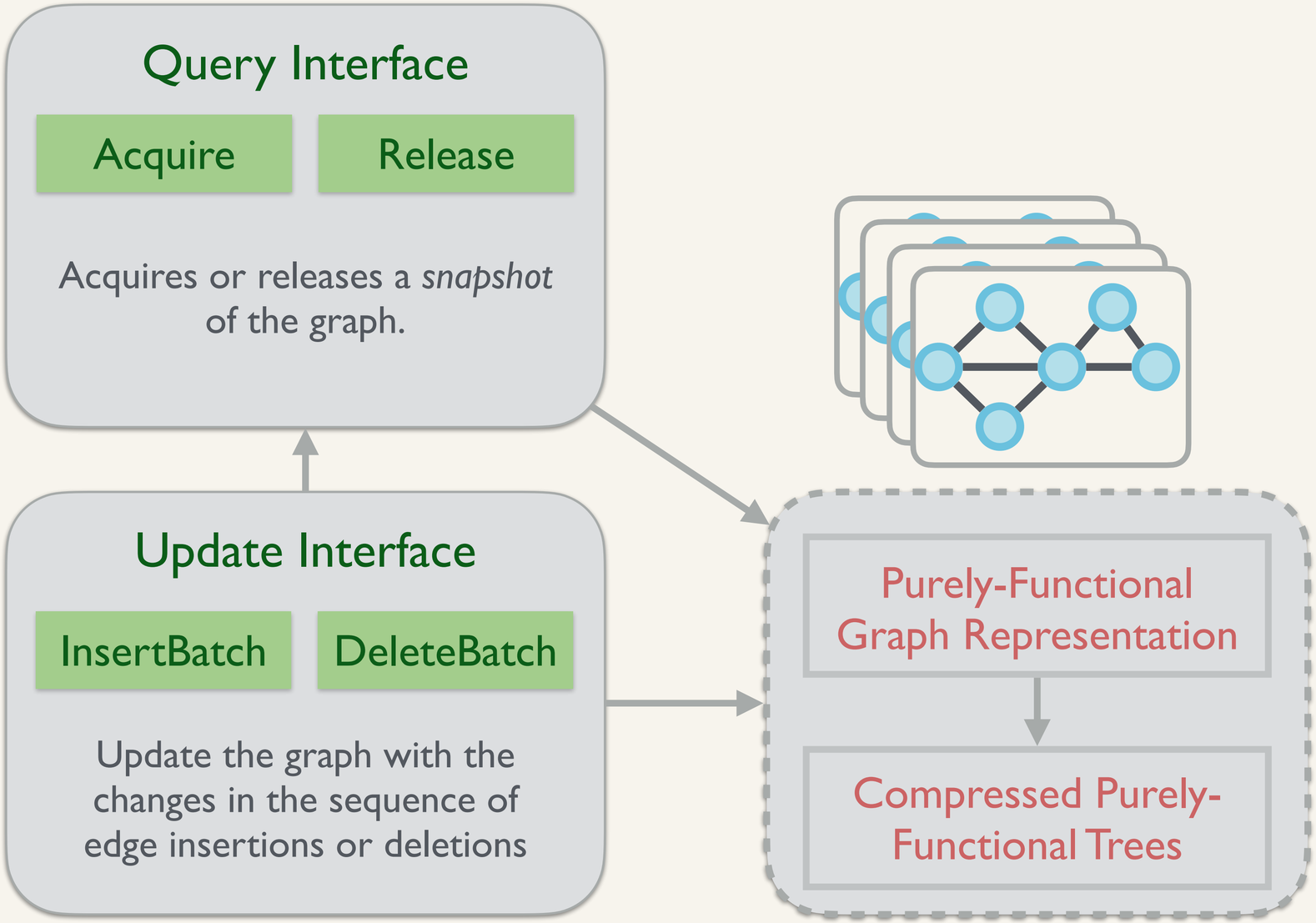
Low-Latency Graph Streaming using Compressed Purely-Functional Trees [DBS'19]

Can we design a system that can
compactly represent and
concurrently update and query
the largest real-world graphs?

Aspen: A Low-Latency Graph Streaming System



Aspen: A Low-Latency Graph Streaming System



Aspen: A Low-Latency Graph Streaming System

Graph Algorithm Interface

Snapshots *implement the GBBS interface*, making it possible to run parallel graph algorithms from GBBS on snapshots in Aspen.

Breadth-First Search

Maximal Independent Set

Parallel Connectivity

And many others

Query Interface

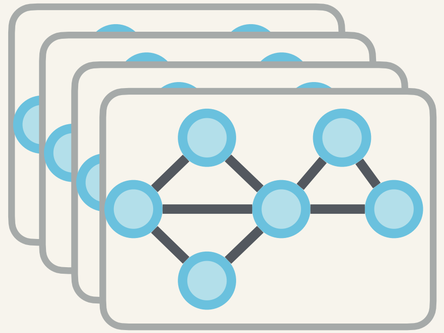
Acquire | Release

Acquires or releases a *snapshot* of the graph.

Update Interface

InsertBatch | DeleteBatch

Update the graph with the changes in the sequence of edge insertions or deletions



Purely-Functional Graph Representation

Compressed Purely-Functional Trees

Aspen: A Low-Latency Graph Streaming System

Graph Algorithm Interface

Snapshots *implement the GBBS interface*, making it possible to run parallel graph algorithms from GBBS on snapshots in Aspen

Breadth-First Search

Maximal Independent Set

Parallel Connectivity

And many others

Query Interface

Acquire

Release

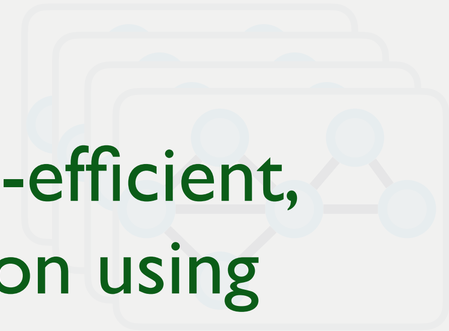
Acquires or releases a *snapshot* of the graph

Update Interface

InsertBatch

DeleteBatch

Update the graph with the changes in the sequence of edge insertions or deletions

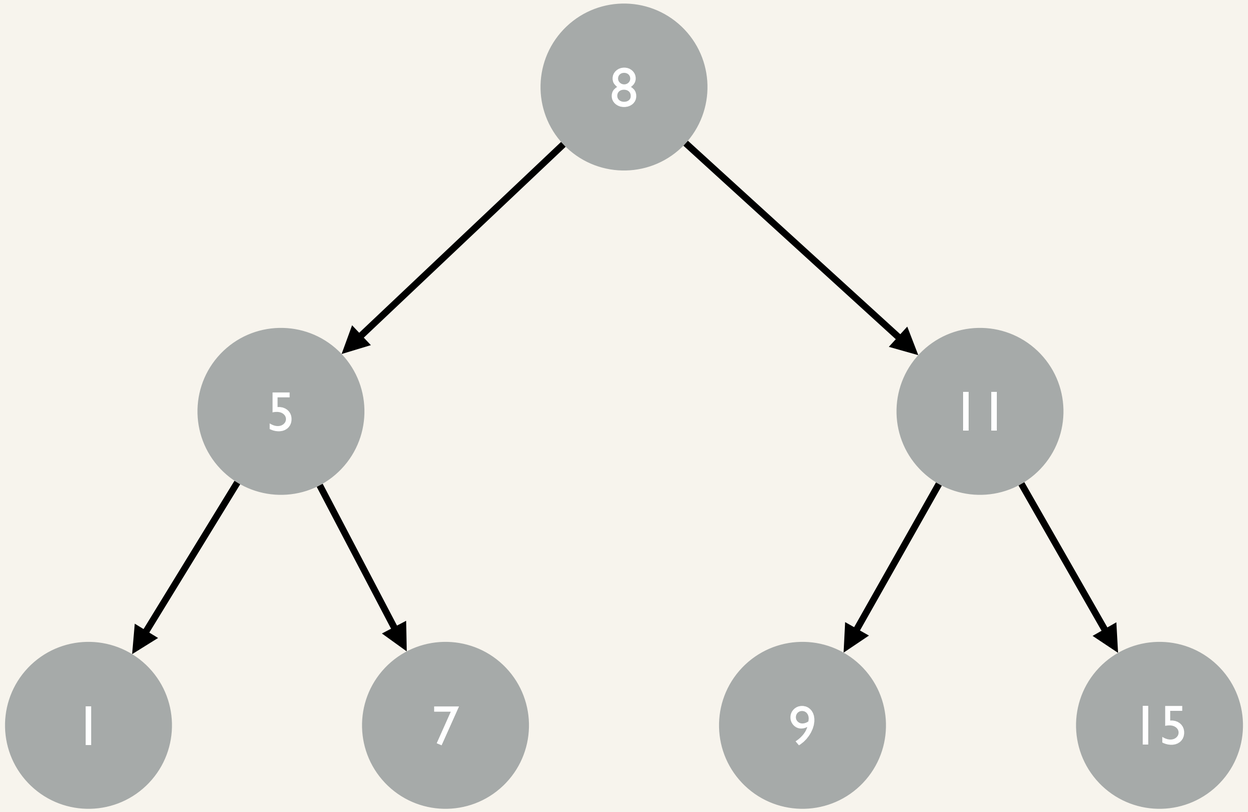


Purely-Functional Graph Representation

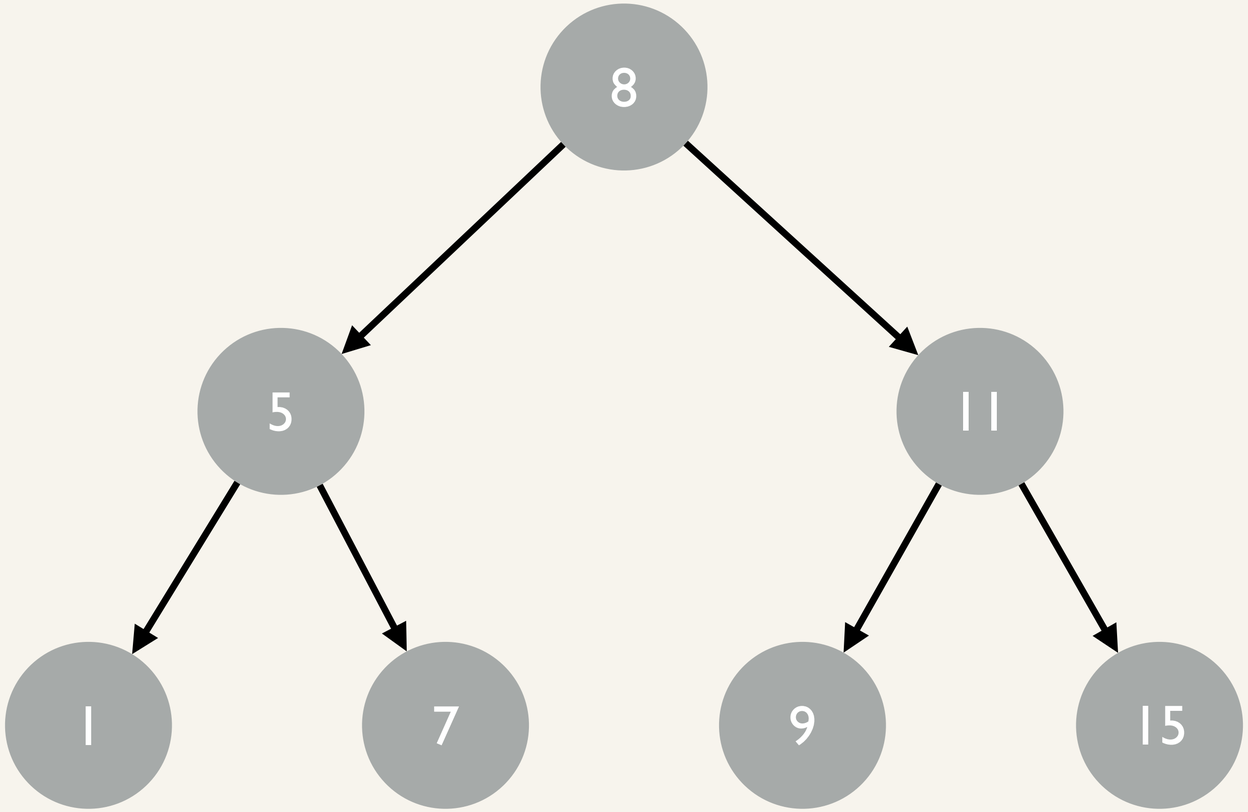
Compressed Purely-Functional Trees

Main contribution: designing a scalable, space-efficient, and efficiently-updatable graph representation using compressed purely-functional trees

Purely-Functional Trees

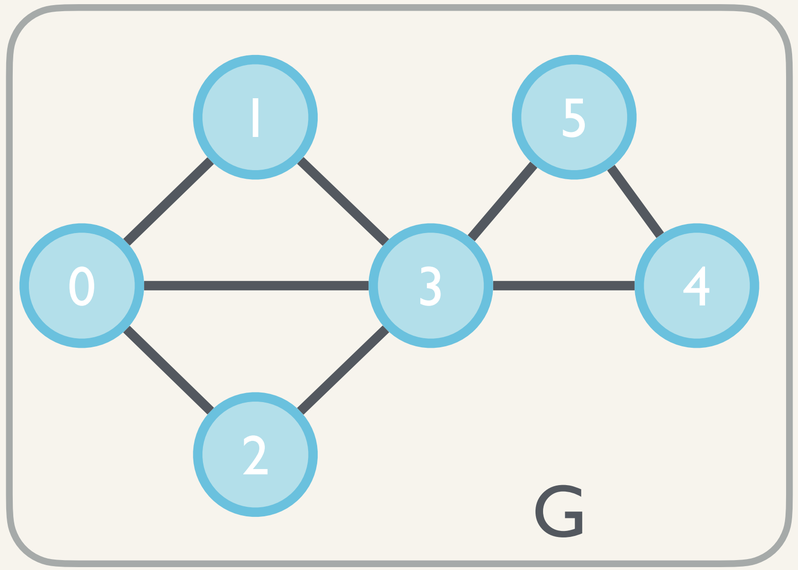


Purely-Functional Trees

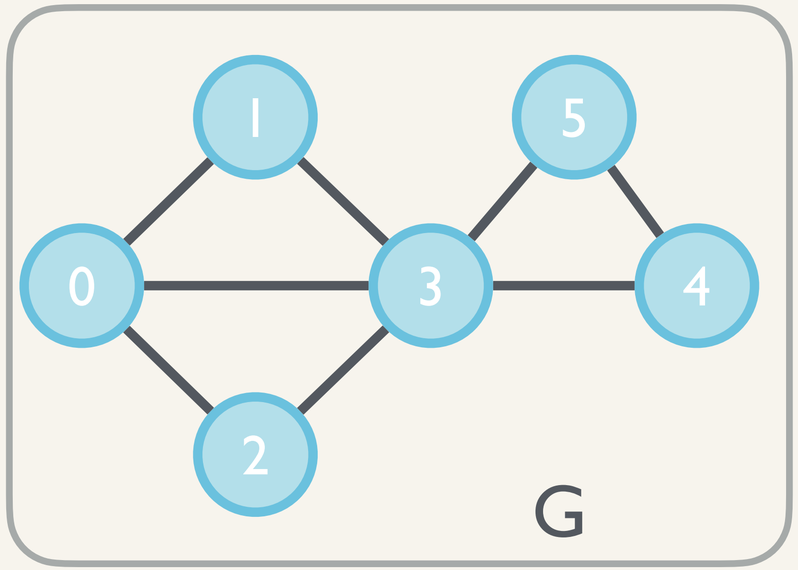


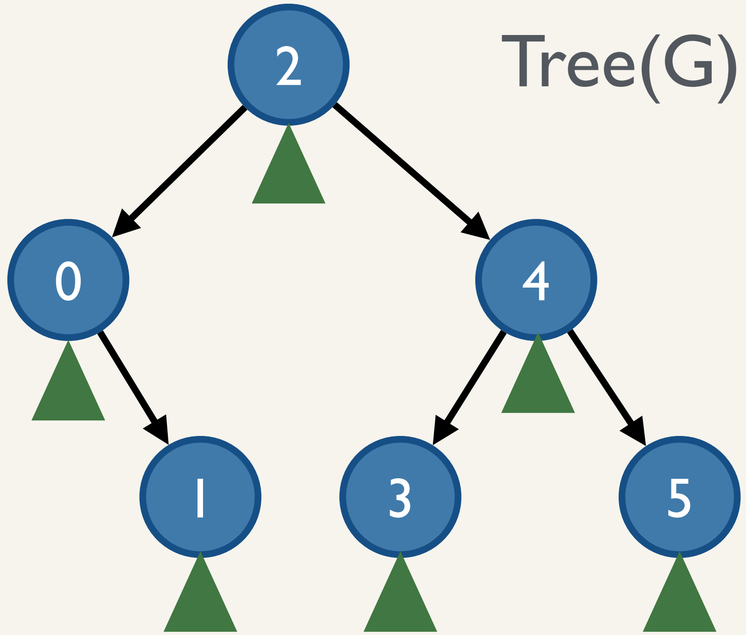
Red-black, AVL, or weight-balanced trees

