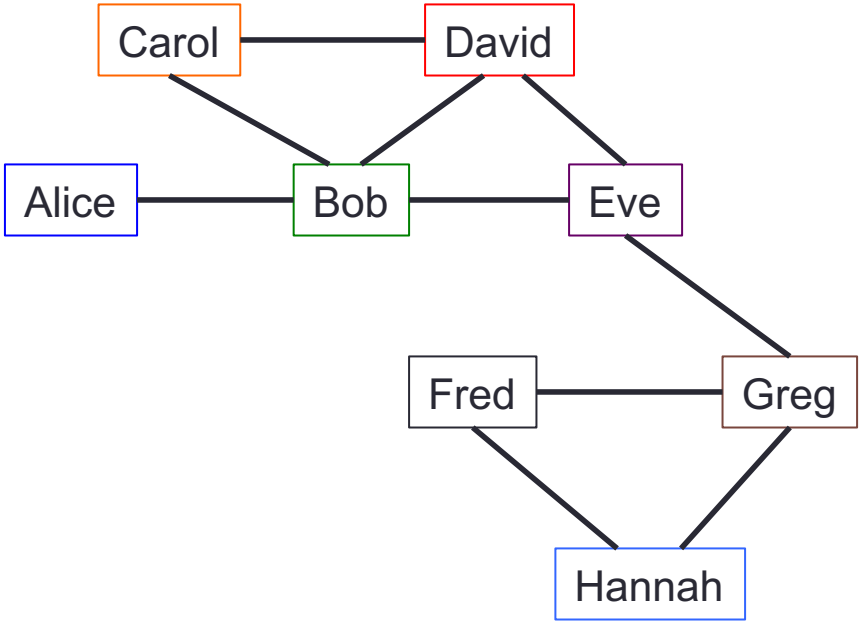
# High-Performance Frameworks for Static and Streaming Graph Processing

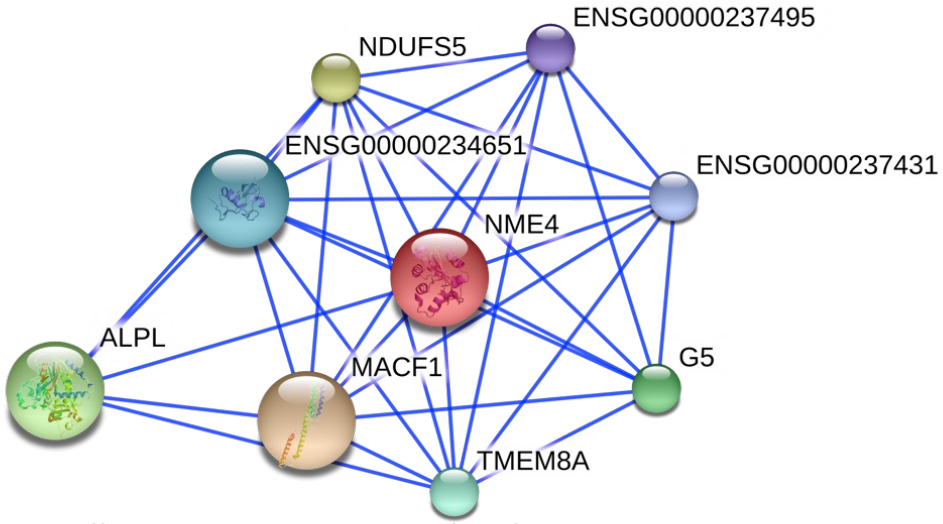
Julian Shun (MIT CSAIL)

Joint work with Saman Amarasinghe, Riyadh Baghdadi, Guy Blelloch, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Emily Furst, Changwan Hong, Claire Hsu, Tommy Jung, Shoaib Kamil, Mark Oskin, Dustin Richmond, Max Ruttenberg, Daniel Sanchez, Michael Taylor, Mengjiao Yang, Victor Ying, Yunming Zhang