Representing Graphs using Trees

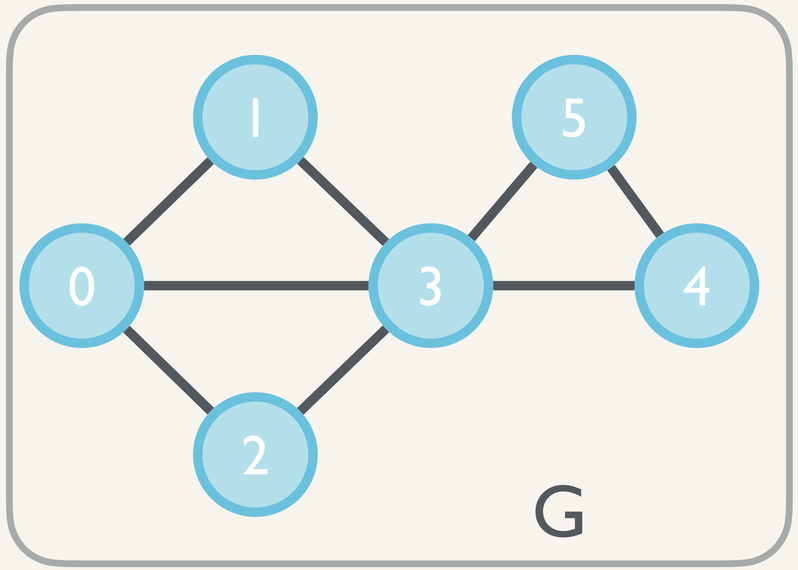


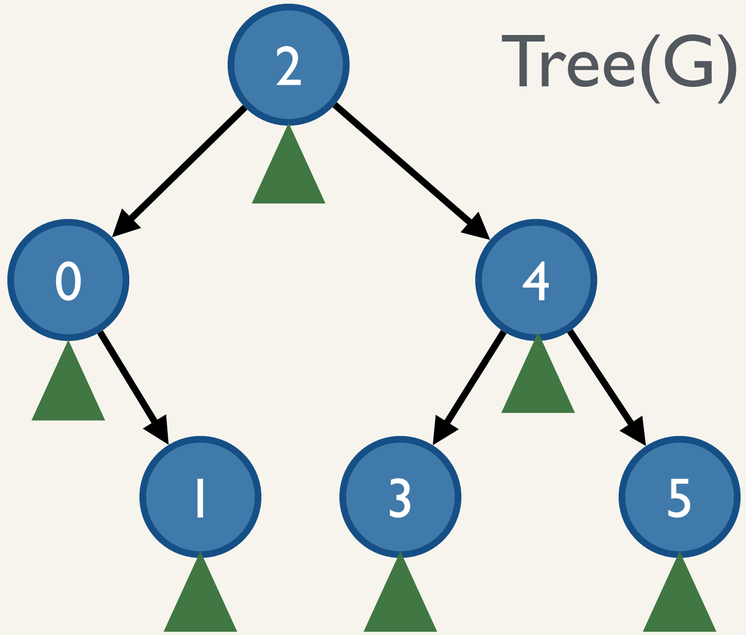
Representing Graphs using Trees



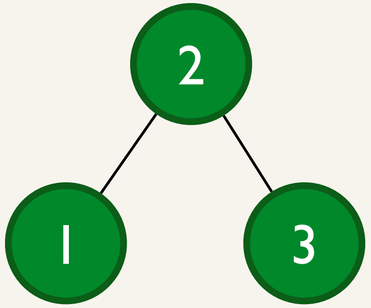


Representing Graphs using Trees

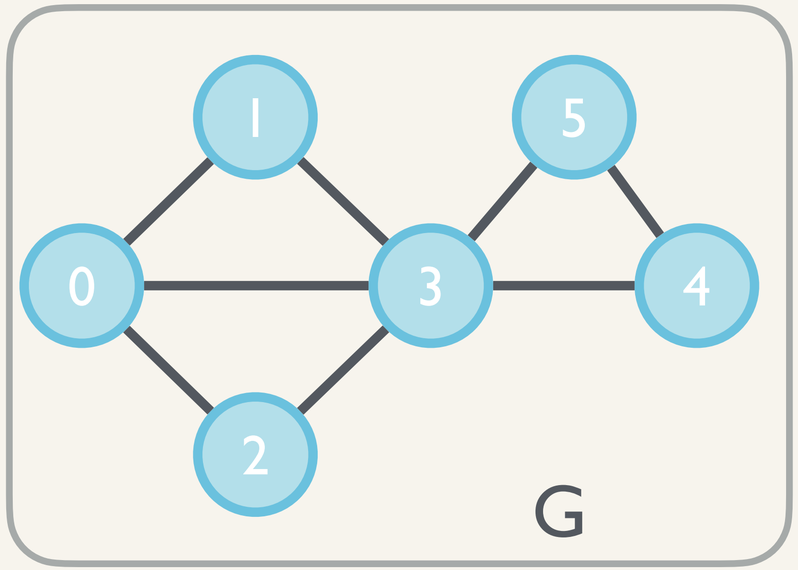


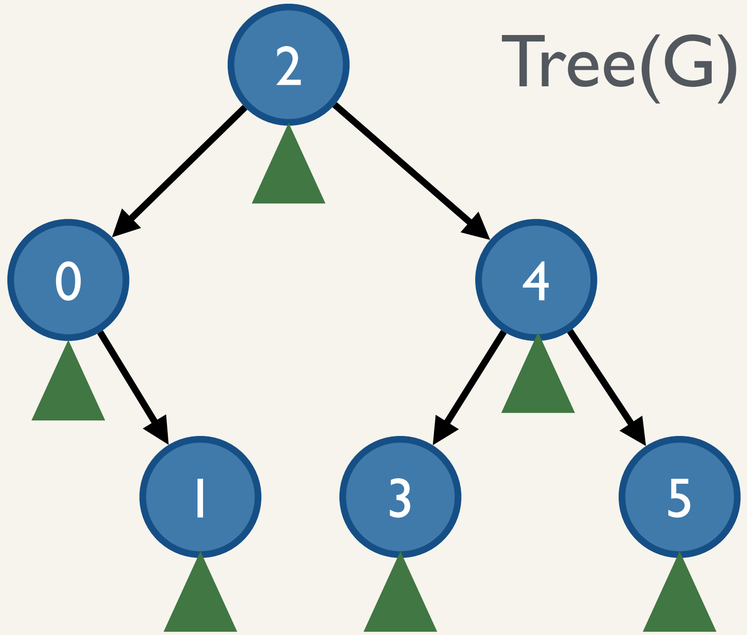


Vertex 0's
Edge Tree

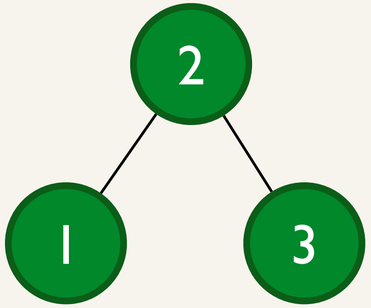


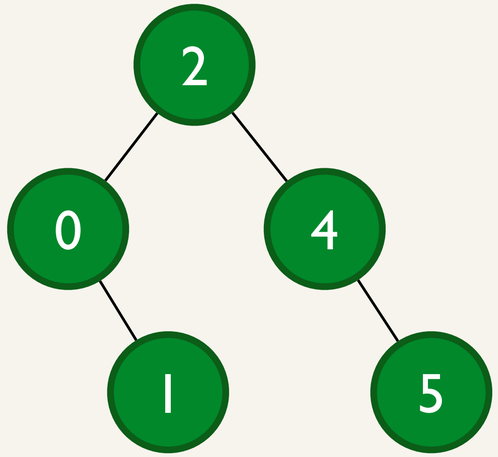
Representing Graphs using Trees





Vertex 0's
Edge Tree





Vertex 3's
Edge Tree

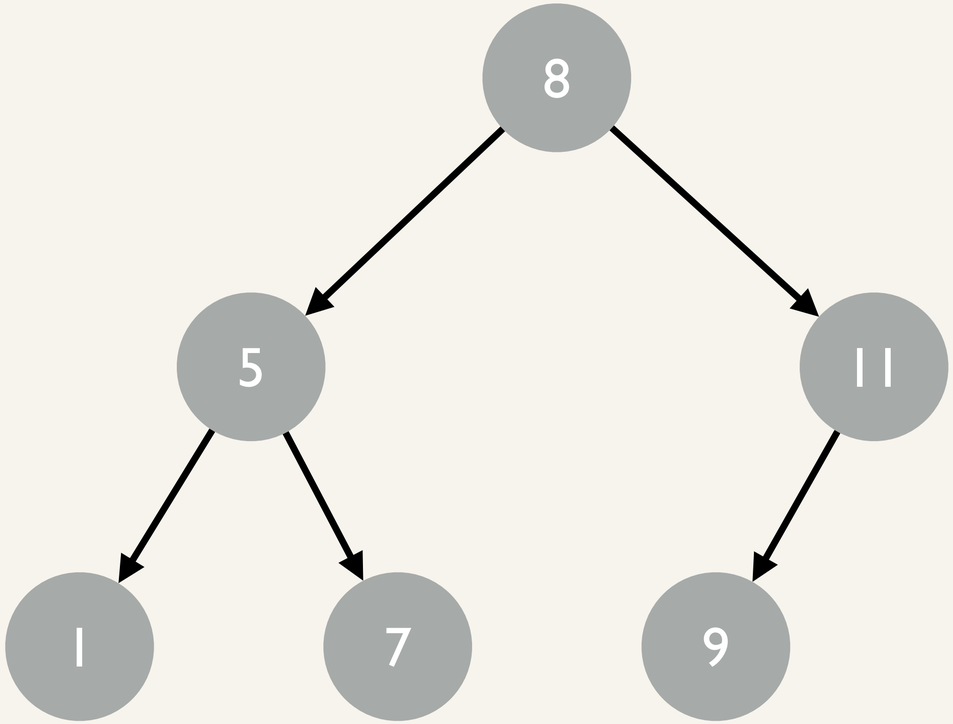
Trees enable Simple Snapshots

Trees enable Simple Snapshots

A snapshot is just a tree root

Trees enable Simple Snapshots

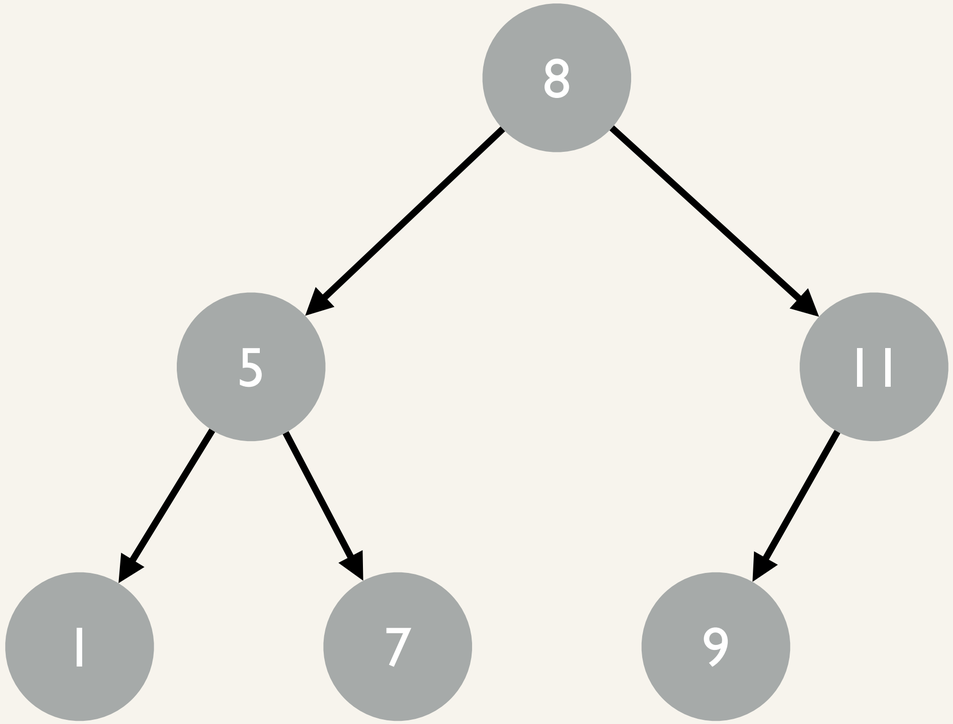
A snapshot is just a tree root



Trees enable Simple Snapshots

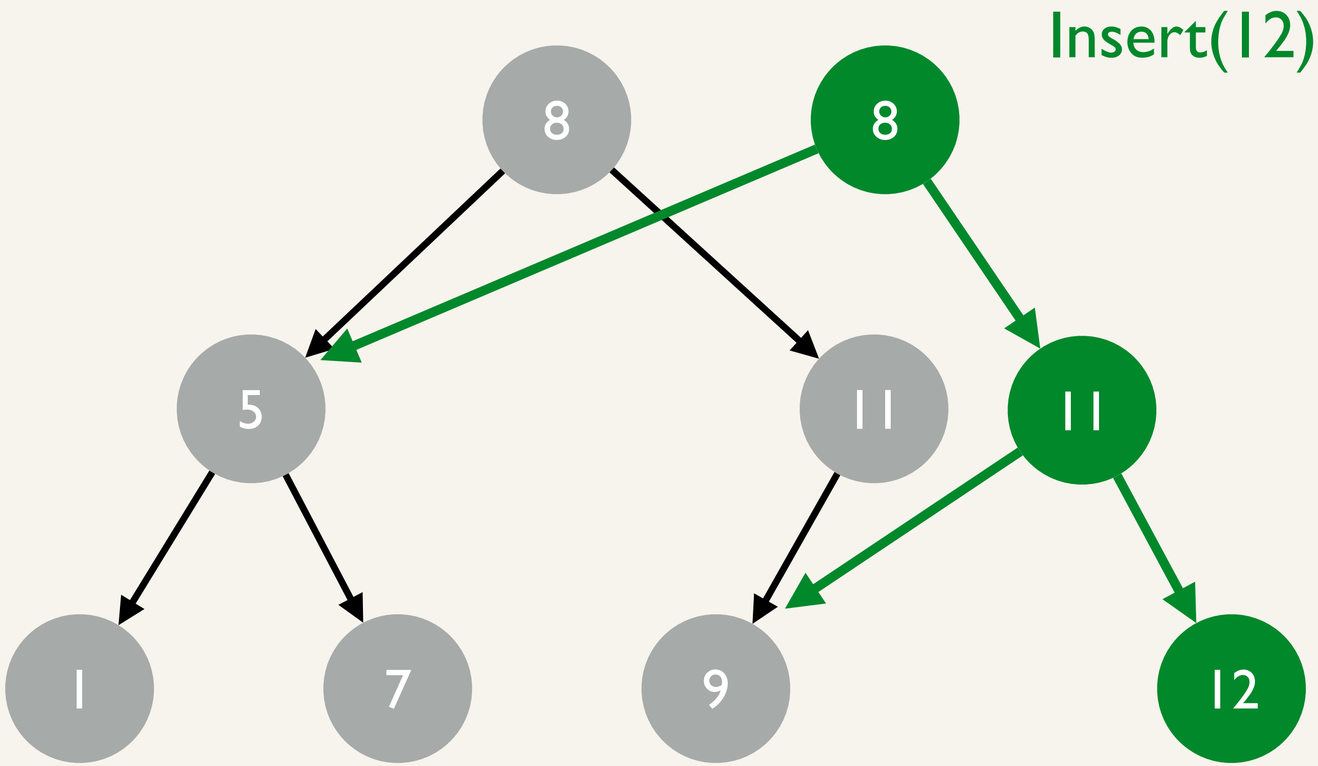
A snapshot is just a tree root

Insert(12)



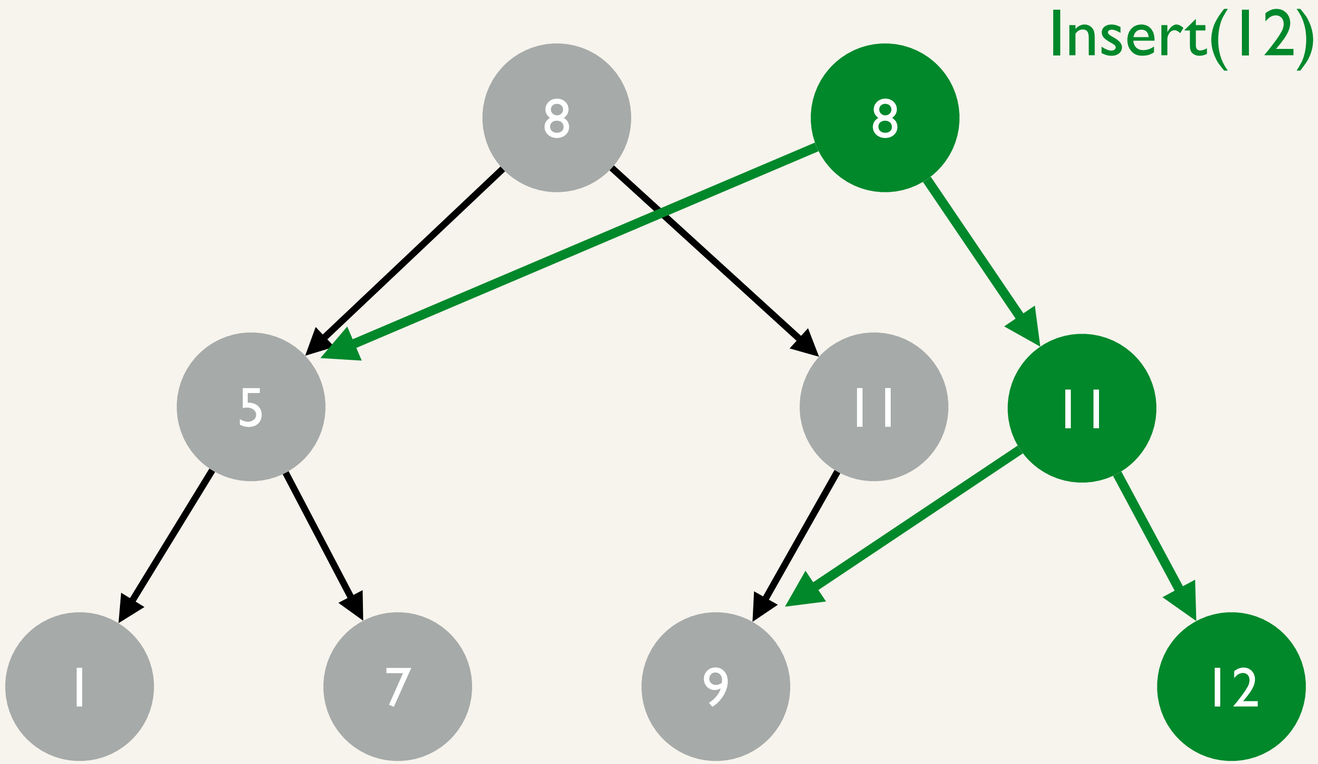
Trees enable Simple Snapshots

A snapshot is just a tree root



Trees enable Simple Snapshots

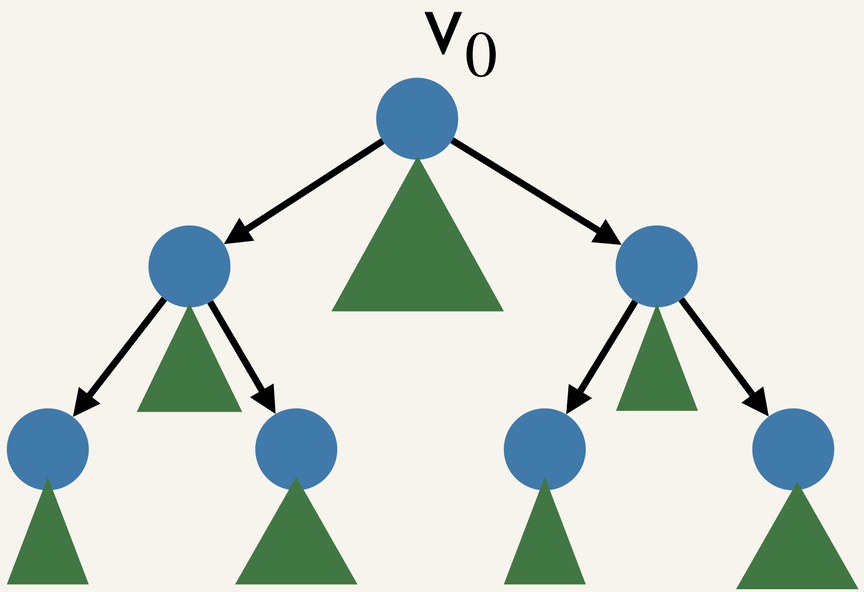
A snapshot is just a tree root



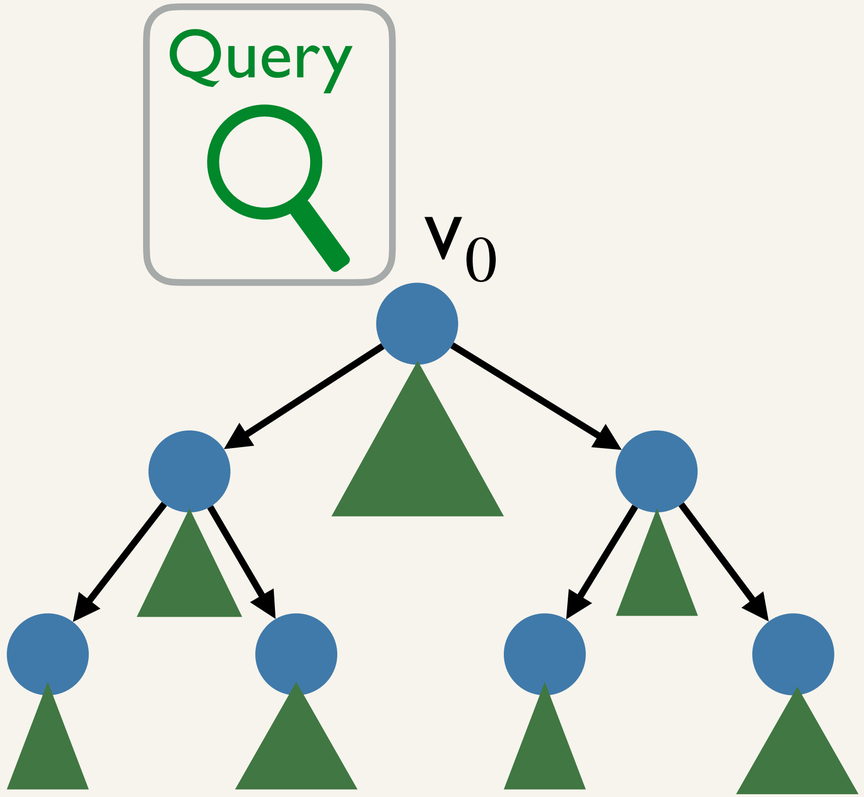
Algorithms generalize to handle *batches* of updates in low work/depth [BFS'16]

Purely-Functional Trees are Safe for Concurrency

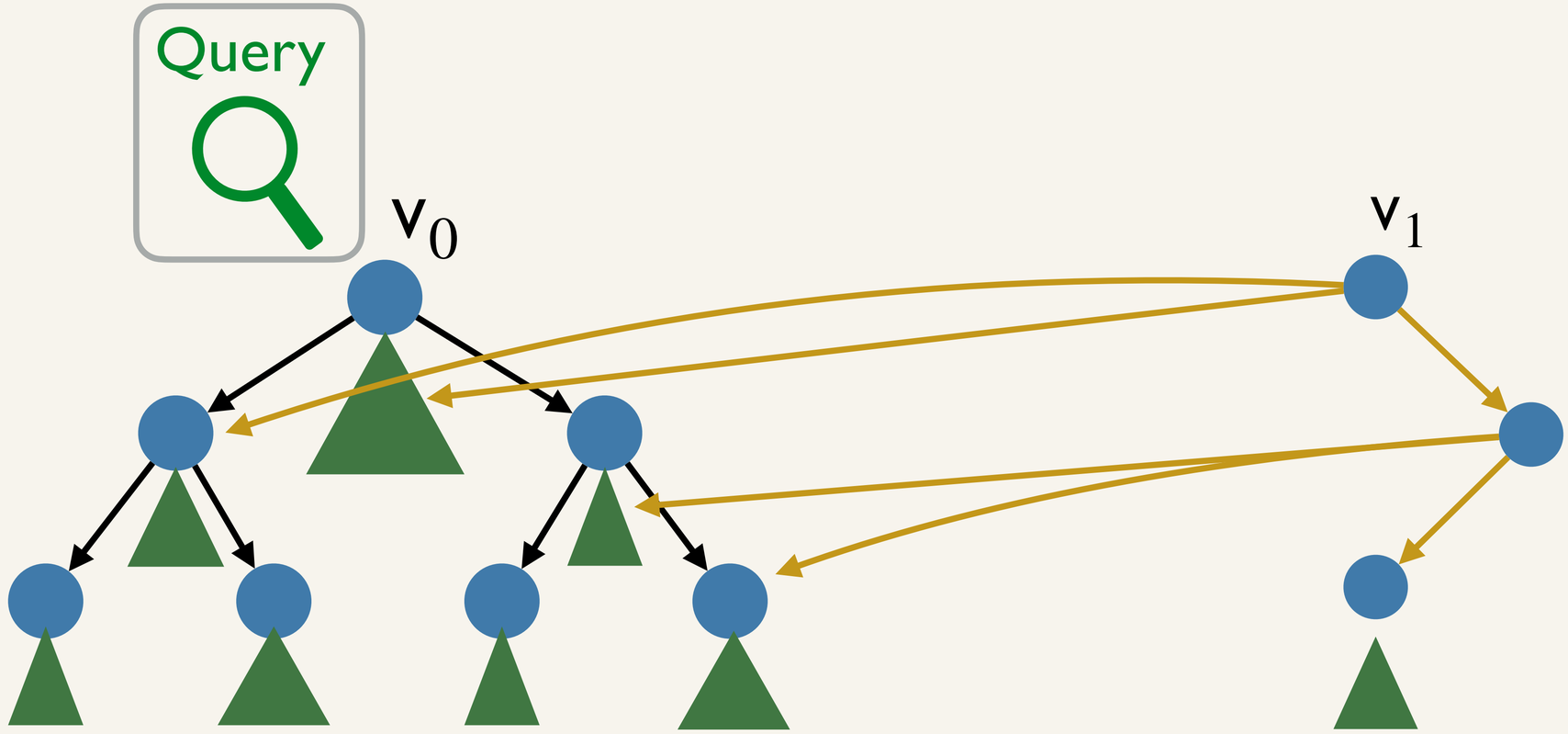
Purely-Functional Trees are Safe for Concurrency



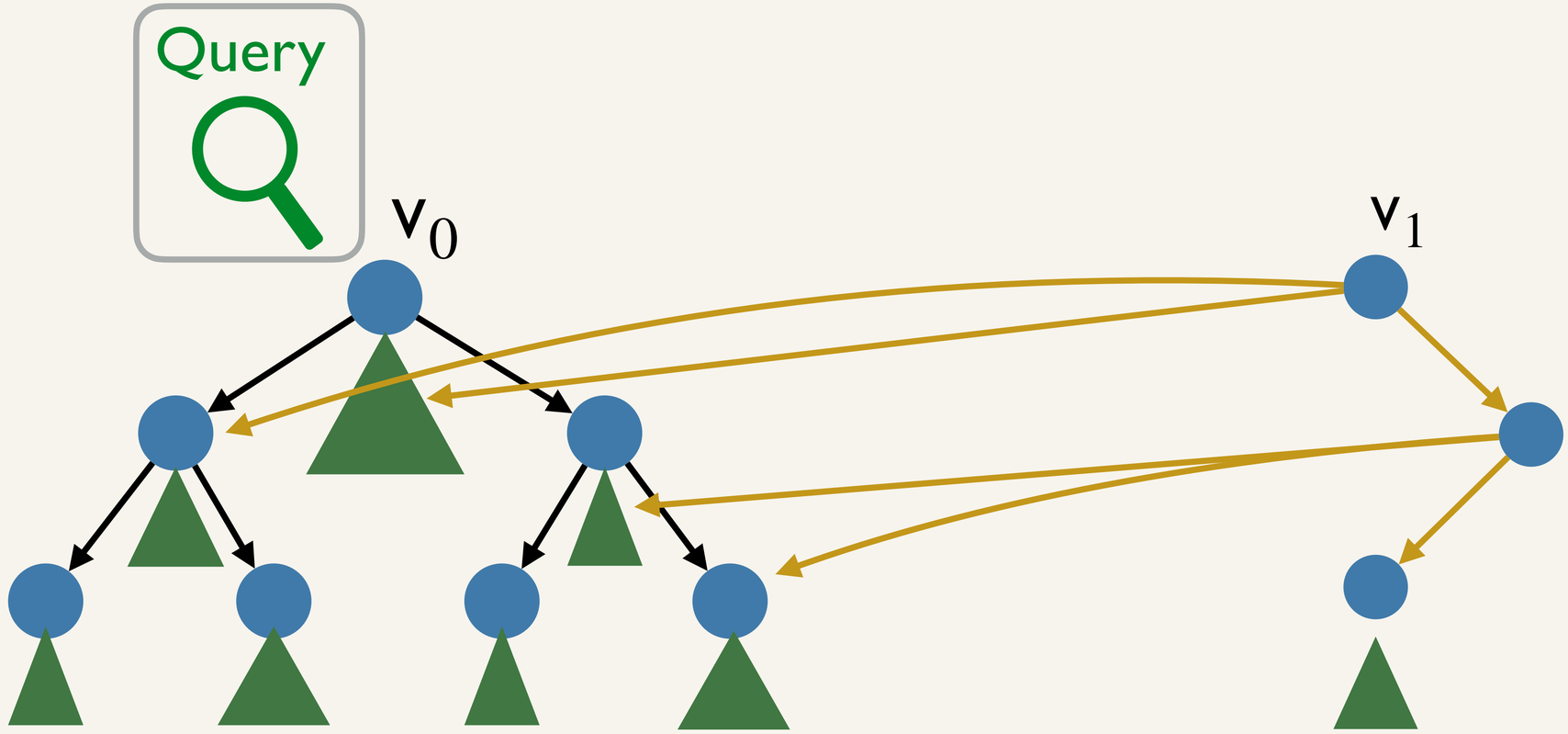
Purely-Functional Trees are Safe for Concurrency



Purely-Functional Trees are Safe for Concurrency



Purely-Functional Trees are Safe for Concurrency



Queries are serialized once they
acquire a tree root

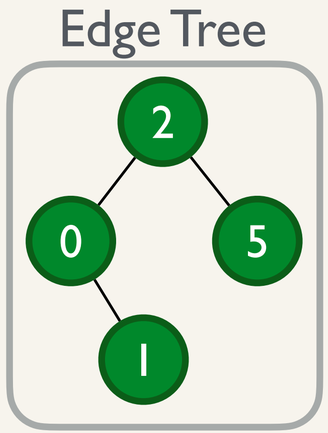
Challenges

Challenges

Poor Cache Usage

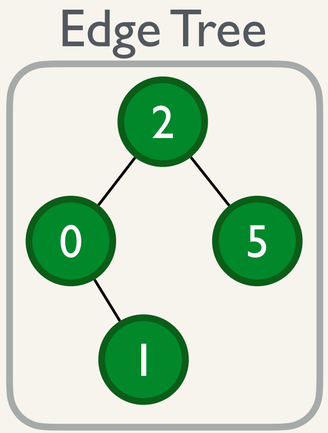
Challenges

Poor Cache Usage



Challenges

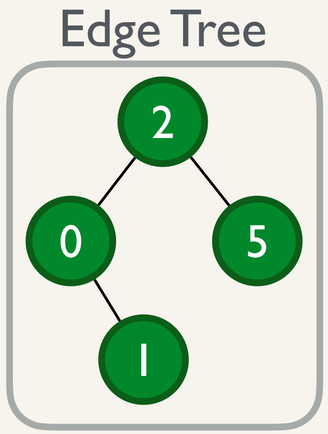
Poor Cache Usage



Space Inefficiency

Challenges

Poor Cache Usage

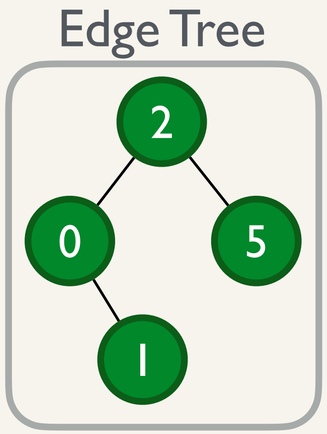


Space Inefficiency

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists

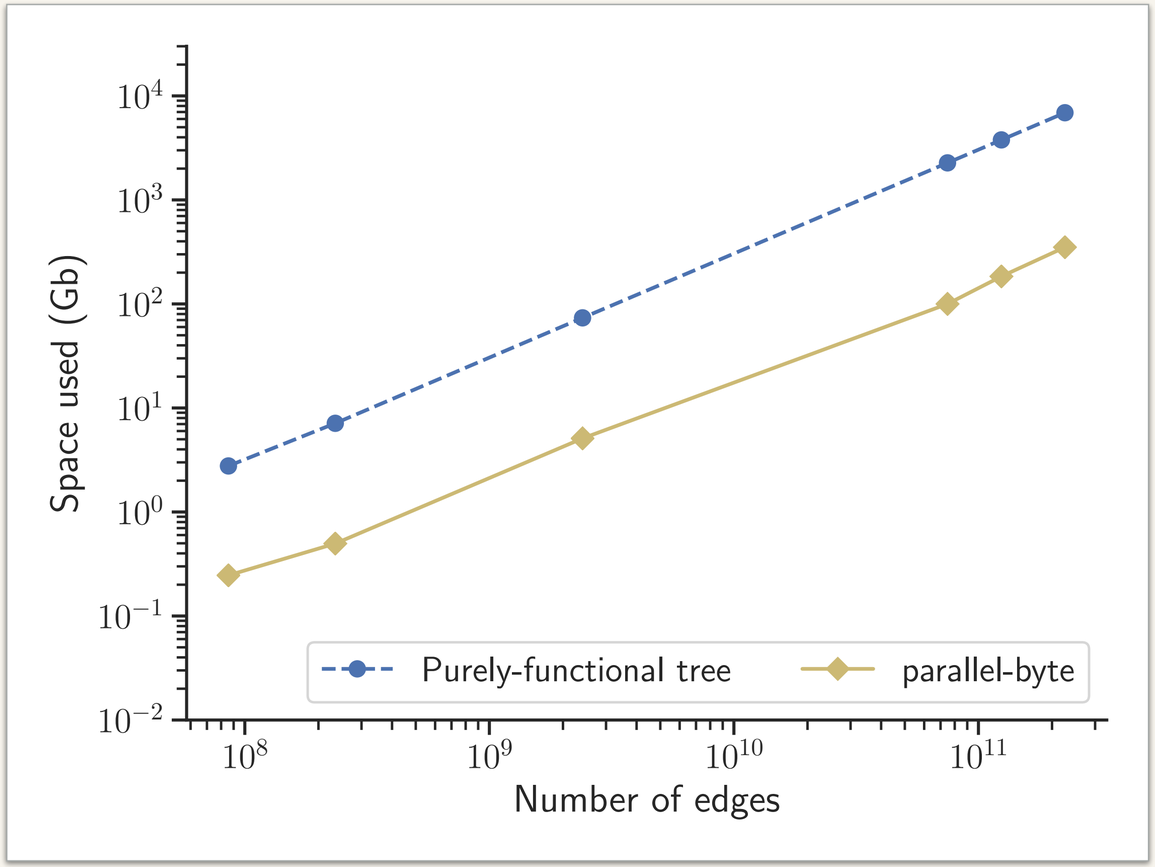
Challenges

Poor Cache Usage



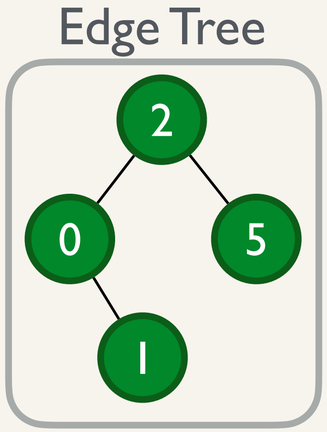
Space Inefficiency

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists



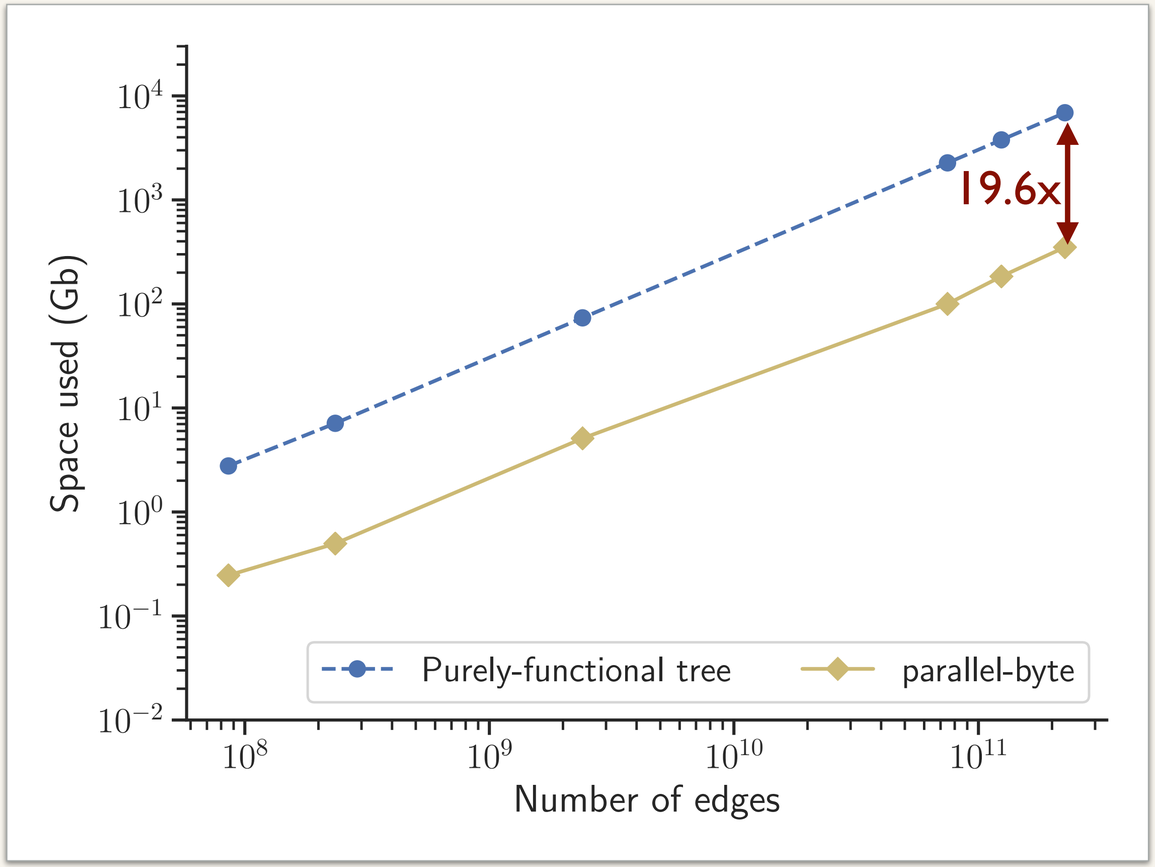
Challenges

Poor Cache Usage



Space Inefficiency

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists



Aspen using naive approach requires
7 TB of memory for WDC2012 graph

Challenges

Poor Cache Usage

Space Inefficiency

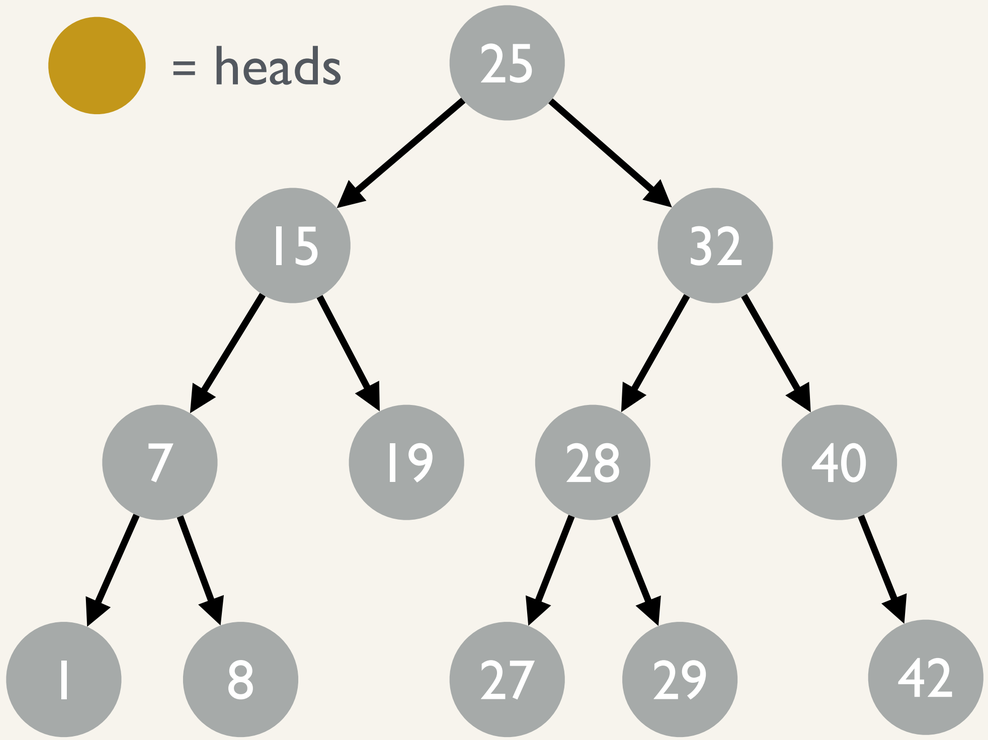
- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists

Aspen using naive approach requires
7 TB of memory for WDC2012 graph

To overcome these challenges we designed
C-trees: compressed purely-functional trees