# What is a graph?

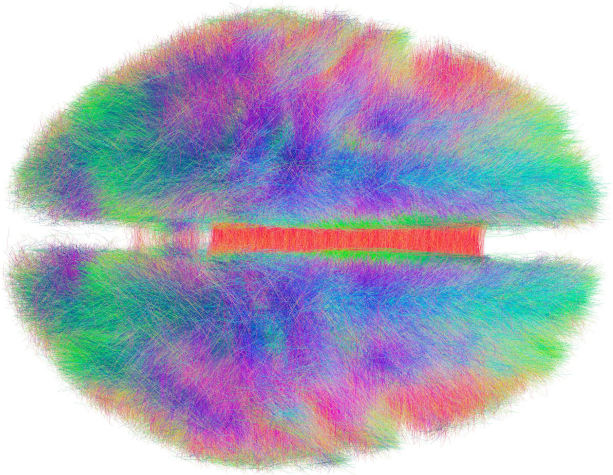

- **Vertices** model objects
- **Edges** model relationships between objects



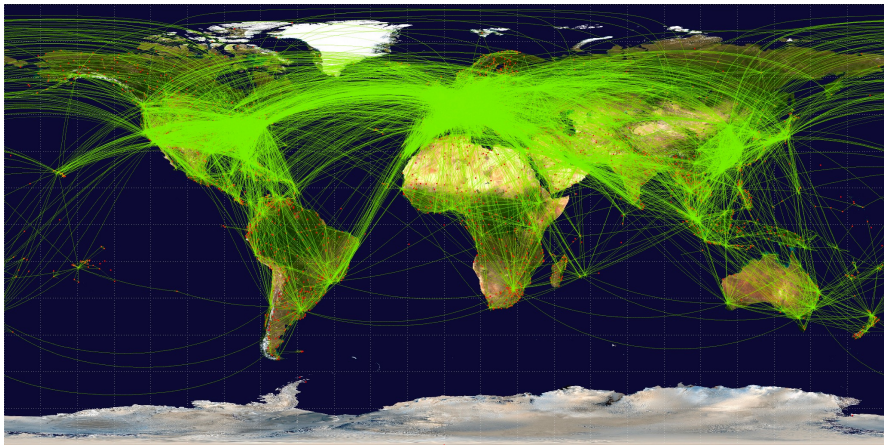




https://commons.wikimedia.org/wiki/File:Protein_Interaction_Network_for_TMEM8A.png

# Graph Applications



Connectomics



Transportation

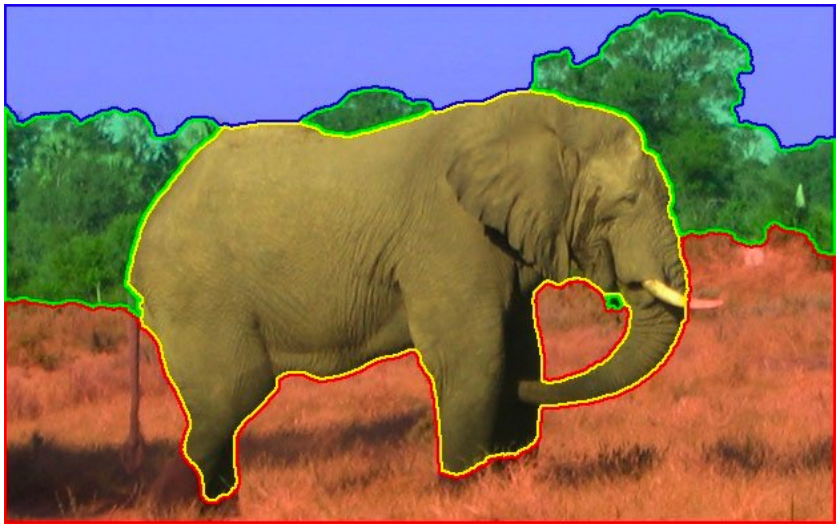
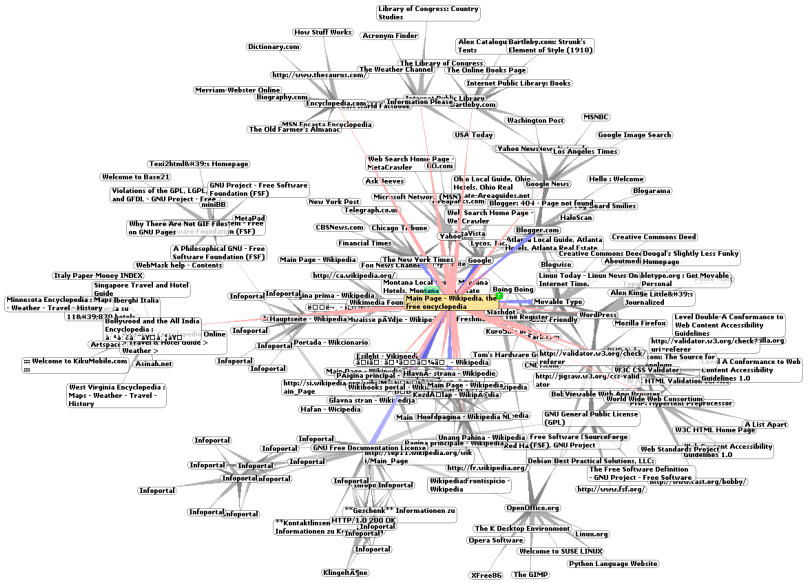


Image Segmentation



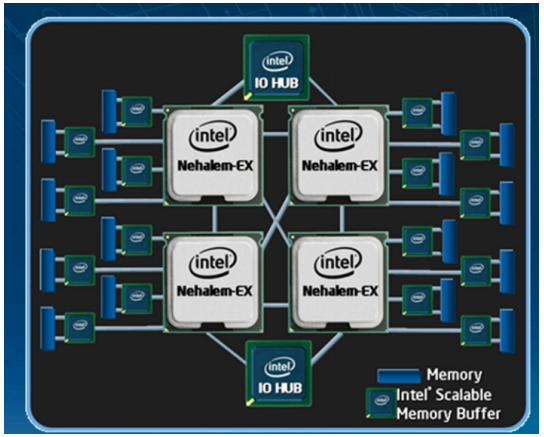
Internet

# Graphs are becoming very large

| | Size | |
|---|---|---|
| Common Crawl | 3.5 billion vertices<br>128 billion edges | *Largest publicly available graph* |
| YAHOO! | 272 billion vertices<br>5.9 trillion edges | *Proprietary graph* |
| Google | > 100 billion vertices<br>6 trillion edges | *Proprietary graph* |