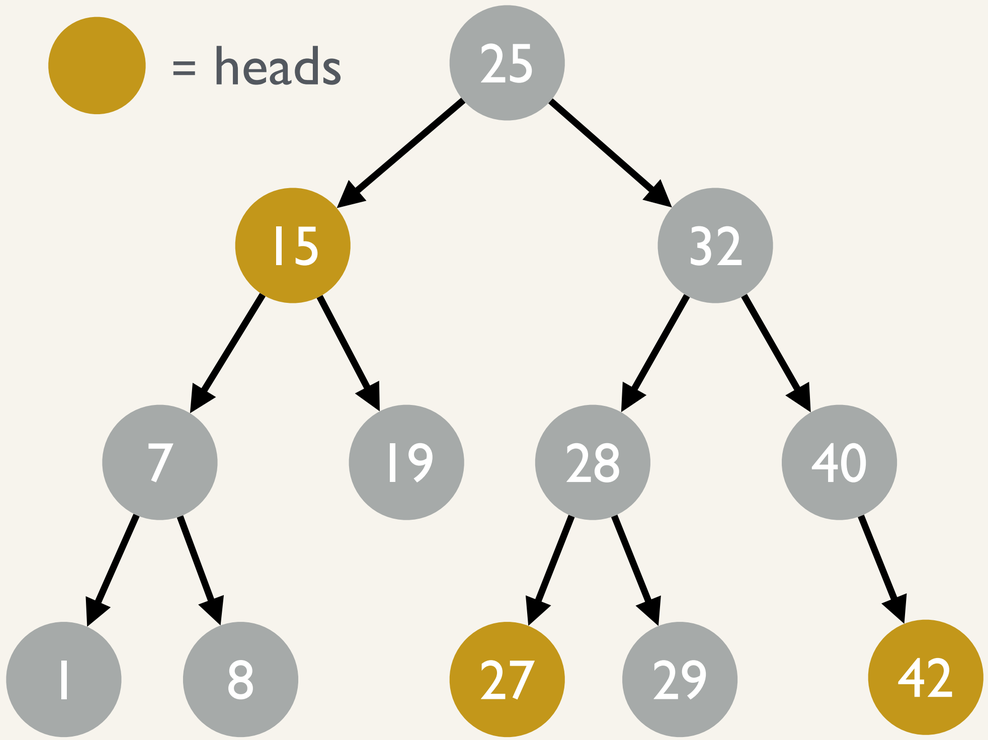
C-trees

- Chunking parameter *B*. Fix a hash function, *h*
- Select elements as *heads* with probability *1/B* using *h*



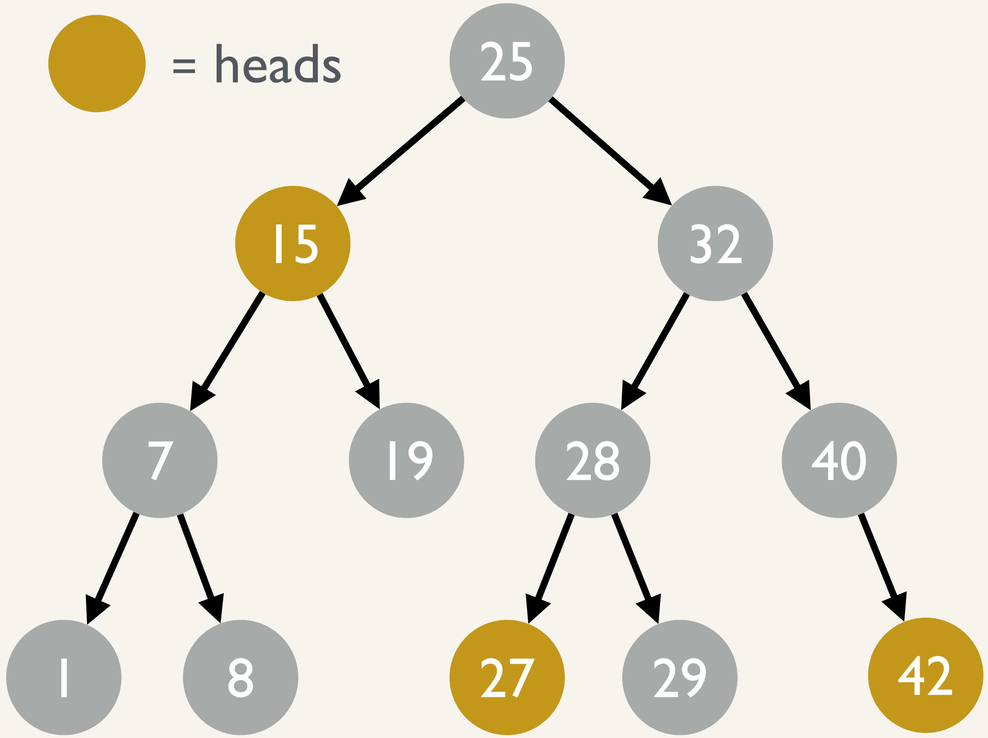
C-trees

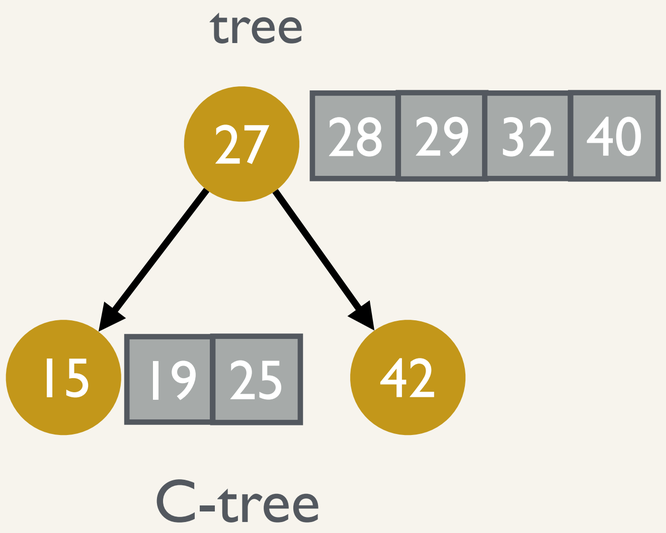
- Chunking parameter *B*. Fix a hash function, *h*
- Select elements as *heads* with probability *1/B* using *h*



C-trees

- Chunking parameter *B*. Fix a hash function, *h*
- Select elements as *heads* with probability *1/B* using *h*

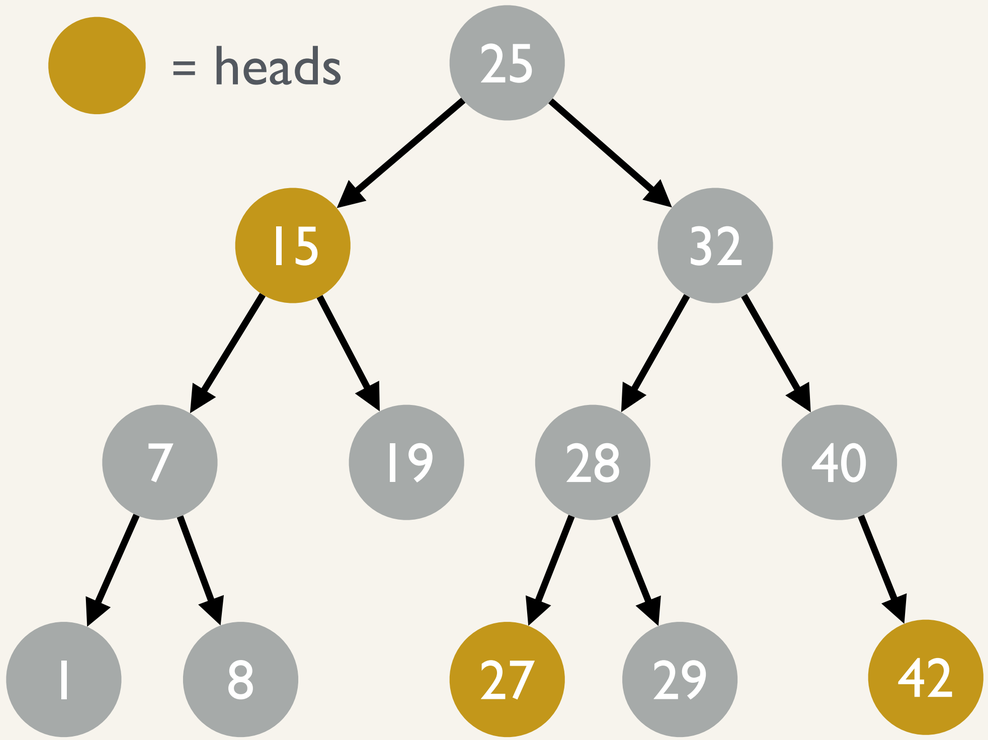


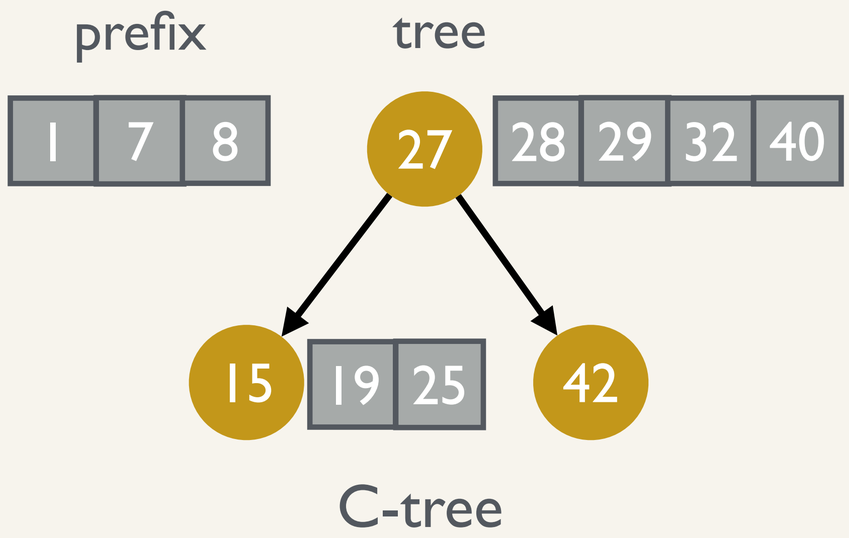


C-tree

C-trees

- Chunking parameter *B*. Fix a hash function, *h*
- Select elements as *heads* with probability *1/B* using *h*

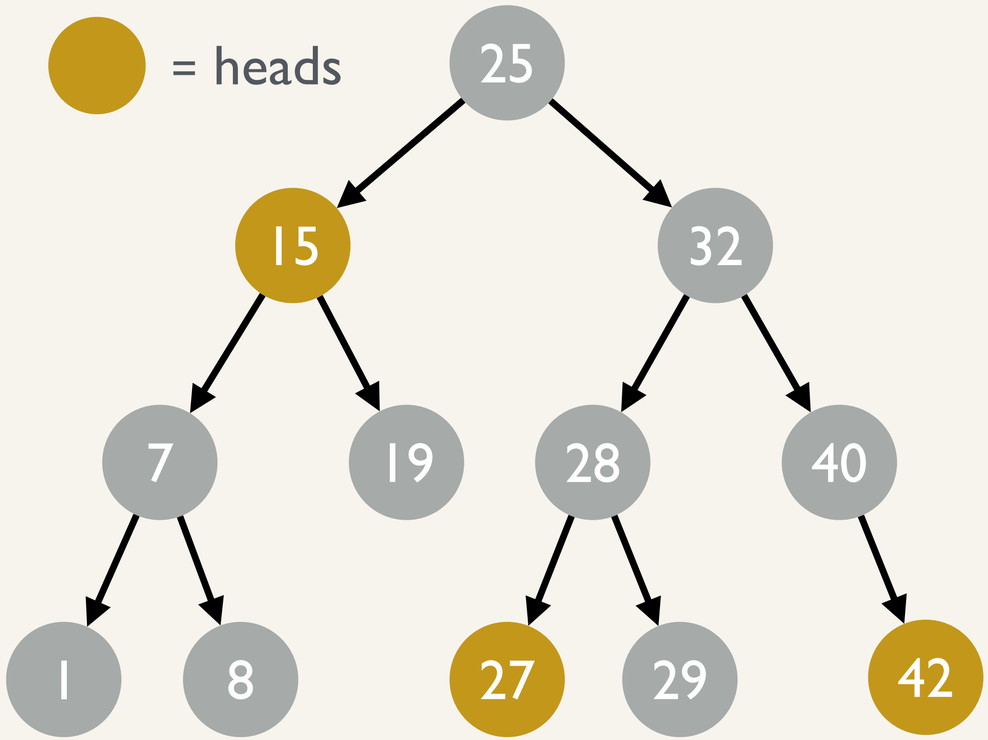




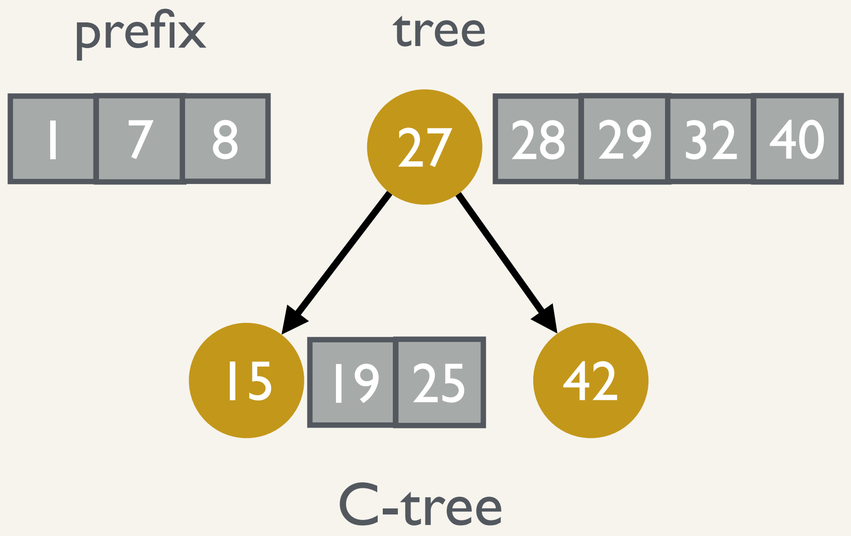
C-tree

C-trees

- Chunking parameter *B*. Fix a hash function, *h*
- Select elements as *heads* with probability *1/B* using *h*

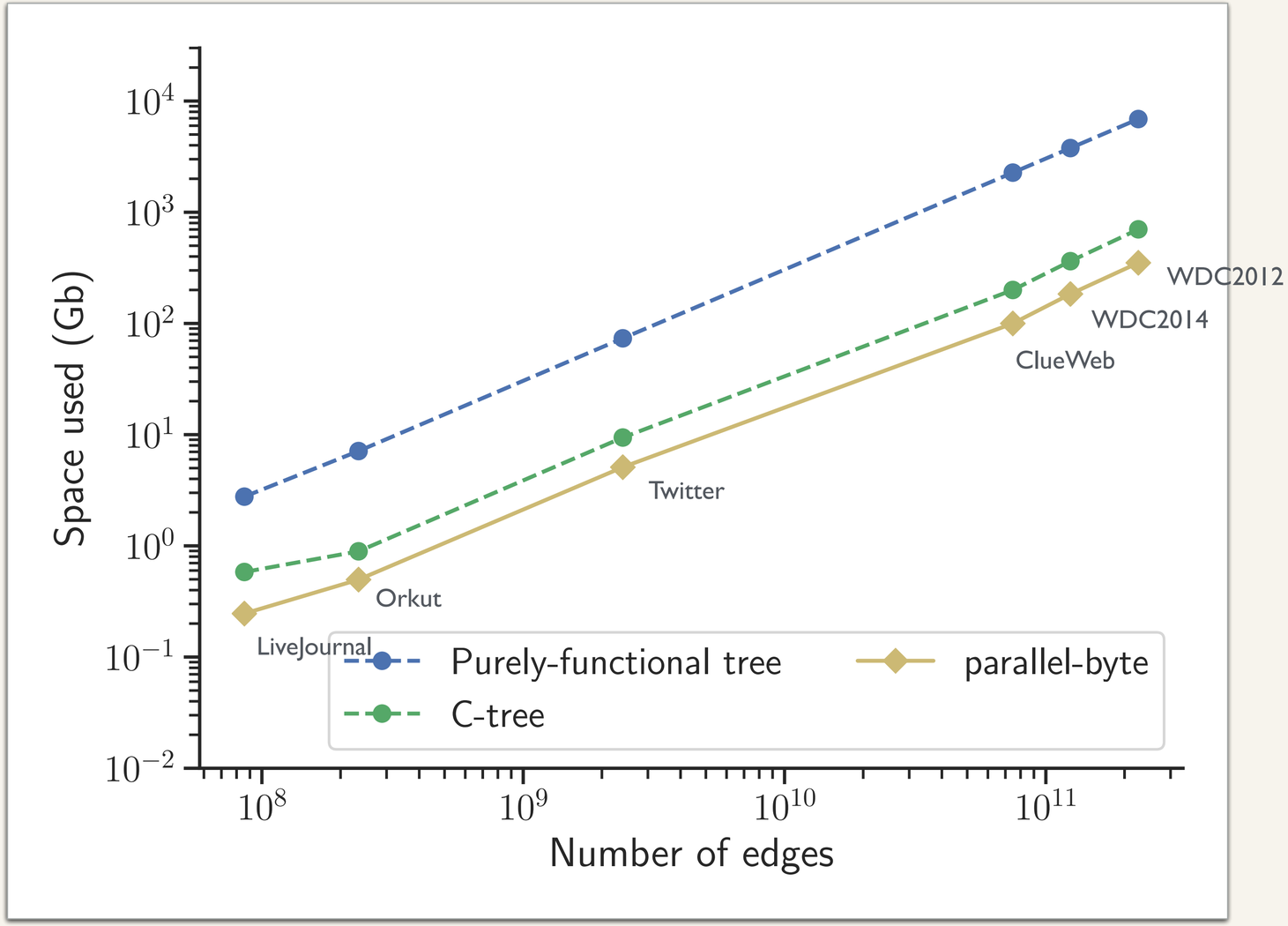


Further improve space usage for integer C-trees by difference encoding chunks

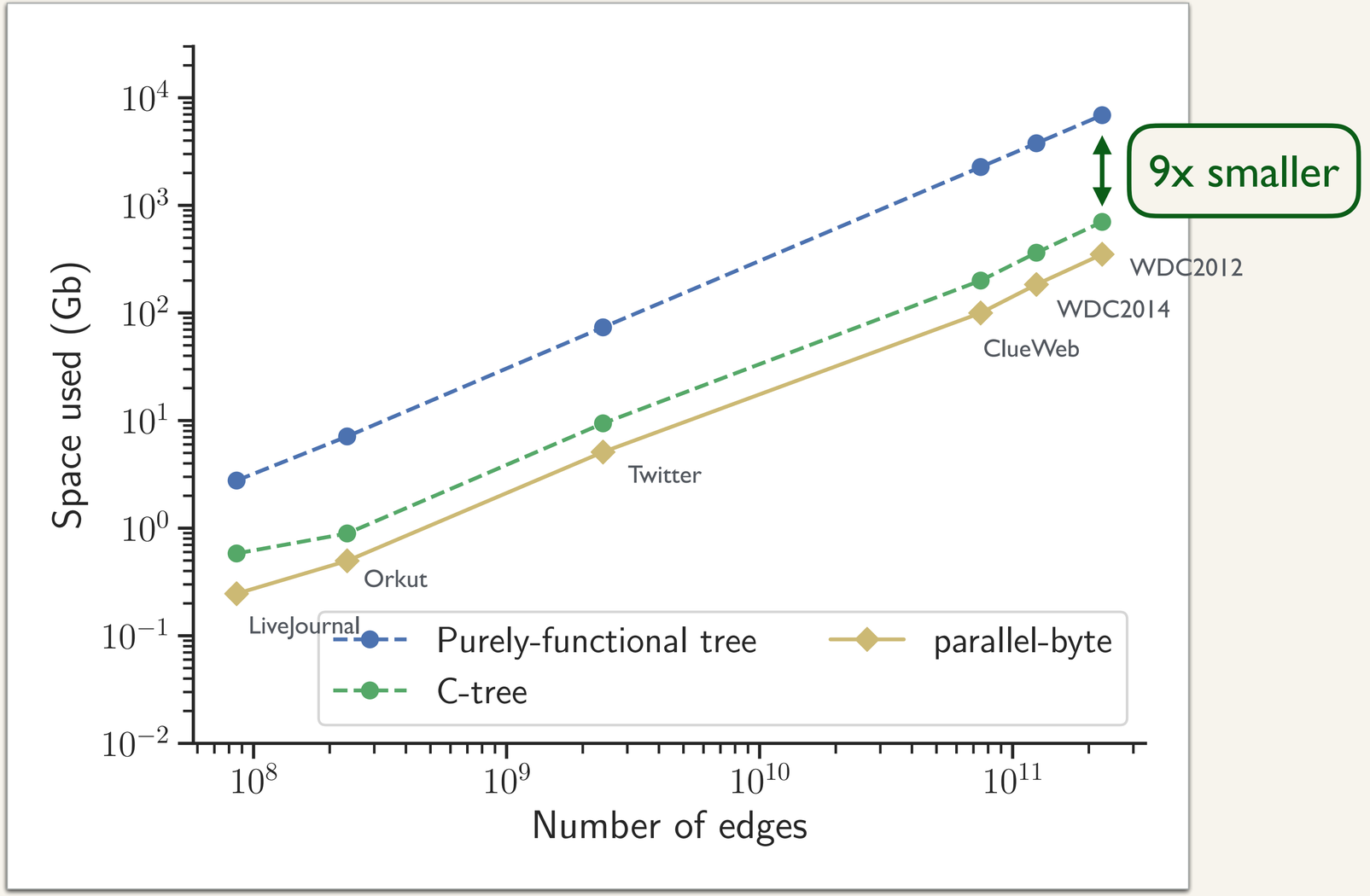


C-tree

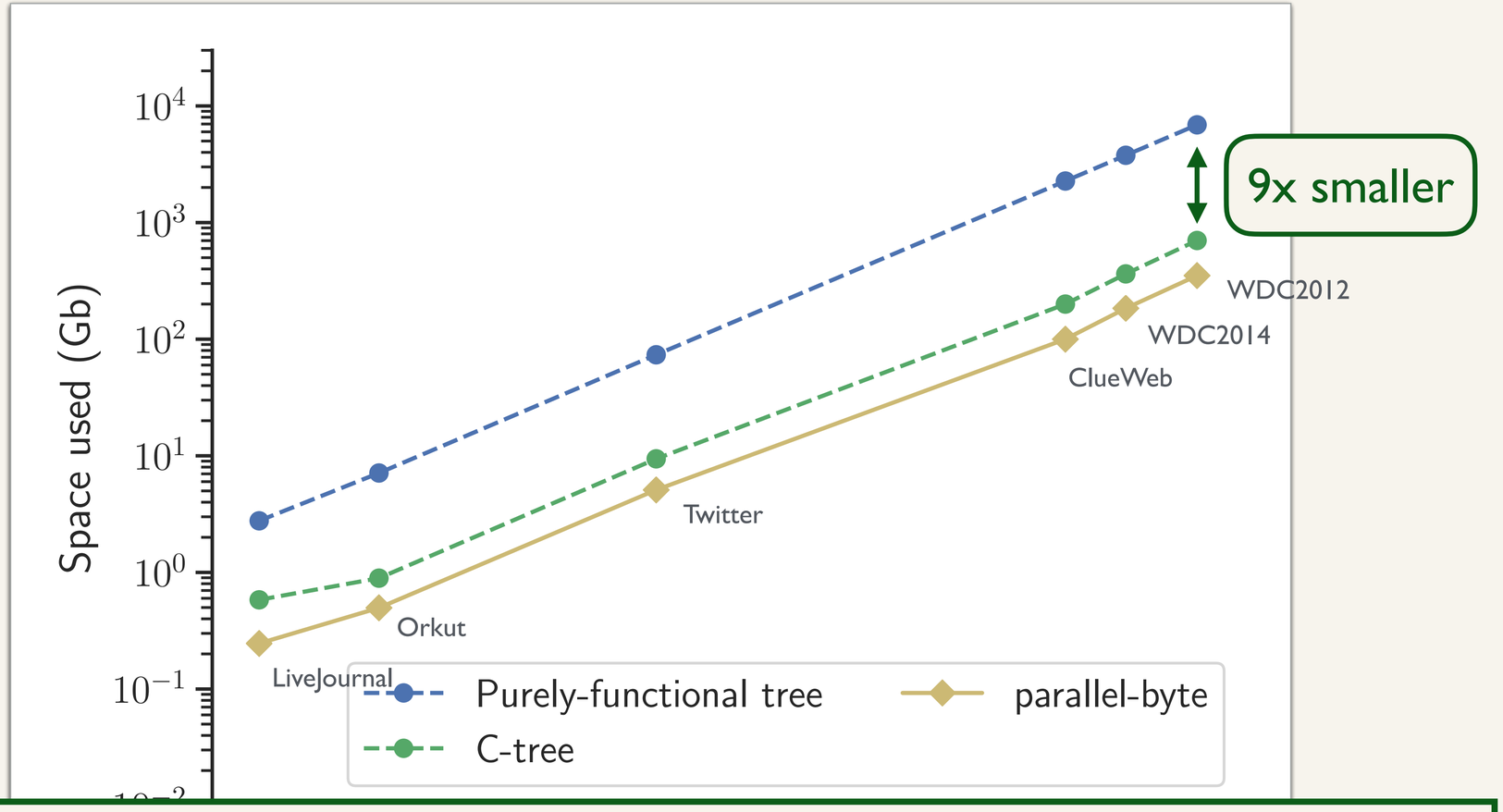
Space Improvement in Aspen using C-trees



Space Improvement in Aspen using C-trees



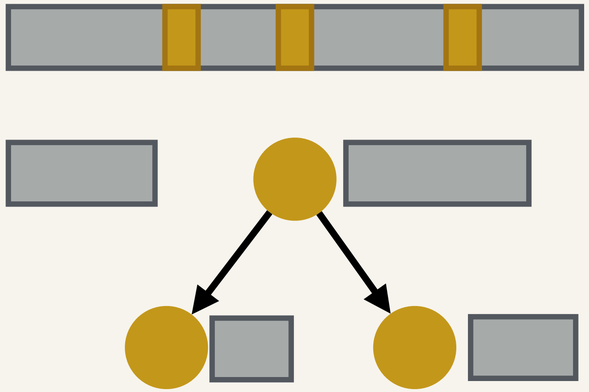
Space Improvement in Aspen using C-trees



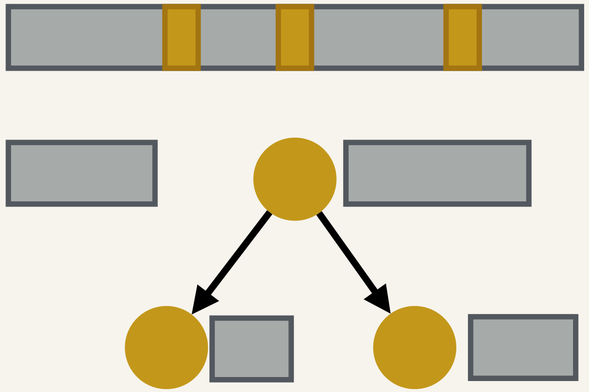
Fully-dynamic representation of the WebDataCommons hyperlink graph using 700GB of memory

Operations on C-trees

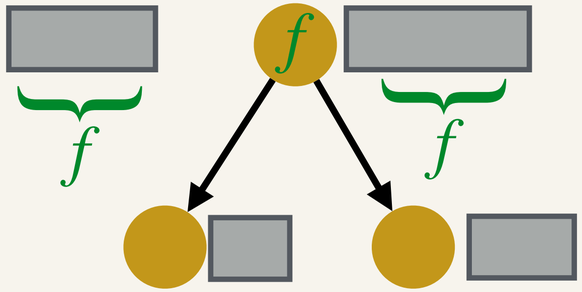
Build(Seq S)



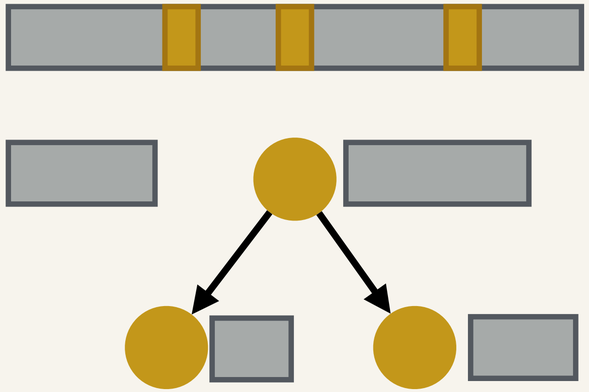
Build(Seq S)



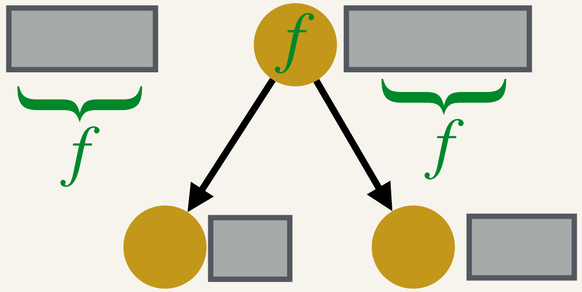
Map(Ctree C, f)



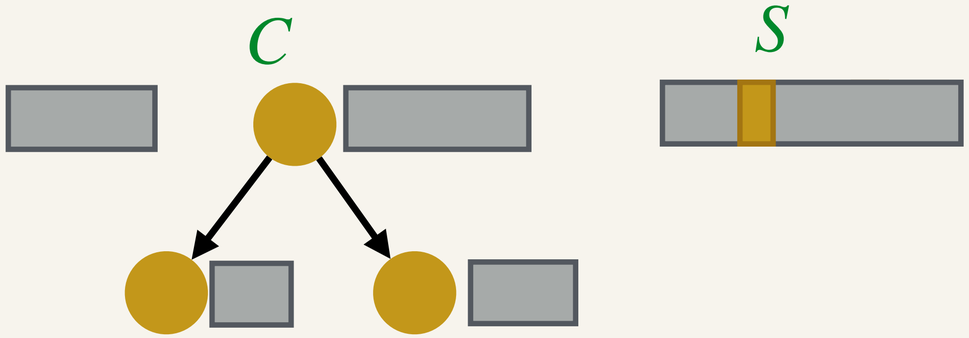
Build(Seq S)



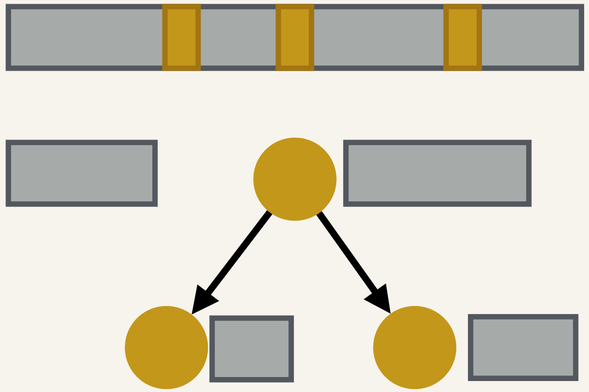
Map(Ctree C, f)



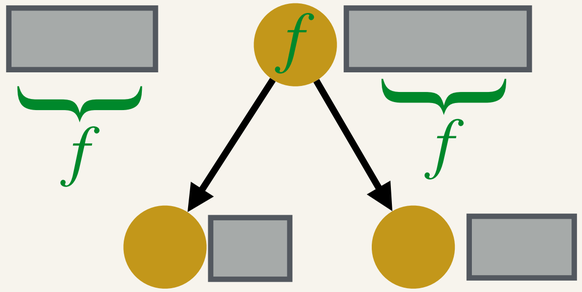
MultiInsert(Ctree C, Seq S)



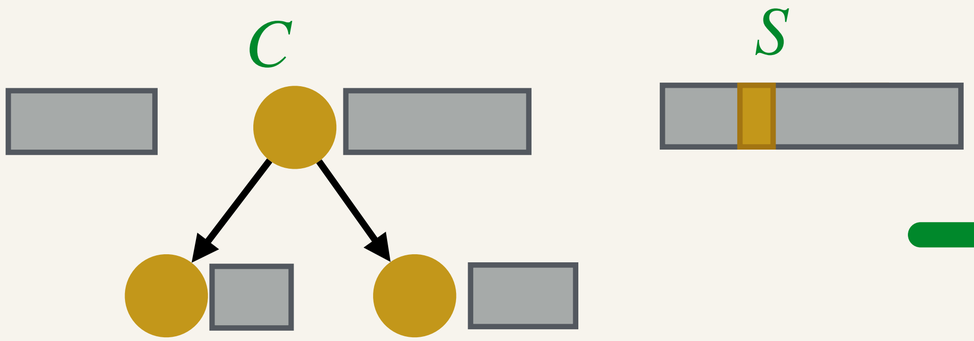
Build(Seq S)



Map(Ctree C, f)



MultiInsert(Ctree C, Seq S)



C_S = Build(Seq S)

Output = Union(C, C_S)

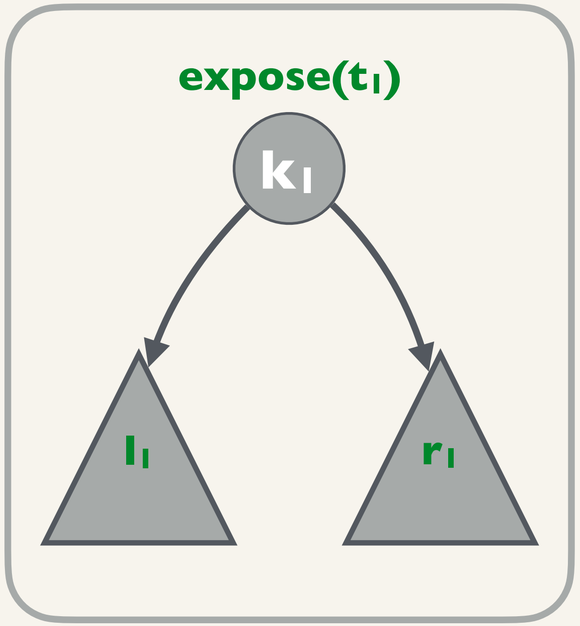
Batch Updates on Trees

union(t_1, t_2)

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

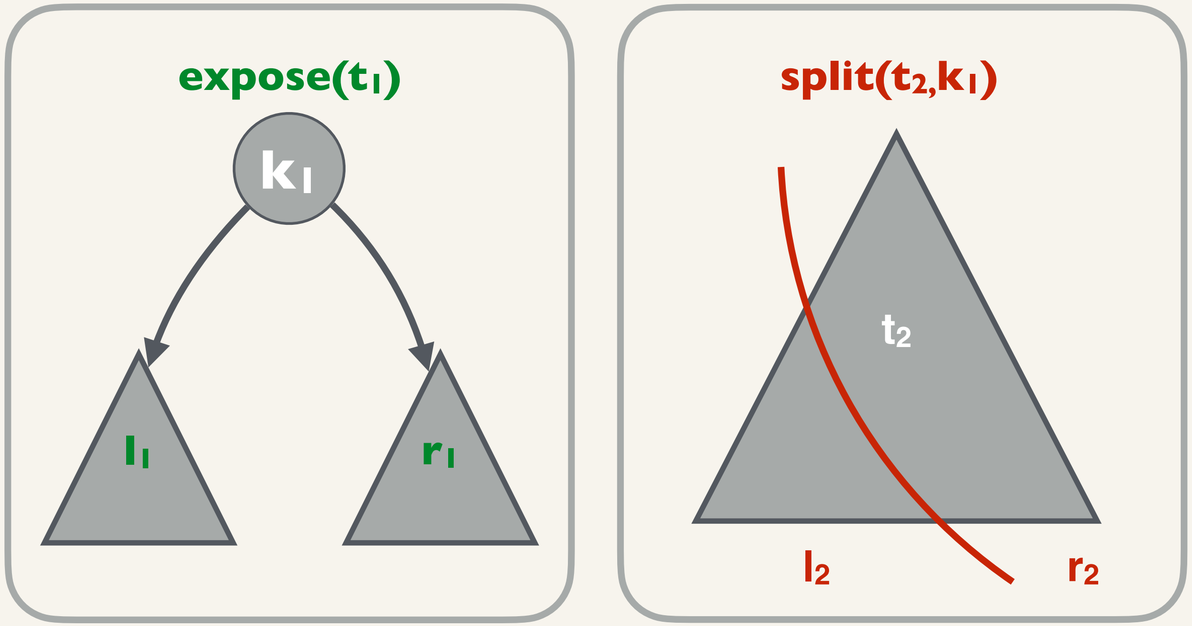
union(t_1, t_2)



[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

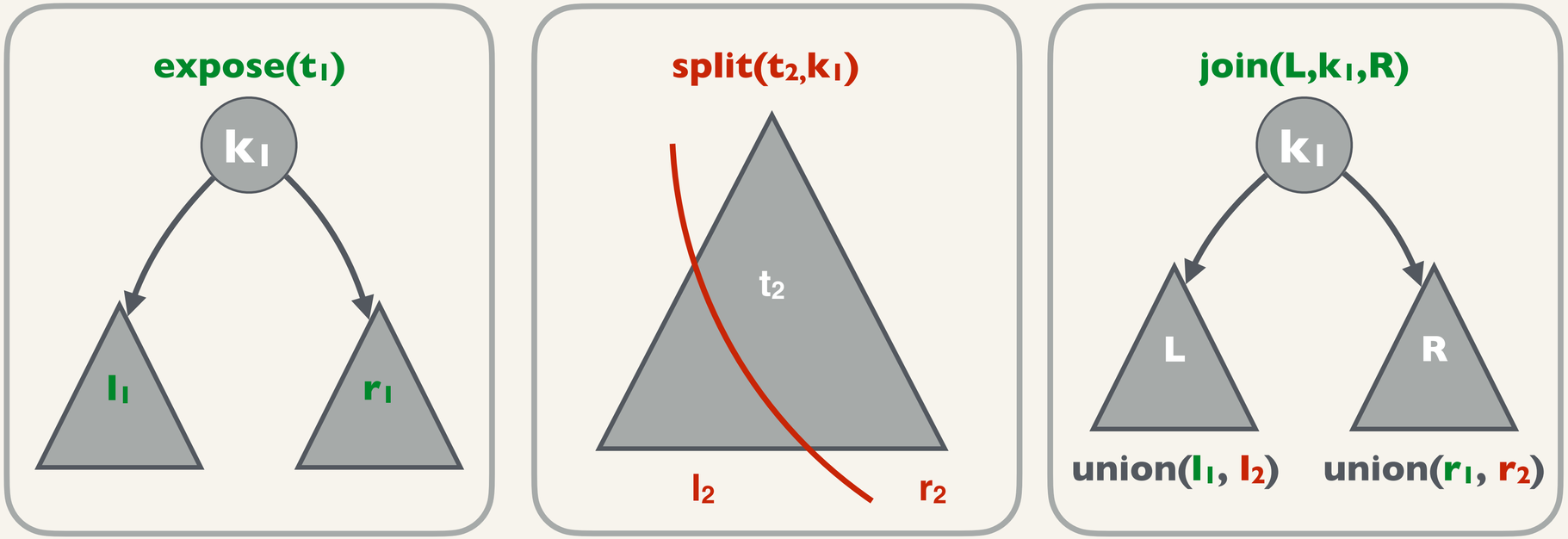
union(t_1, t_2)