*Need high-performance solutions to analyze graphs in a timely fashion*

# High Performance via Parallelism



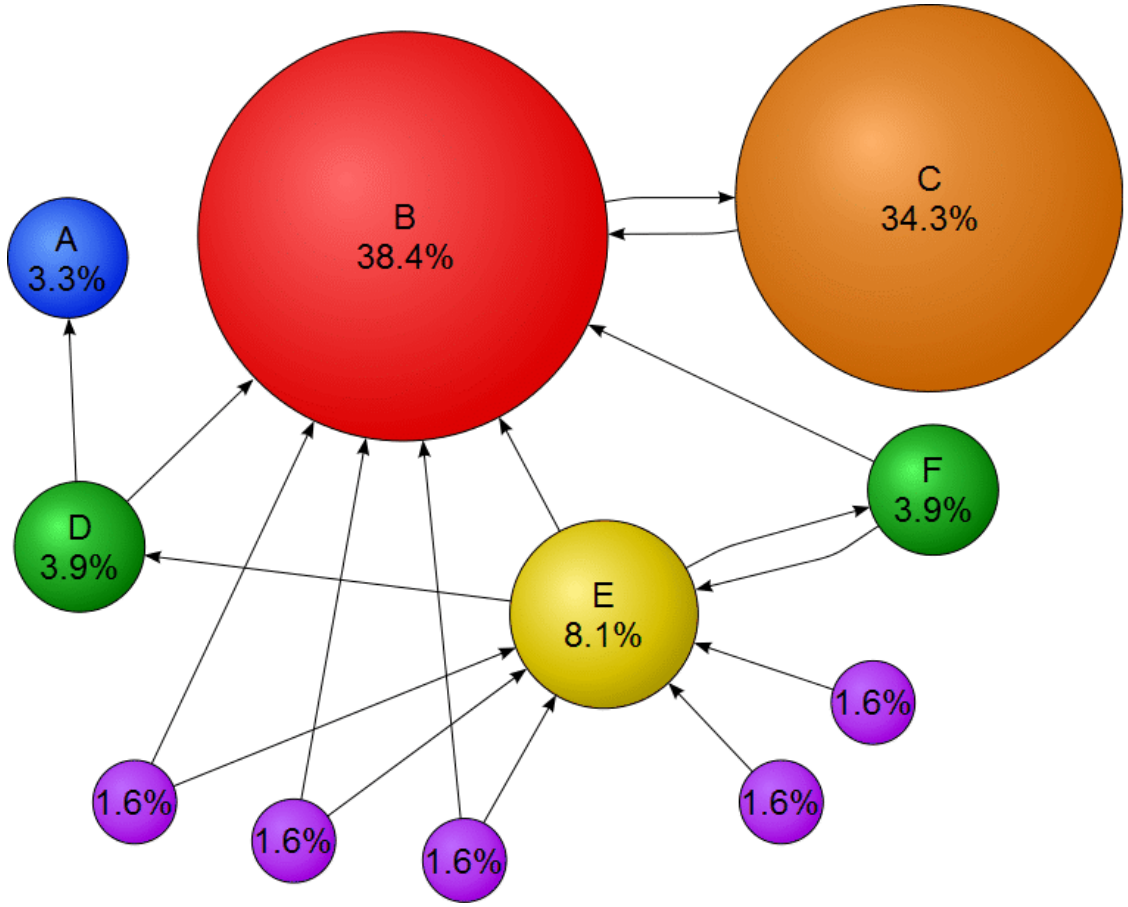

- Need parallel graph algorithms and codes
- Need high-level programming frameworks to enable non-experts to easily write their own parallel graph programs

# Outline

- GraphIt: graph DSL that enables easy exploration of large graph optimization space

- Aspen: streaming graph framework that enables concurrent queries and updates with low latency

# PageRank




$$\mathrm{PR}[v] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{\mathrm{PR}[u]}{deg^+(u)}$$

# Simple PageRank Code in C++

```
void pagerank(Graph &graph, double * new_rank, double * old_rank, int * out_degree, int max_iter){
    for (i = 0; i < max_iter; i++) {
        for (src : graph.vertices()) {
            for (dst : graph.getOutgoingNeighbors(node)) {
                new_rank[dst] += old_rank[src]/out_degree[src]; } }
        for (node : graph.vertices()) {
            new_rank[node] = base_score + damping*new_rank[node]; }
        swap (old_rank, new_rank);
    }
}
```

# High-Performance PageRank Code

```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (beta_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

## More than 23x faster

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores.

**Multi-Threaded**

**Load Balanced**

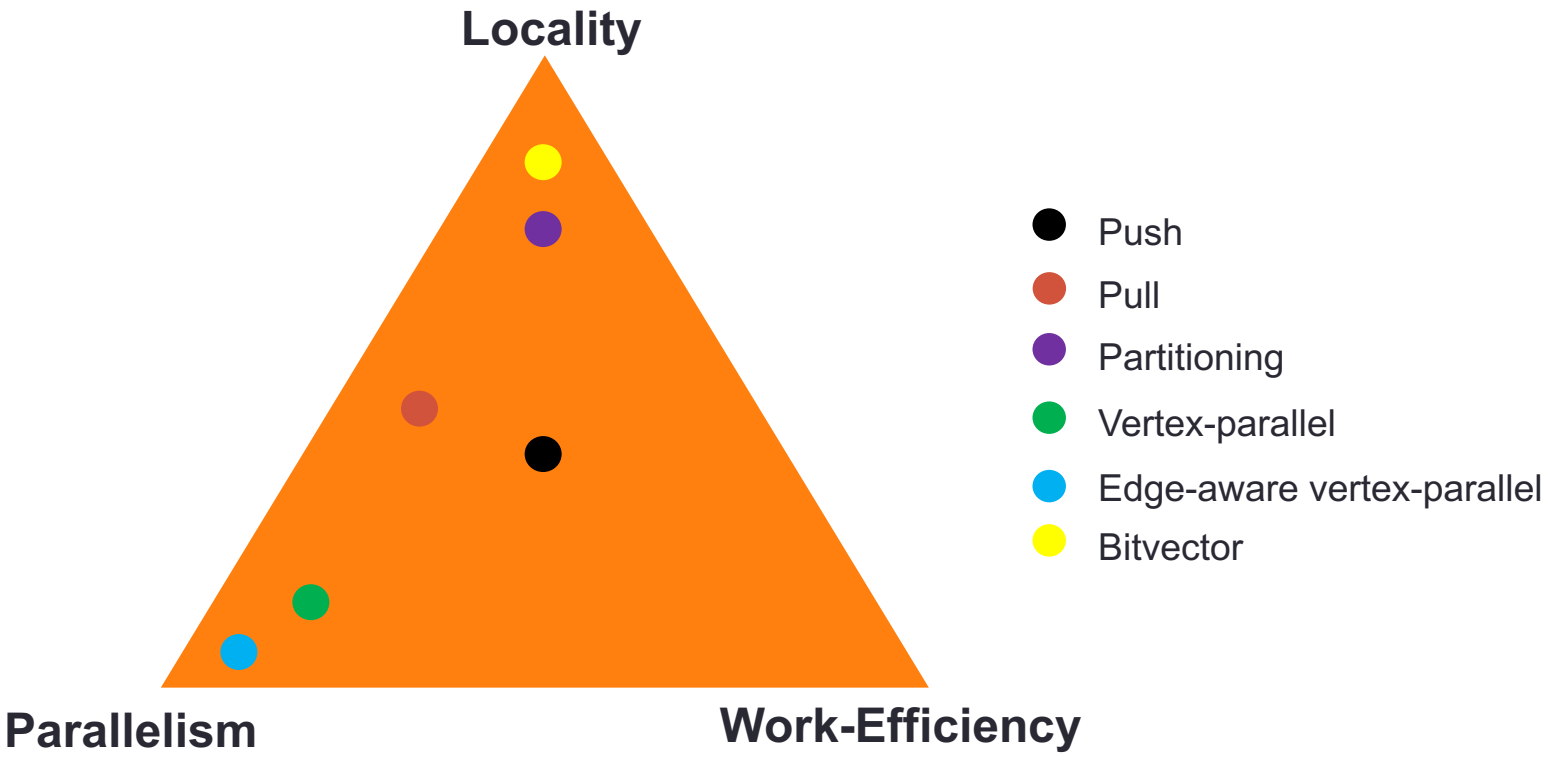
**Cache Optimized**

**NUMA Optimized**

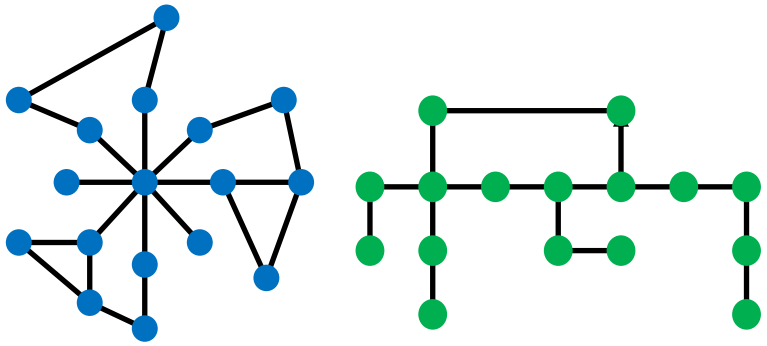
(1) Hard to write correctly

(2) Extremely difficult to experiment with different combinations of optimizations

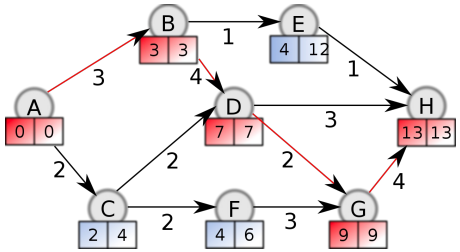
# Graph Optimization Tradeoff Space

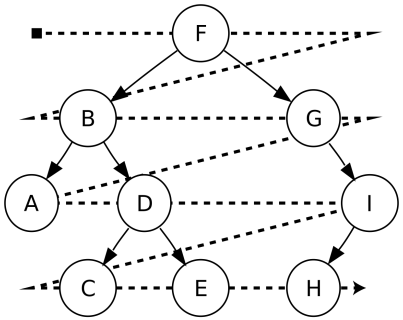


# Graph Optimization Tradeoff Space

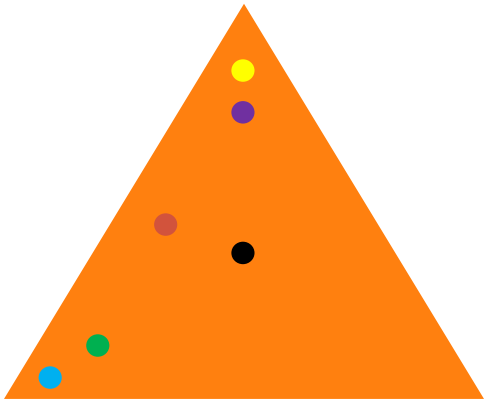


Graphs




Algorithms



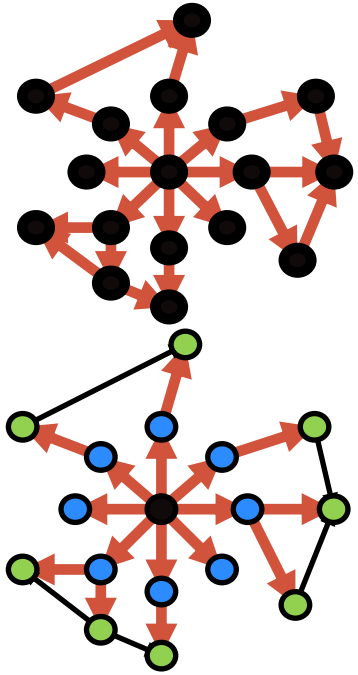
Optimizations

Hardware

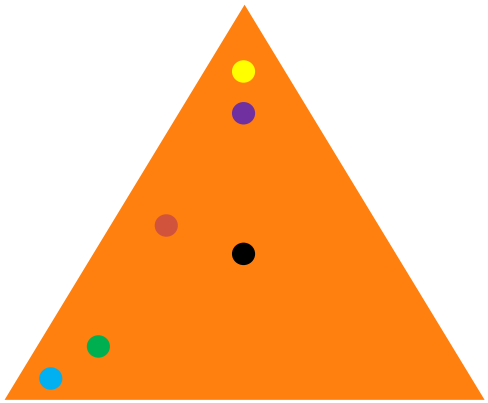
# GraphIt – A Domain-Specific Language for Graph Analytics

- Decouple Algorithm from Optimization
  - Algorithm language: What to compute
  - Optimization (scheduling) language: How to compute
- Compiler will generate optimized code for desired platform
  - Easy to use for programmers to try different combinations of optimizations

# GraphIt – A Domain-Specific Language for Graph Analytics



**Algorithm Language**



**Optimization Representation**

- Scheduling Language
- Schedule Representation (i.e., Graph Iteration Space)
- Compiler

**Autotuner**

# GraphIt Algorithm Language

- Data structures for representing sets of vertices and sets of edges
- Operators
  - Function for mapping computation over a set of vertices
  - Function for mapping computation over edges associated with a set of vertices
- No low-level implementation details (atomics, deduplication) needed
- Exposes plenty of opportunities for optimization

# PageRank in GraphIt

**Algorithm Specification**

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
func main()
    for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
end
```