[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

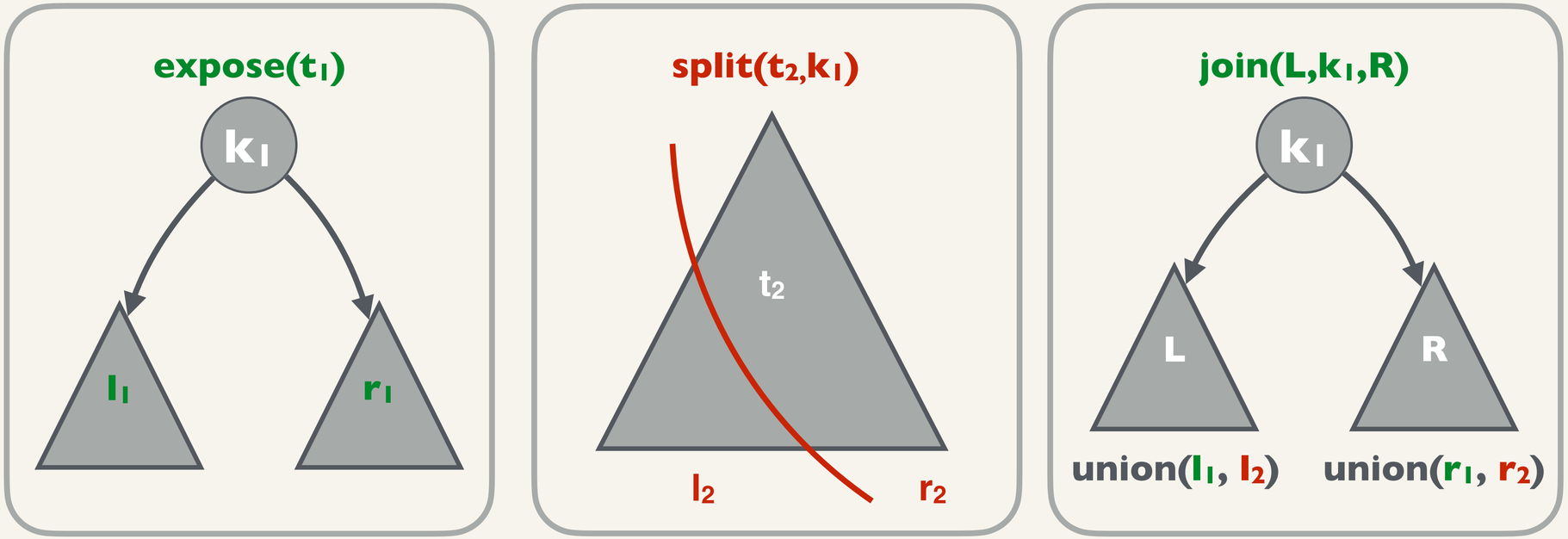
union(t_1, t_2)



[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

union(t_1, t_2)

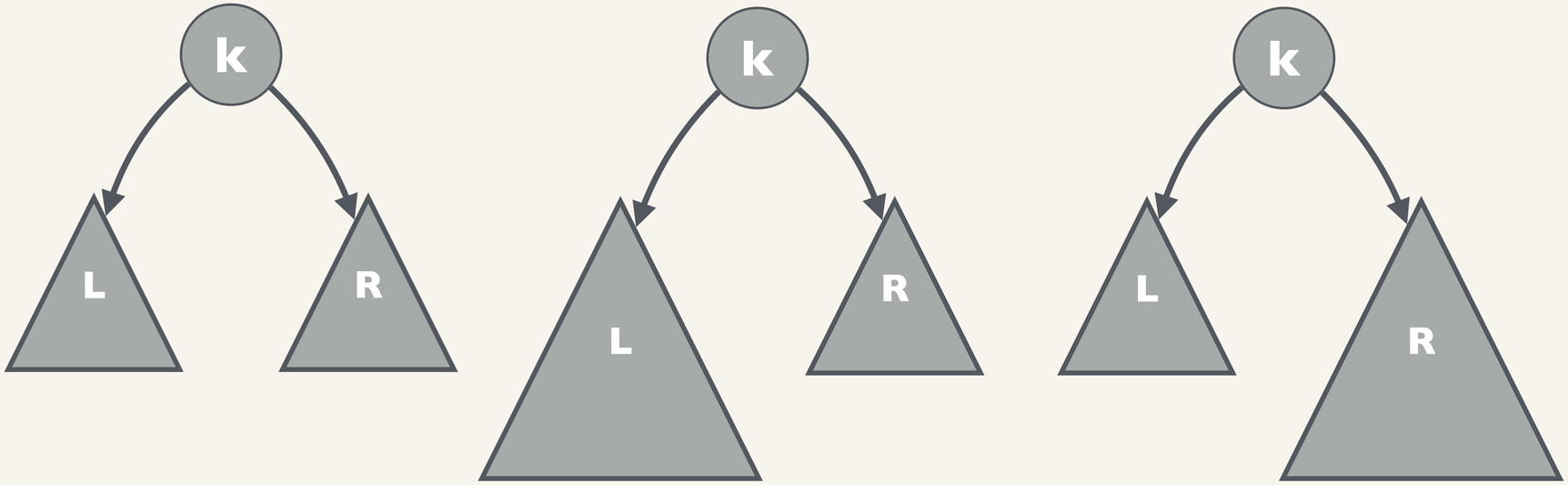


Similar algorithms for difference and intersection

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

The Join Function

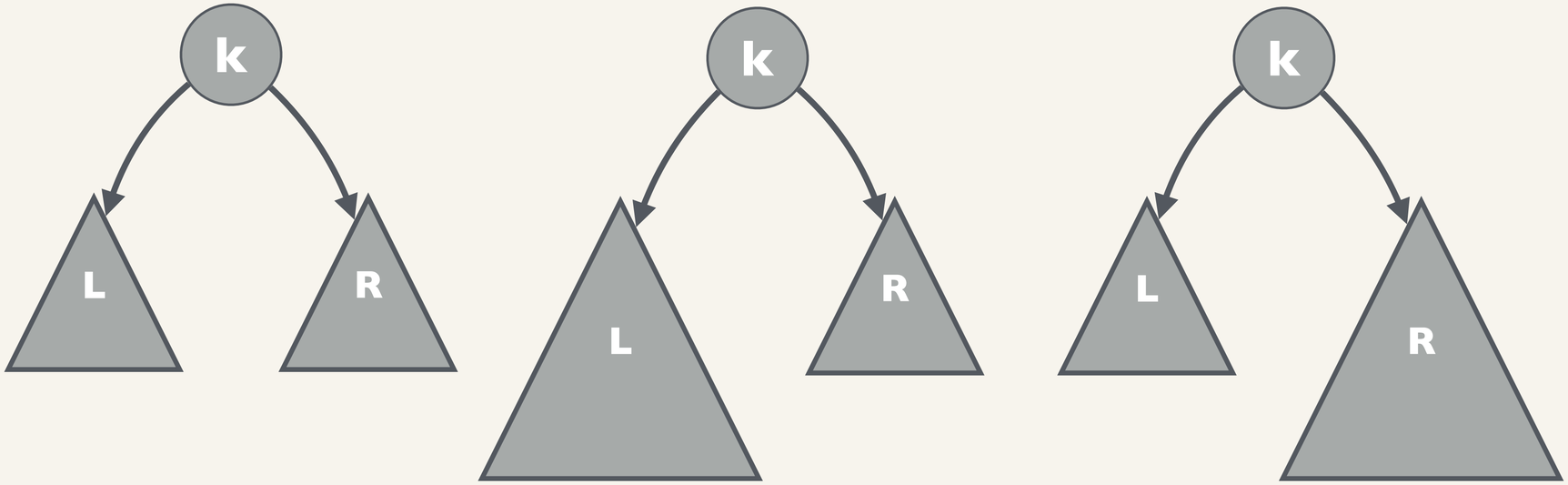
join(L, k, R)



[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

The Join Function

join(L, k, R)

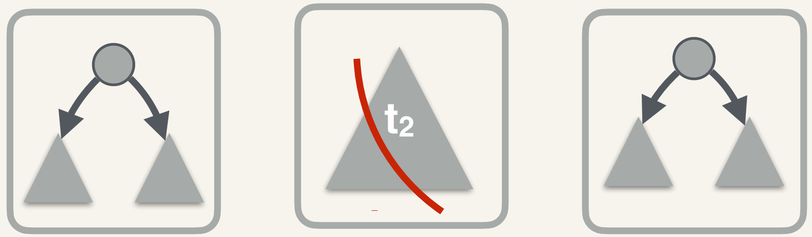


Join enables balance-agnostic expression of all other primitives[1]

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

expose split join



union(t_1, t_2) runs in

$$O\left(m \log\left(\frac{n}{m}+1\right)\right)$$ work and $$O\left(\log n \log m\right)$$ depth

Batch Updates on Trees

union(t_1, t_2) runs in

$$O\left(m \log\left(\frac{n}{m}+1\right)\right)$$ work and $$O\left(\log n \log m\right)$$ depth

Proof idea from [1]:

Splitting a tree costs $$O\left(\log |T|\right)$$ work and depth

Overall cost = work done over all **splits**

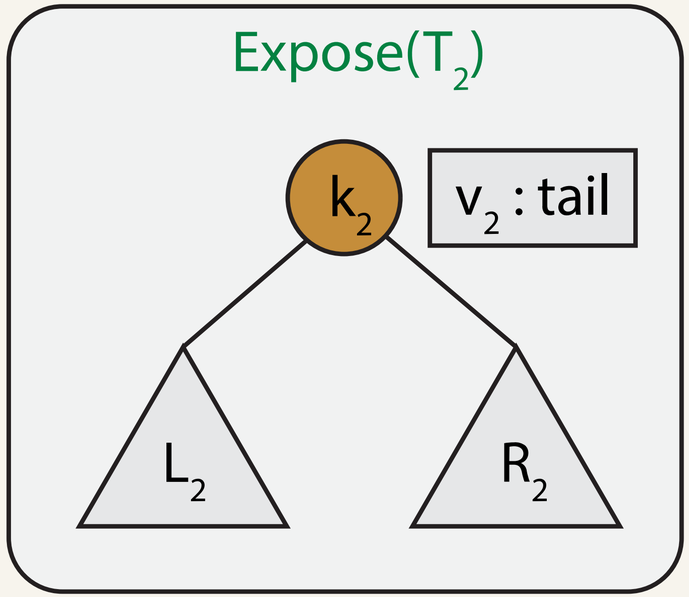
[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA'16)

Batch Updates on Trees

$$\text{union}(C_1=(T_1, P_1)), C_2=(T_2, P_2)))$$

Batch Updates on Trees

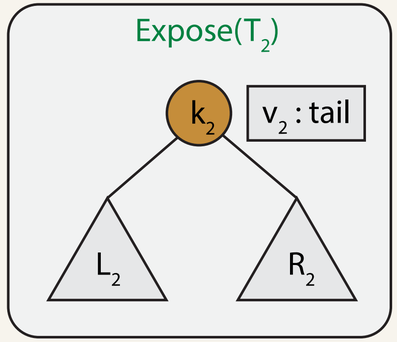
union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))

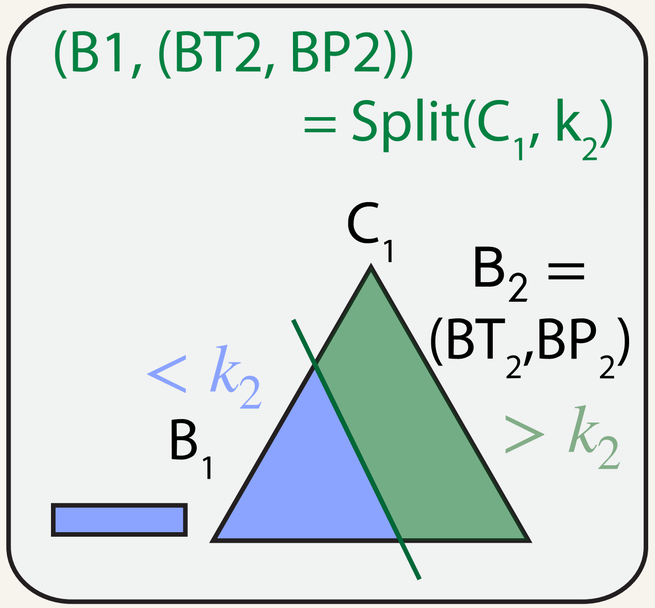


Expose one of the trees

Batch Updates on Trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))

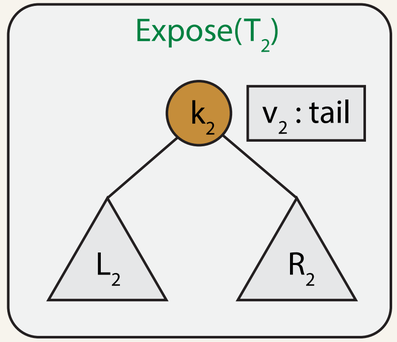


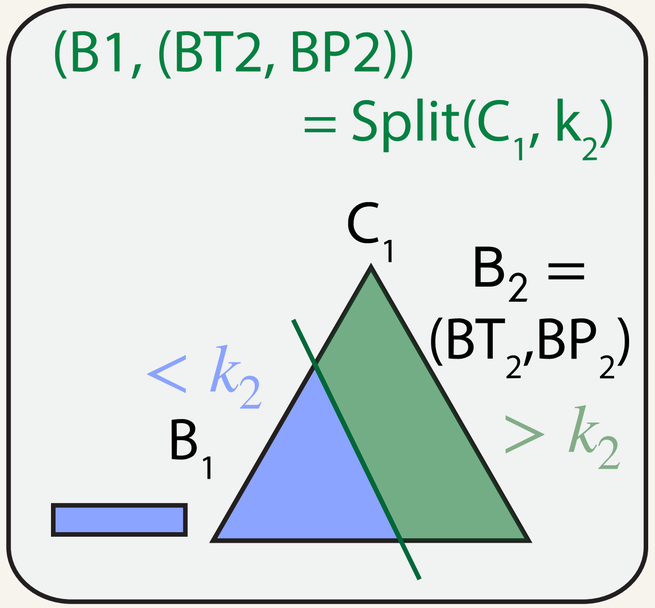


Split the other *C-tree* with k_2

Batch Updates on Trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))



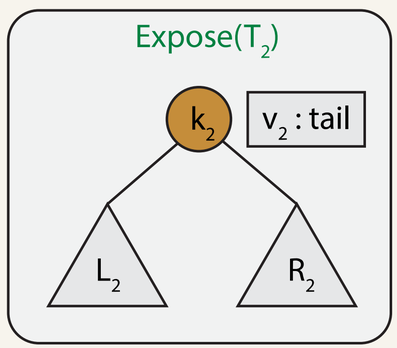


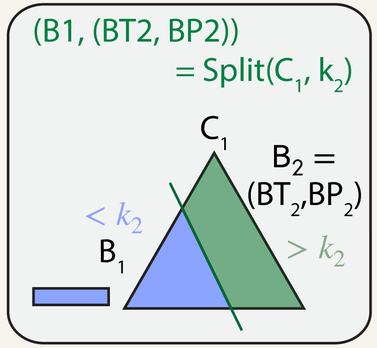
Split the other *C-tree* with k_2

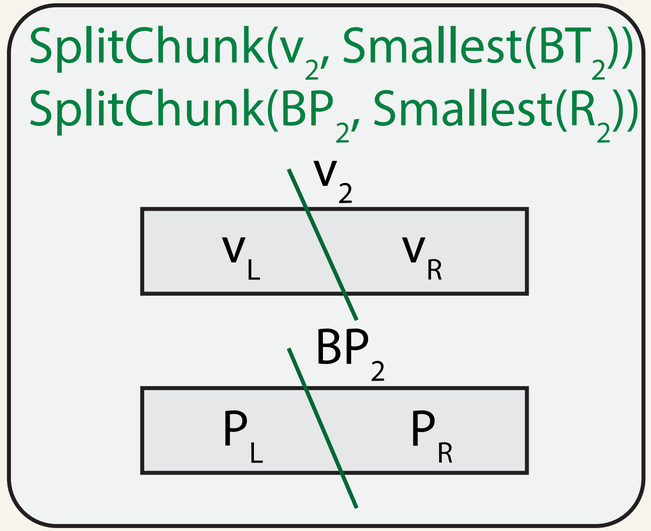
Part of v_2 may belong in BT_2, similarly with BP_2

Batch Updates on C-trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))



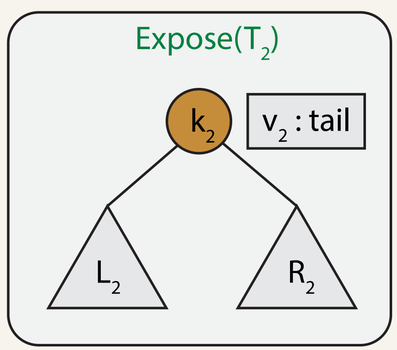


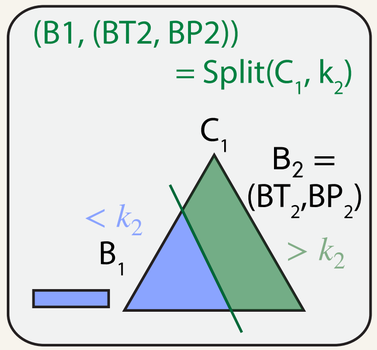


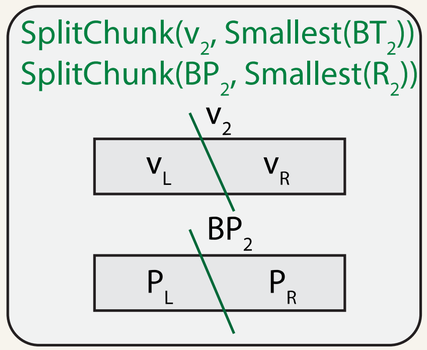
Split v_2 based on BT_2,
BP_2 based on R_2

Batch Updates on Trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))

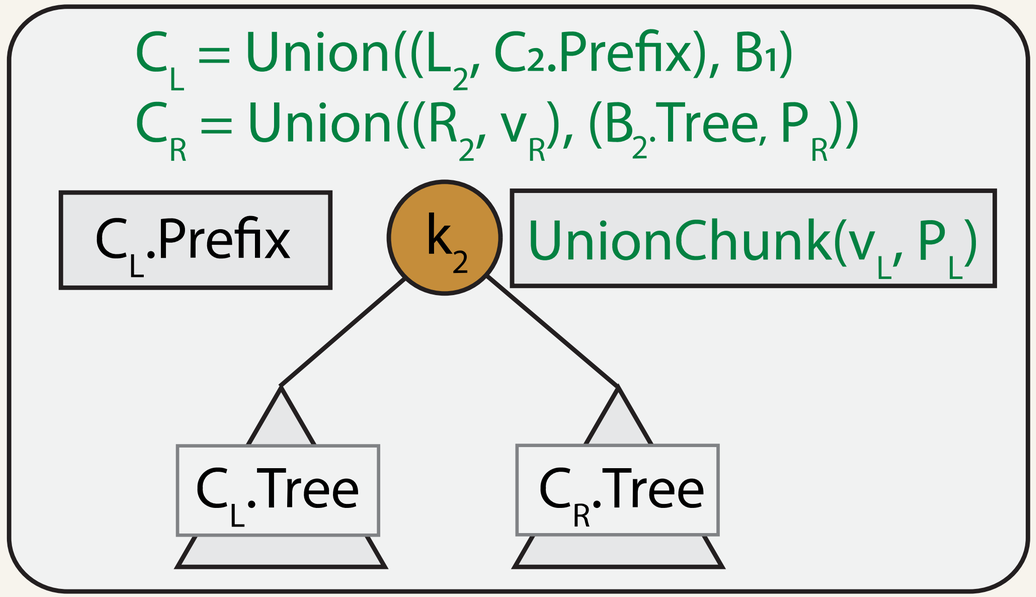






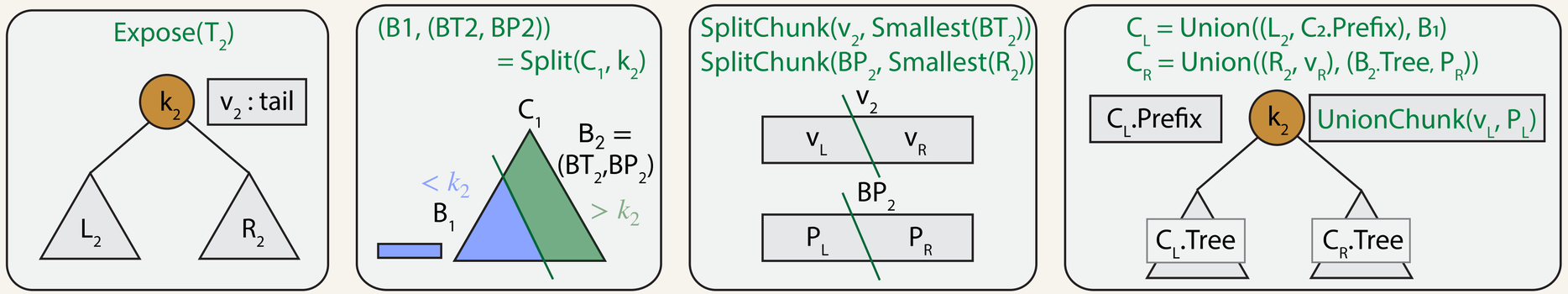
Recursive union of two C-trees

Join done on the underlying purely-functional tree



Batch Updates on Trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2)))



union(C_1, C_2) runs in

$$O\left(B^2 m \log\left(\frac{n}{m}+1\right)\right)$$ expected work

$$O\left(B \log n \log m\right)$$ depth *whp*

Experiments

Our Machine

Dell PowerEdge R930

- 72-cores, 2-way hyper-threaded*
- 1TB of main memory
- Cost: about 20k USD

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)