# GraphIt Scheduling Language – Schedule 1

**Algorithm Specification**

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
func main()
    for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
end
```

**Pseudo Generated Code**

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  +=  old_rank[src] / out_degree[src];
   }
}

….
```

**Scheduling Functions**

```
schedule:
  program->configApplyDirection("s1", "SparsePush");
```

# GraphIt Scheduling Language – Schedule 2

**Algorithm Specification**

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
func main()
    for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
end
```

**Pseudo Generated Code**

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
      atomic_add (new_rank[dst],
          old_rank[src] / out_degree[src] );
   }
}

….
```

**Scheduling Functions**

```
schedule:
  program->configApplyDirection("s1", "SparsePush");
  program->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

# GraphIt Scheduling Language – Schedule 3

**Algorithm Specification**

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
func main()
    for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
end
```

**Pseudo Generated Code**

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID dst : vertices) {
  for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
   }
}

….
```

**Scheduling Functions**

```
schedule:
  program->configApplyDirection("s1", "DensePull");
  program->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

# GraphIt Scheduling Language – Schedule 4

**Algorithm Specification**

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
func main()
    for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
end
```

**Pseudo Generated Code**

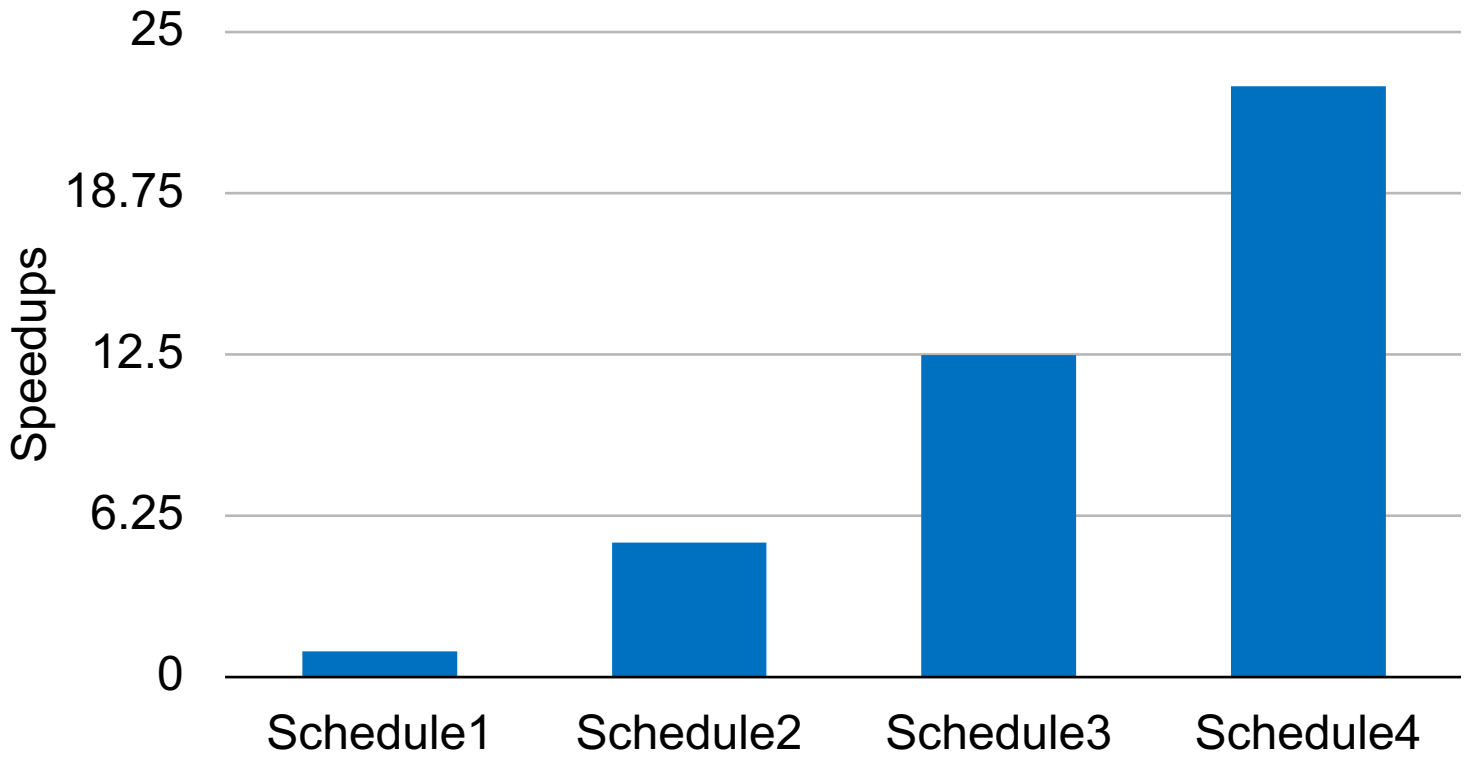
```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…
for (Subgraph sg : G.subgraphs) {
  parallel_for (NodeID dst : vertices) {
    for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
    }
  }
}
….
```

**Scheduling Functions**

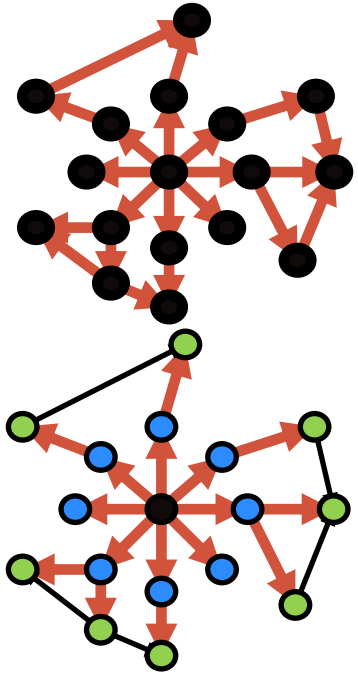
```
schedule:
  program->configApplyDirection("s1", "DensePull");
  program->configApplyParallelization("s1", "dynamic-vertex-parallel");
  program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

# Speedups of Schedules

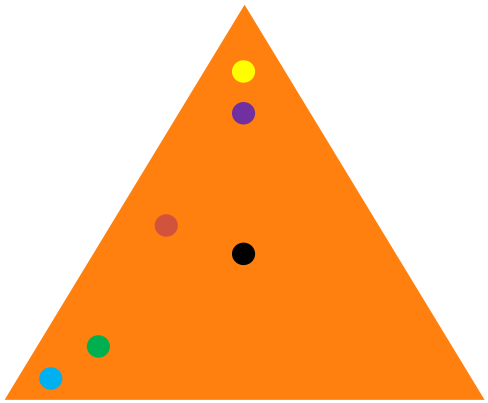



Twitter graph with 41M vertices and 1.5B edges
Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

# GraphIt – A Domain-Specific Language for Graph Analytics



**Algorithm Language**



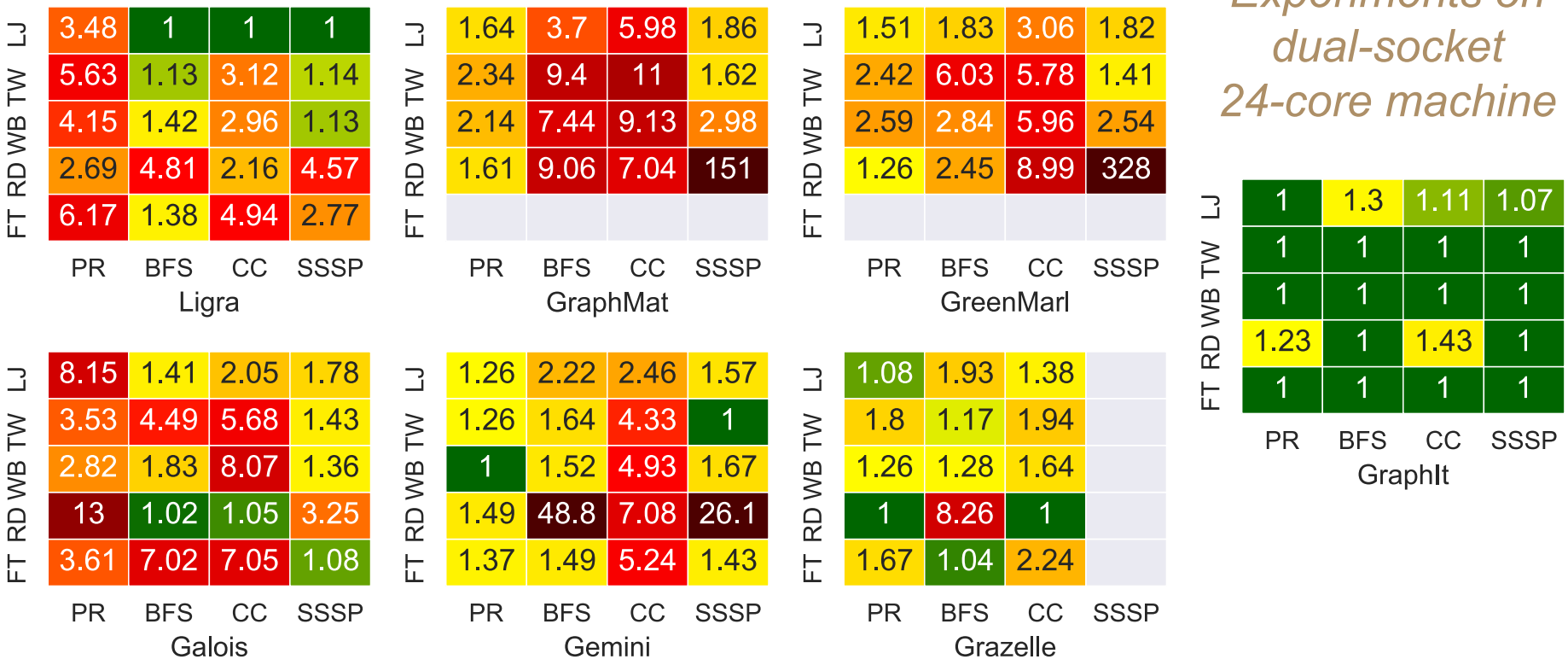
**Optimization Representation**

- Scheduling Language
- Schedule Representation (i.e., Graph Iteration Space)
- Compiler

**Autotuner**

# Multicore GraphIt Performance Results

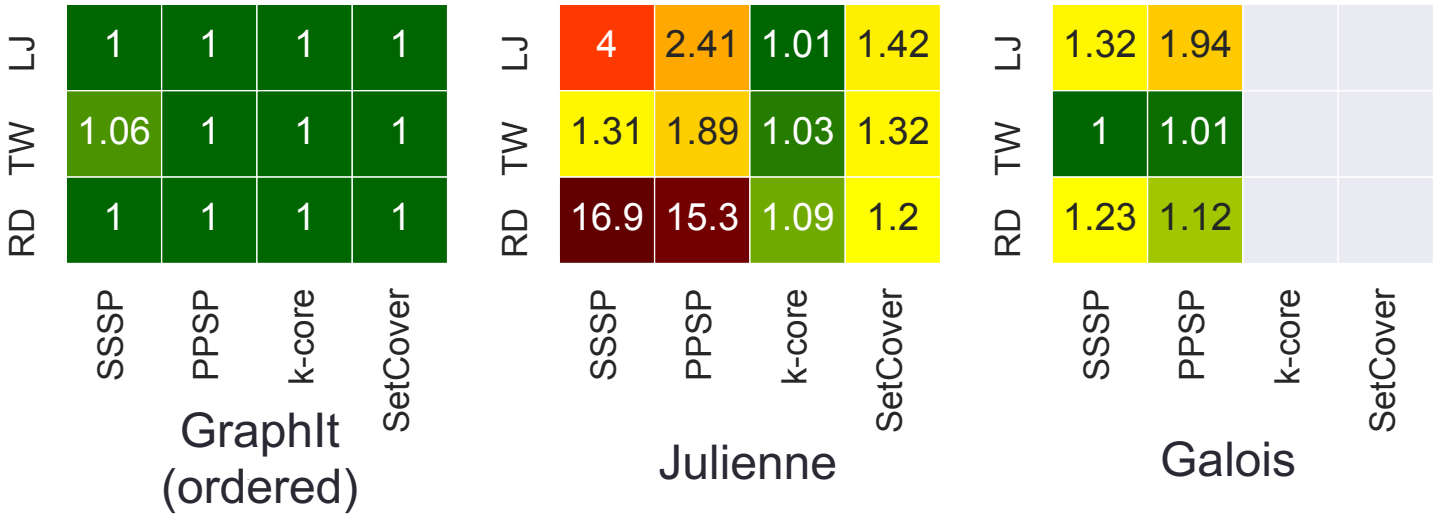
*Experiments on dual-socket 24-core machine*