Our Machine

Dell PowerEdge R930

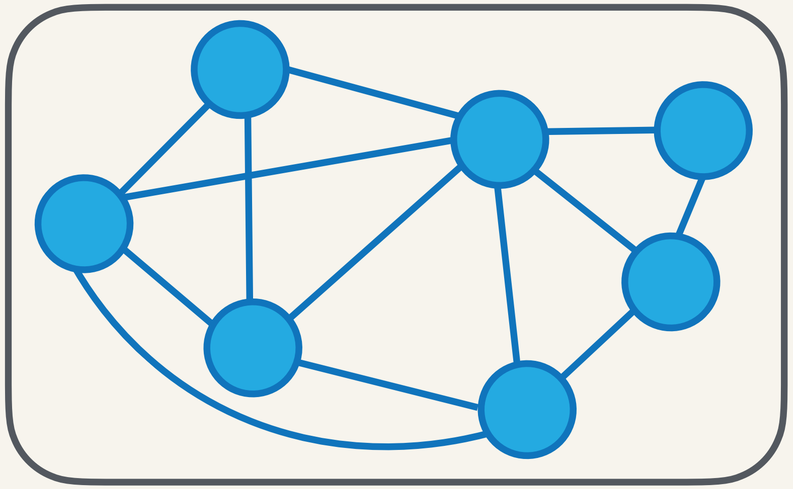
- 72-cores, 2-way hyper-threaded*
- 1TB of main memory
- Cost: about 20k USD

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)

Streaming Experiment

Streaming Experiment

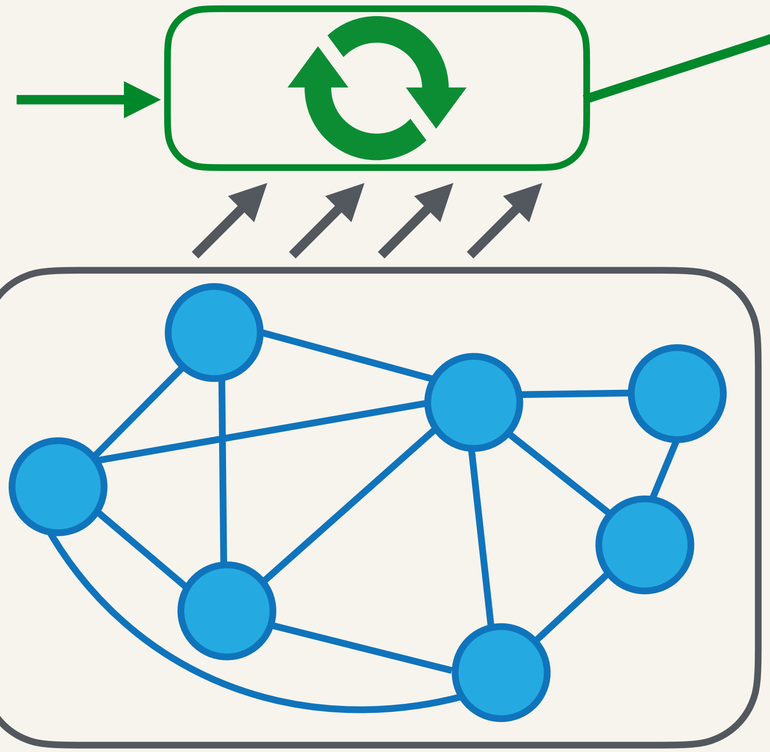
Sampled insertions
+ deletions from G



Streaming Experiment

Stream of Parallel BFS queries from random vertices

BFS trees

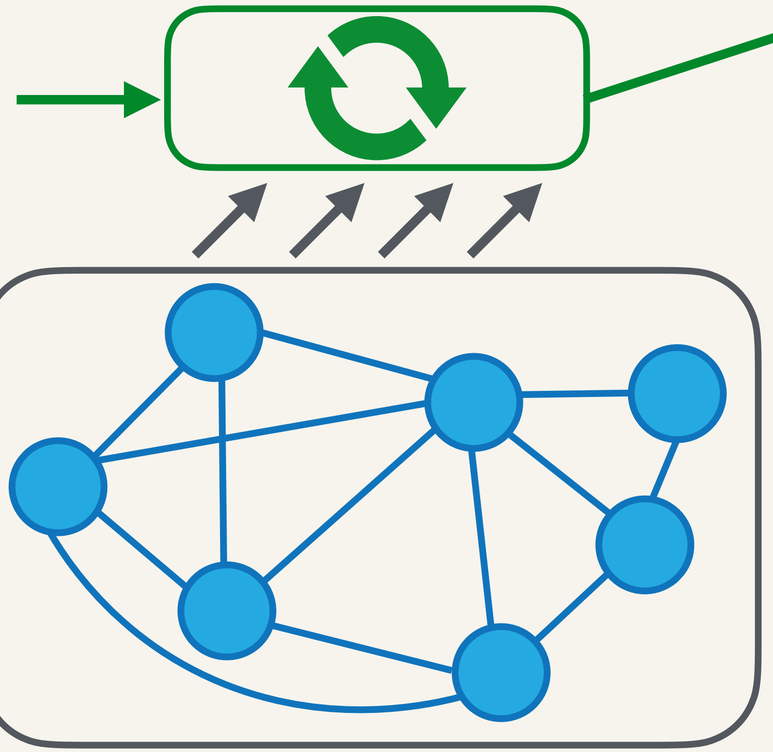


Sampled insertions + deletions from G

Streaming Experiment

Stream of Parallel BFS queries from random vertices

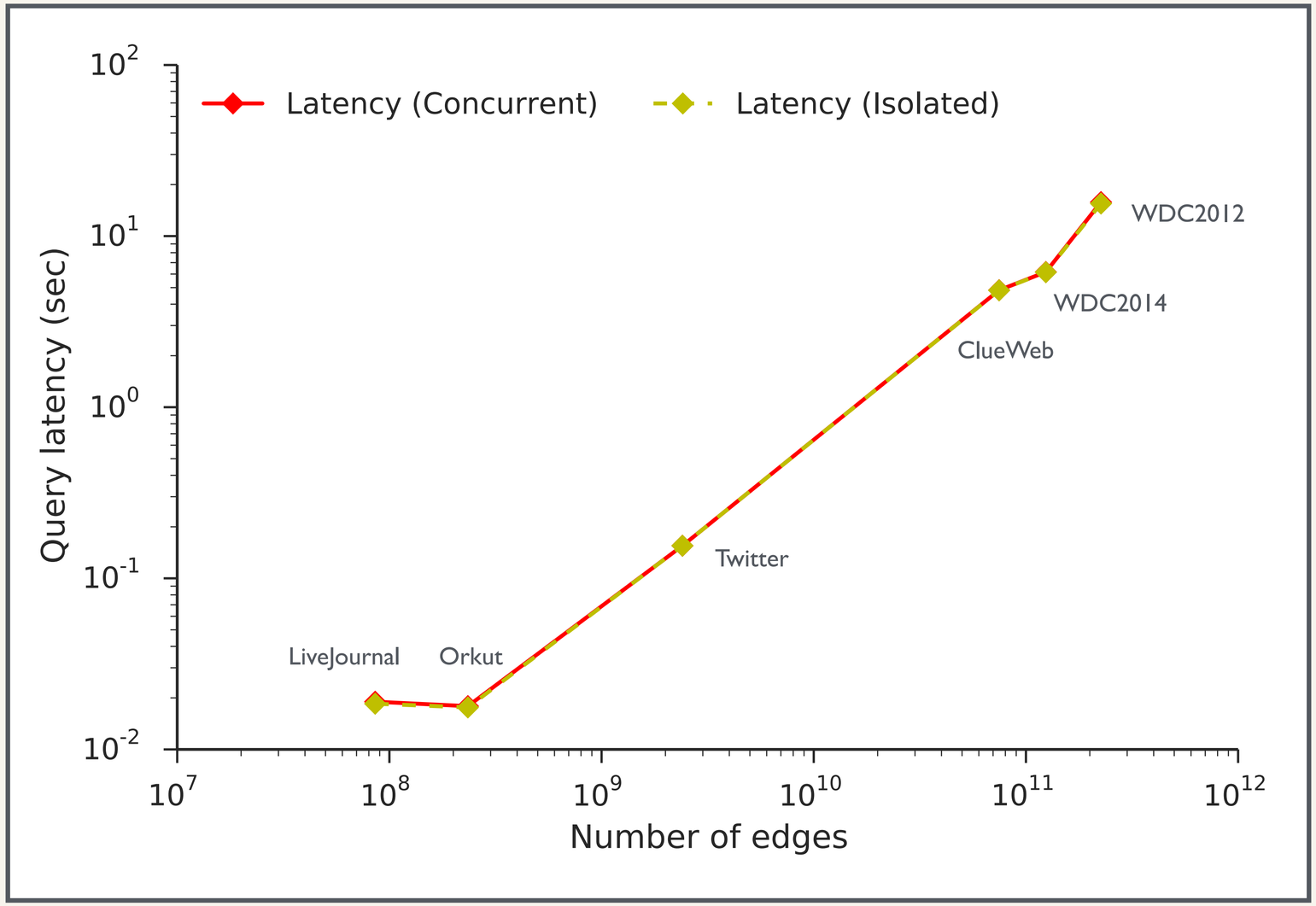
BFS trees



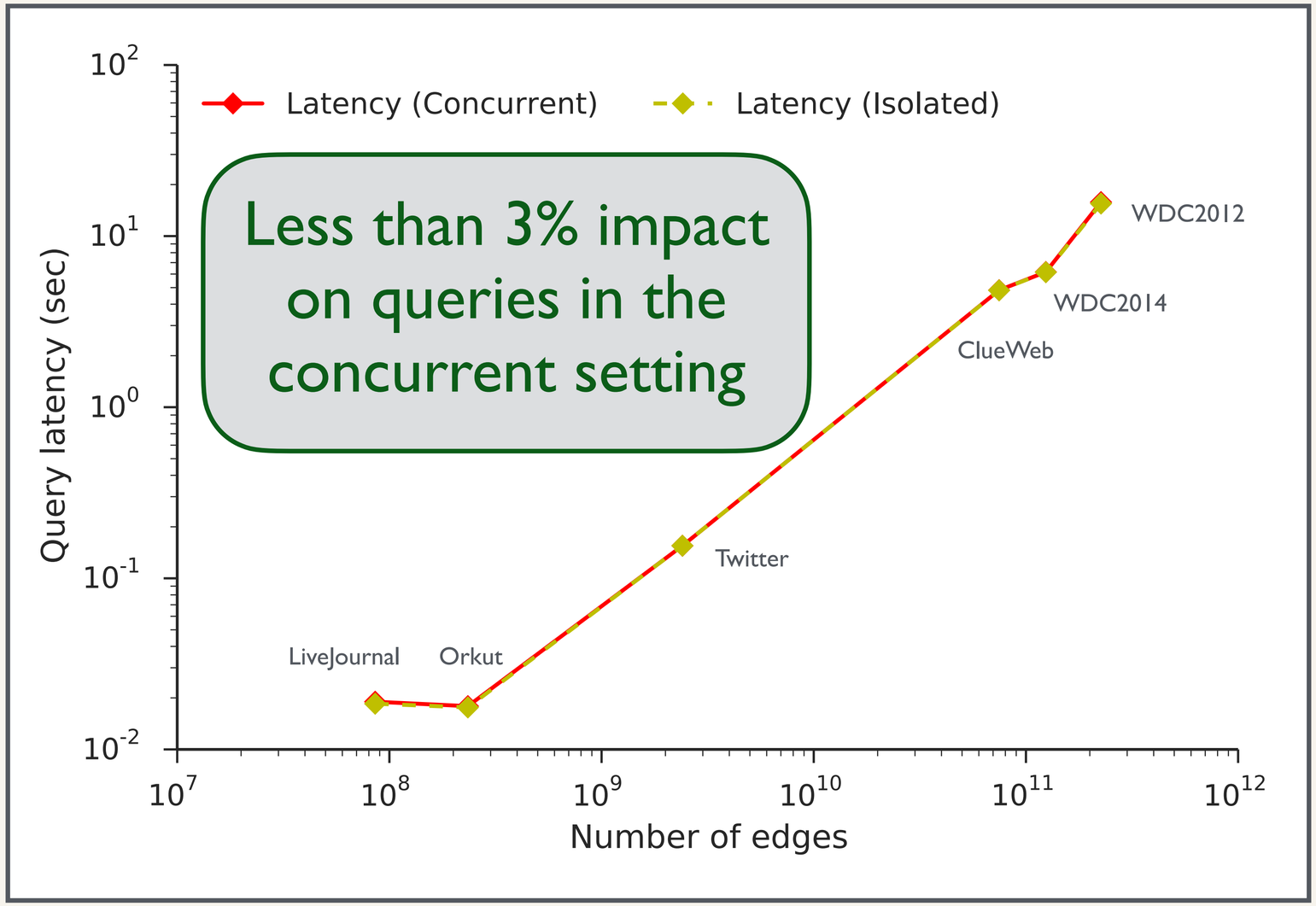
Sampled insertions + deletions from G

What's the impact on the concurrent execution on latency?

Streaming Experiment



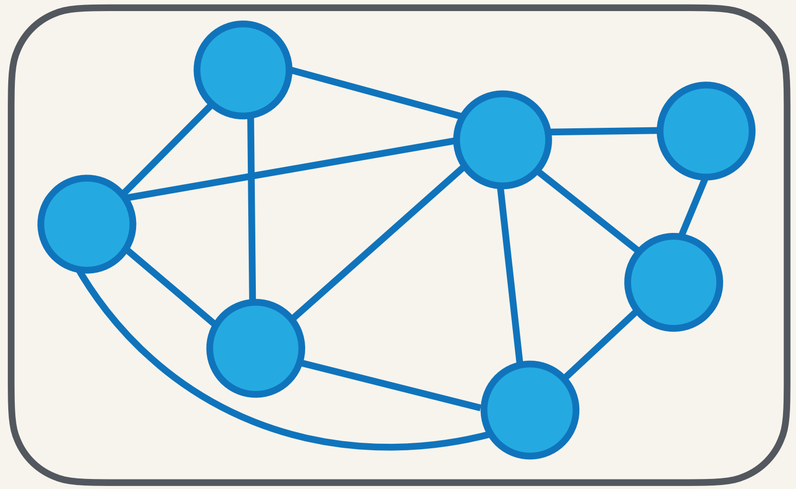
Streaming Experiment



Batch Update Experiment

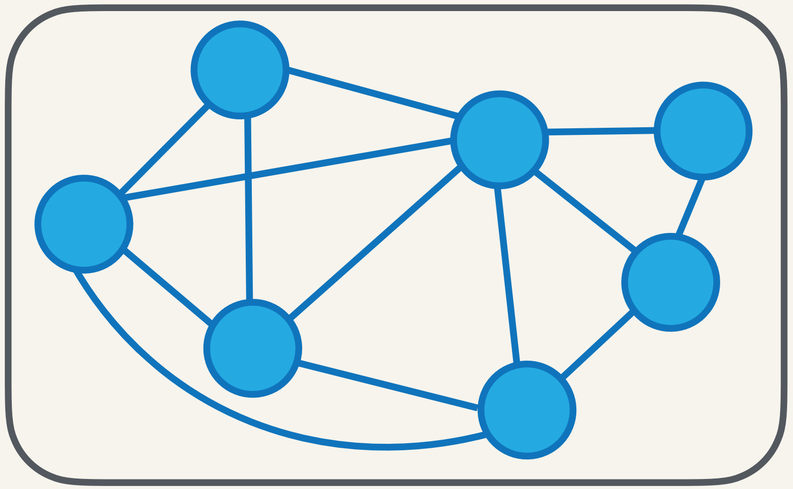
Batch Update Experiment

Edge insertions
drawn from RMAT



Batch Update Experiment

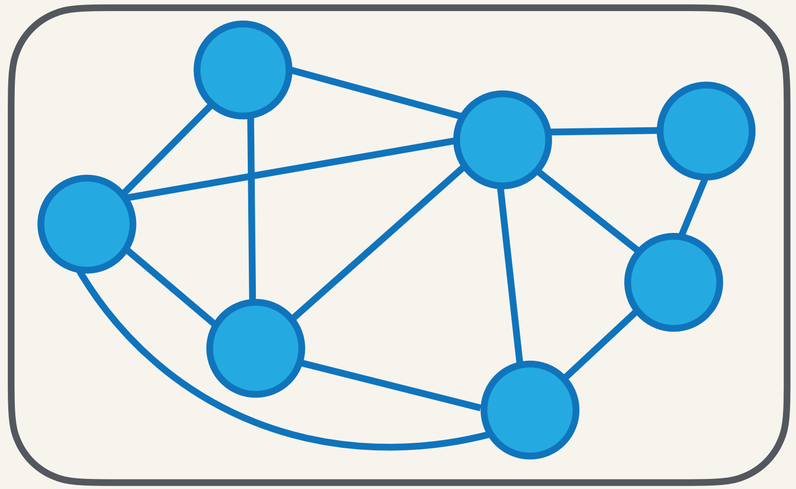
Edge insertions
drawn from RMAT



Represent G using
Aspen and STINGER

Batch Update Experiment

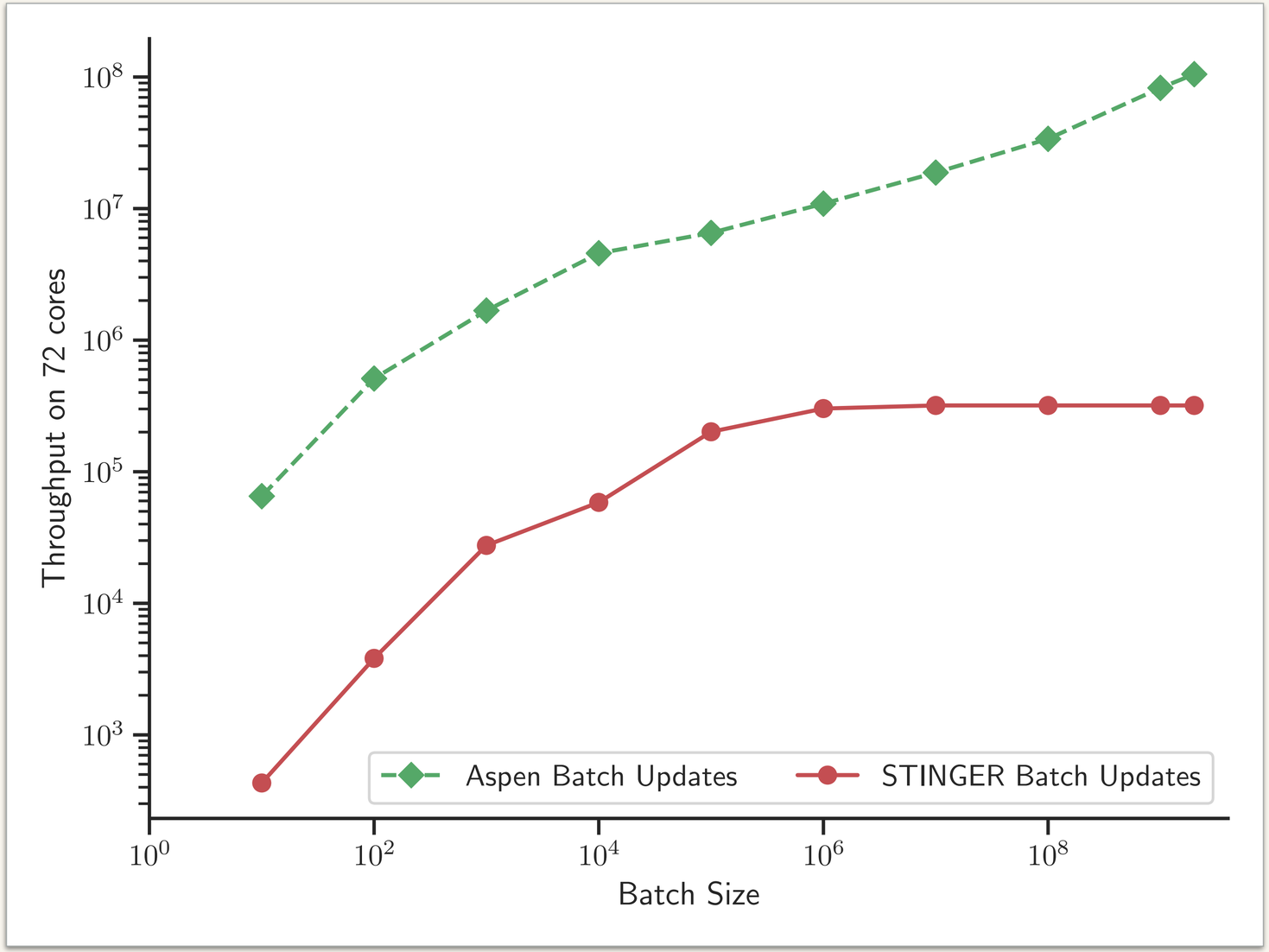
Edge insertions
drawn from RMAT



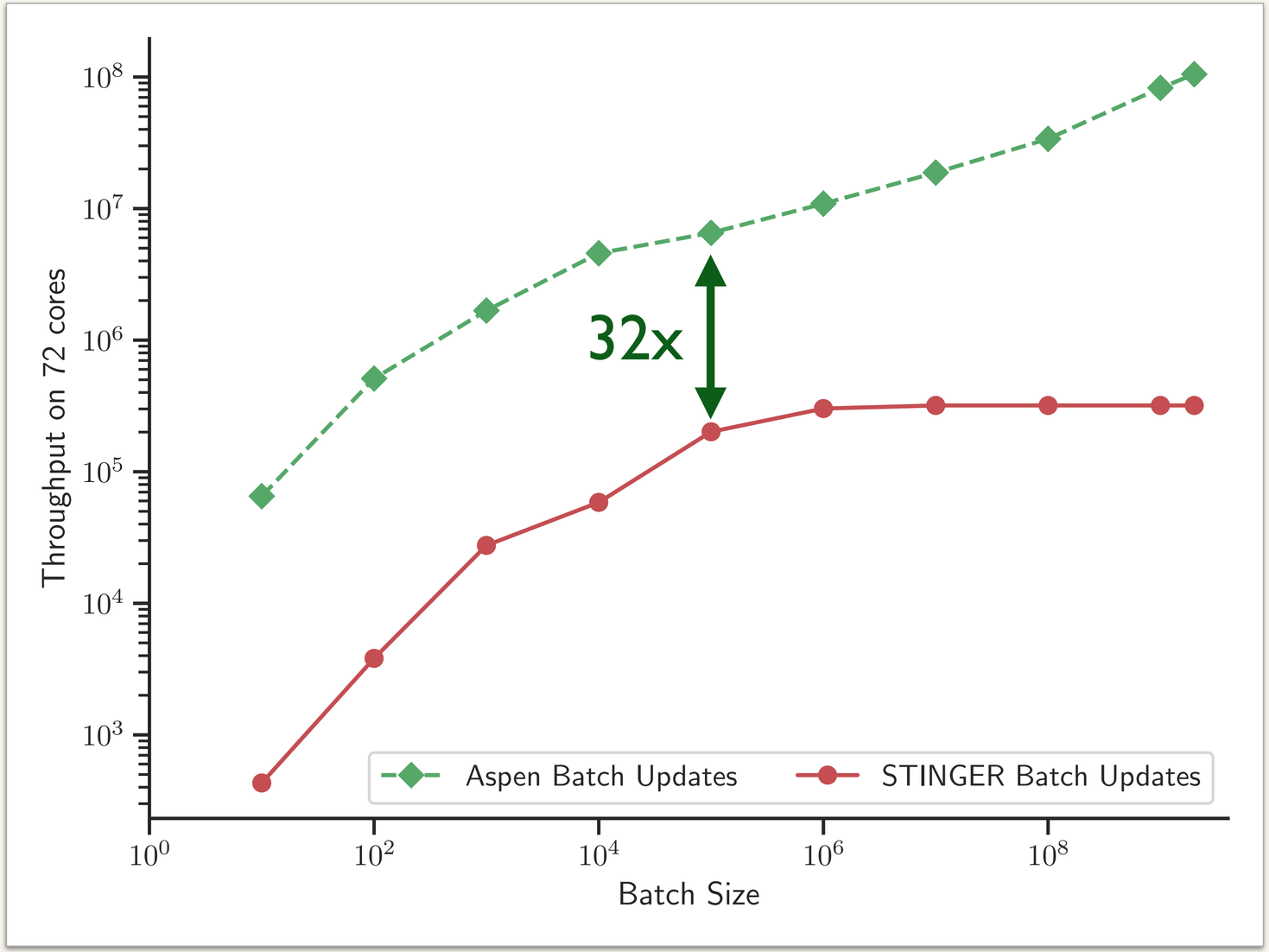
Represent G using
Aspen and STINGER

How does the throughput scale
as a function of batch size?

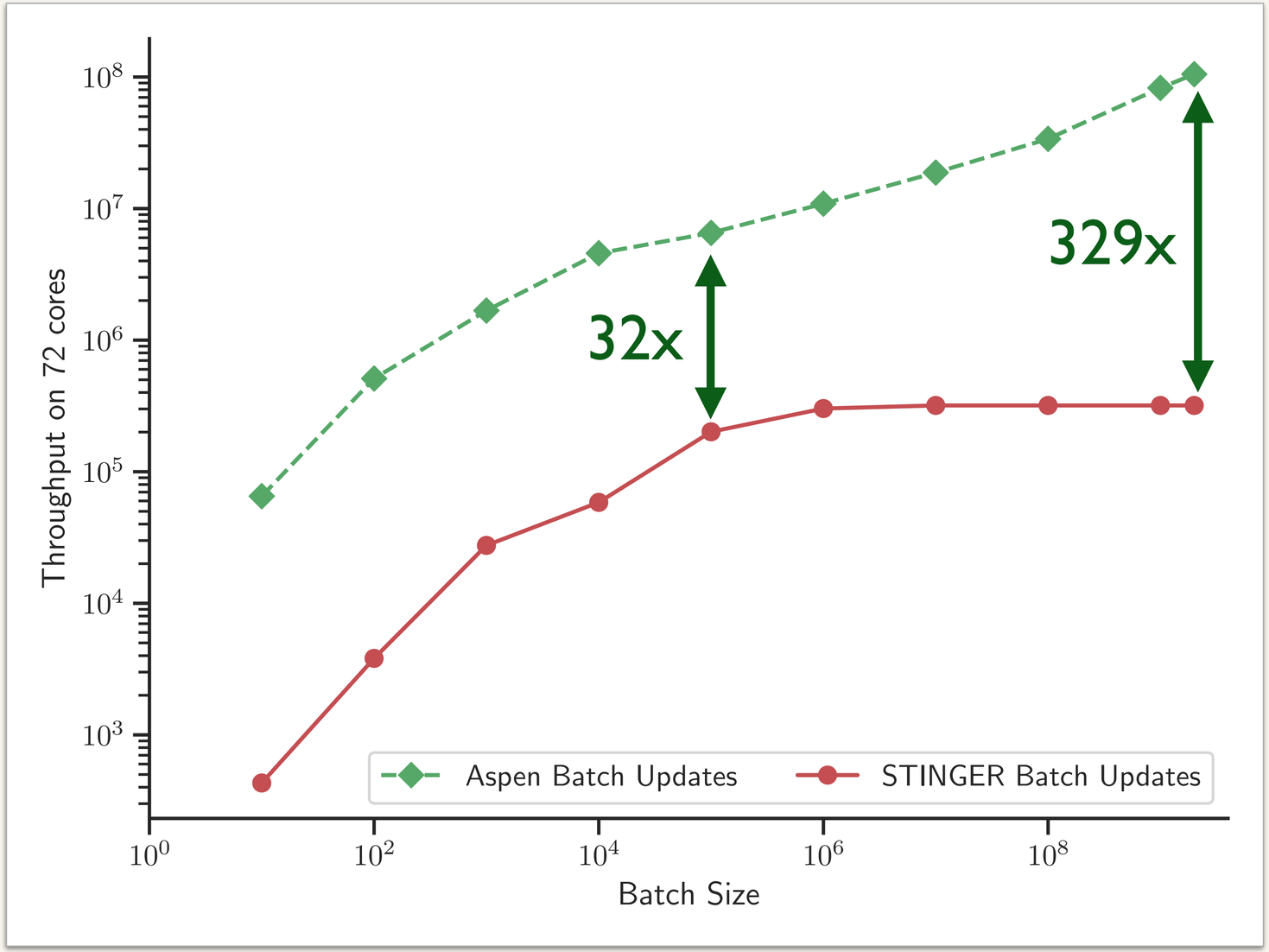
Batch Update Performance



Batch Update Performance

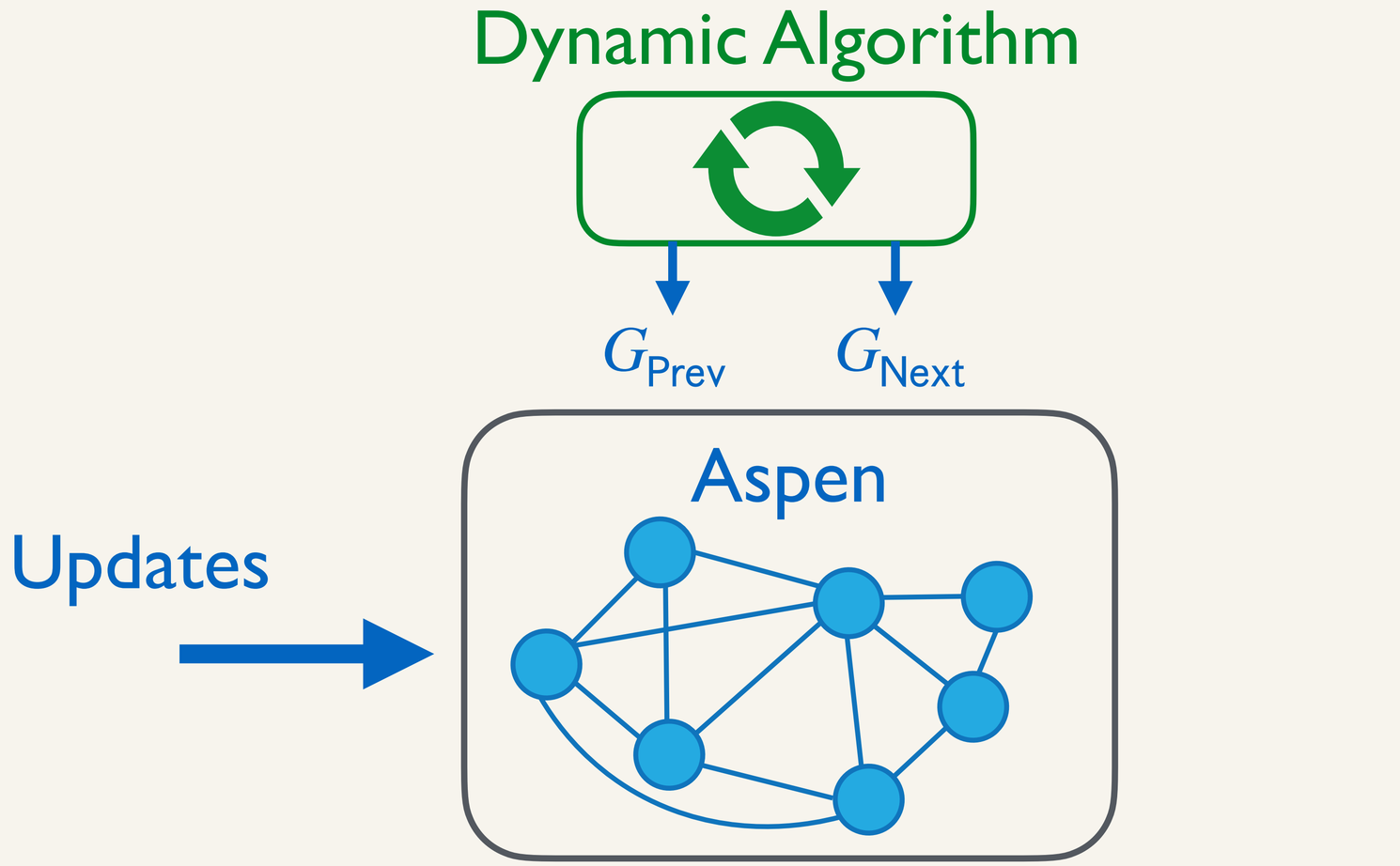


Batch Update Performance



Building on Aspen and C-trees

Batch-Dynamic Graph Processing



E.g.: Connected components, clustering coefficients, graph clusterings, etc

Batch-Dynamic Algorithms

Forest Conn.

Batch-Parallel Euler Tour Trees*

Thomas Tseng[†] Laxman Dhulipala[‡] Guy Blelloch[§]

Abstract

The dynamic trees problem is to maintain a forest undergoing edge insertions and deletions while supporting queries for information such as connectivity. There are many existing data structures for this problem, but few of them are capable of exploiting parallelism in the batch setting, in which large batches of edges are inserted or deleted from the forest at once. In this paper, we demonstrate that the Euler tour tree, an existing sequential dynamic trees data structure, can be parallelized in the batch setting. For a batch of k updates over a forest of n vertices, our parallel Euler tour trees achieve $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability. Our work bound is asymptotically optimal, and we improve on the depth bound achieved by Acar et al. for the batch-parallel dynamic trees problem [1].

Our main building block for parallelizing Euler tour trees is a batch-parallel skip list data structure, which we believe may be of independent interest. Euler tour trees require a sequence data structure capable of joins and splits. Traditionally, balanced binary trees are used, but they are difficult to join or split in parallel when processing batches of updates. We show that skip lists, on the other hand, support batches of joins or splits of size k over n elements with $O(k \log(1 + n/k))$ work in expectation and $O(\log n)$ depth with high probability. We also achieve the same efficiency bounds for augmented skip lists, which allows us to augment our Euler tour trees to support subtree queries.

Our data structures achieve between 67–96× self-relative speedup on 72 cores with hyper-threading on large batch sizes. Our data structures also significantly outperform the fastest existing sequential dynamic trees data structures empirically.

*This is the full version of the paper appearing in ALENEX 2019.

[†]Computer Science Department, Carnegie Mellon University, thomastu@alumni.cmu.edu
[‡]Computer Science Department, Carnegie Mellon University, ldhulipa@cs.cmu.edu
[§]Computer Science Department, Carnegie Mellon University, guyb@cs.cmu.edu

[TDB'18]

Connectivity

Parallel Batch-Dynamic Graph Connectivity*

Umut A. Acar
Carnegie Mellon University
umut@cs.cmu.edu

Daniel Anderson
Carnegie Mellon University
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Abstract

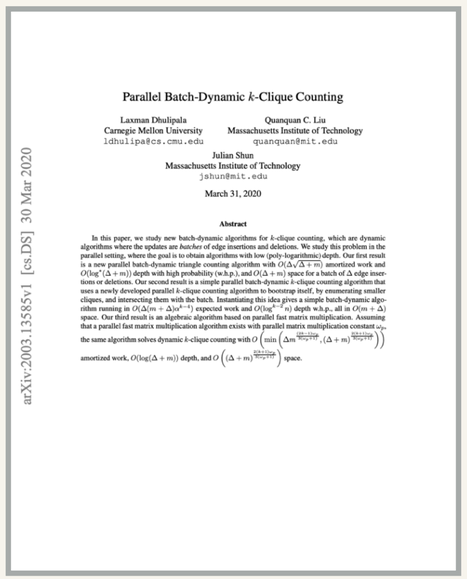
In this paper, we study batch parallel algorithms for the dynamic connectivity problem, a fundamental problem that has received considerable attention in the sequential setting. The most well known sequential algorithm for dynamic connectivity is the elegant level-set algorithm of Holm, de Lichtenberg and Thorup (HDT), which achieves $O(\lg^2 n)$ amortized time per edge insertion or deletion, and $O(\lg n / \lg \lg n)$ time per query.

We design a parallel batch-dynamic connectivity algorithm that is work-efficient with respect to the HDT algorithm for small batch sizes, and is asymptotically faster when the average batch size is sufficiently large. Given a sequence of batched updates, where Δ is the average batch size of all deletions, our algorithm achieves $O(\lg n \lg(1 + n/\Delta))$ expected amortized work per edge insertion and deletion and $O(\lg^2 n)$ depth w.h.p. Our algorithm answers a batch of k connectivity queries in $O(k \lg(1 + n/k))$ expected work and $O(\lg n)$ depth w.h.p. To the best of our knowledge, our algorithm is the first parallel batch-dynamic algorithm for connectivity.

*This is the full version of the paper appearing in the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2019

[AABD'19]

Clique-counting



Parallel Batch-Dynamic k-Clique Counting

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Quanquan C. Liu
Massachusetts Institute of Technology
quanquan@mit.edu

Julian Shun
Massachusetts Institute of Technology
jshun@mit.edu

March 31, 2020

Abstract

In this paper, we study new batch-dynamic algorithms for k-clique counting, which are dynamic algorithms where the updates are *batches* of edge insertions and deletions. We study this problem in the parallel setting, where the goal is to obtain algorithms with low (poly-logarithmic) depth. Our first result is a new parallel batch-dynamic triangle counting algorithm with $O(\Delta\sqrt{\Delta + m})$ amortized work and $O(\log^*(\Delta + m))$ depth with high probability (w.h.p.), and $O(\Delta + m)$ space for a batch of Δ edge insertions or deletions. Our second result is a simple parallel batch-dynamic k-clique counting algorithm that uses a newly developed parallel k-clique counting algorithm to bootstrap itself, by enumerating smaller cliques, and intersecting them with the batch. Instantiating this idea gives a simple batch-dynamic algorithm running in $O(\Delta(m + \Delta)\alpha^{k-4})$ expected work and $O(\log^{k-2} n)$ depth w.h.p., all in $O(m + \Delta)$ space. Our third result is an algebraic algorithm based on parallel fast matrix multiplication. Assuming that a parallel fast matrix multiplication algorithm exists with parallel matrix multiplication constant ω_p, the same algorithm solves dynamic k-clique counting with $O\left(\min\left(\Delta m^{\frac{(k\omega_p+2)\log}{2\log+k\omega_p}}, (\Delta + m)^{\frac{k\omega_p+2\log}{2\log+k\omega_p}}\right)\right)$ amortized work, $O(\log(\Delta + m))$ depth, and $O\left((\Delta + m)^{\frac{k\omega_p+2\log}{2\log+k\omega_p}}\right)$ space.

[DLSY'20]

Interested in practical and memory-efficient dynamic graph algorithms

Thank you!

Aspen

Scalable graph data structures and interfaces for processing streaming graphs

github.com/ldhulipala/aspen

- has strong theoretical bounds
- provides memory-efficient graph representations
- enables lightweight snapshots
- runs on commodity hardware
- can process the largest publicly-available graphs