- Slowdowns compared to fastest implementation (lower is better)
- By exploring orders of magnitude more schedules, GraphIt is able to achieve the best performance in most cases (up to 4.8x faster)
- By using a compiler approach, global optimizations (e.g., data structure transformation and kernel fusion) can easily be applied

# Ordered Graph Algorithms

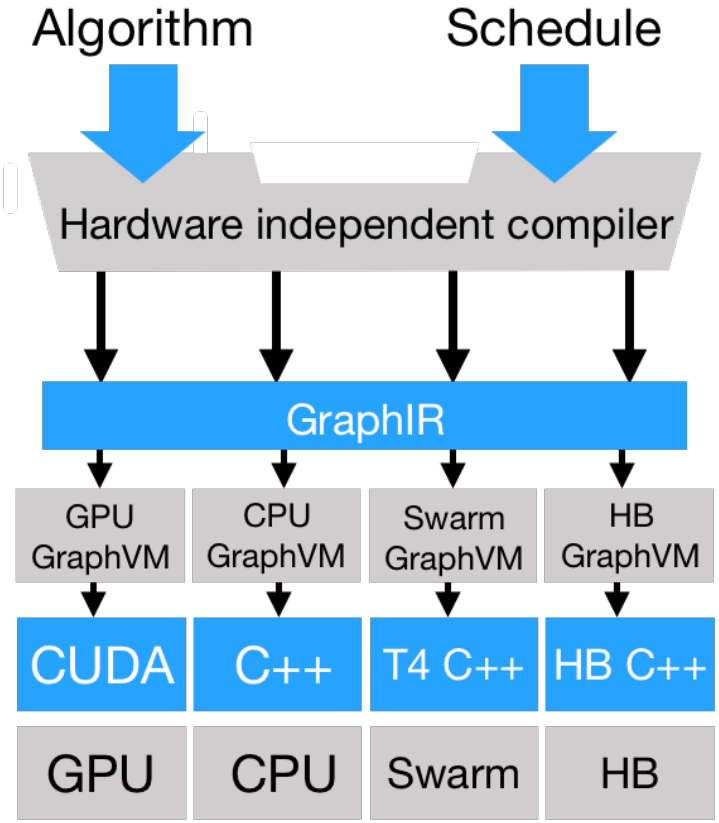
- Original GraphIt was designed for **unordered** graph algorithms, where vertices can be executed in any order per iteration
- **Ordered** graph algorithms, where priorities of vertices change dynamically, can achieve better work-efficiency
- We have extended GraphIt's algorithm and scheduling language to support ordered graph algorithms




*Experiments on dual-socket 24-core machine*

- Up to 2x faster than Galois and 17x faster than Julienne

# Achieving Portability with GraphIR




- **GraphIR** consists of hardware-independent compiler passes
- Extensible to new hardware backends (**GraphVMs**)
- GraphVMs have their own scheduling languages inherited from a base scheduling language

# Outline

- GraphIt: graph DSL that enables easy exploration of large graph optimization space

- **Aspen: streaming graph framework that enables concurrent queries and updates with low latency**

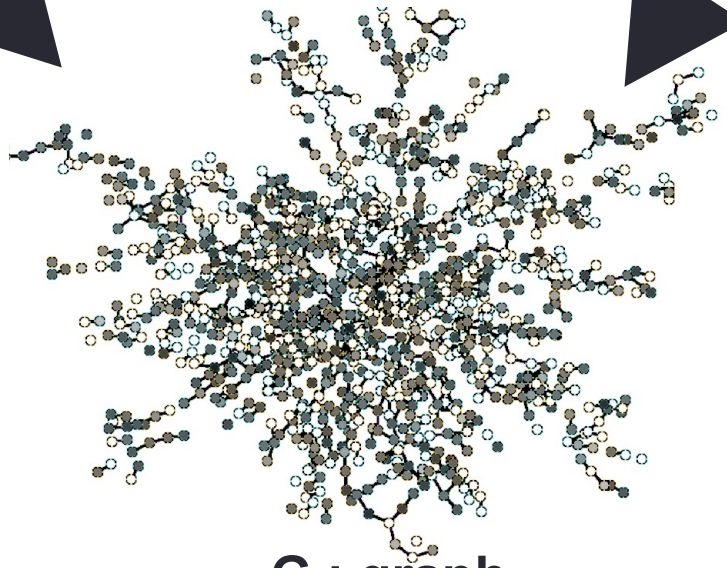
# Streaming Graph Processing

...

...
insert($e_4$)

delete($e_3$)
insert($e_2$)
insert($e_1$)

**Update Stream**



**G : graph**

...

ClusteringCoefficient(v)

Connectivity(G)

Reachable(u—>v)?

**Query Stream**

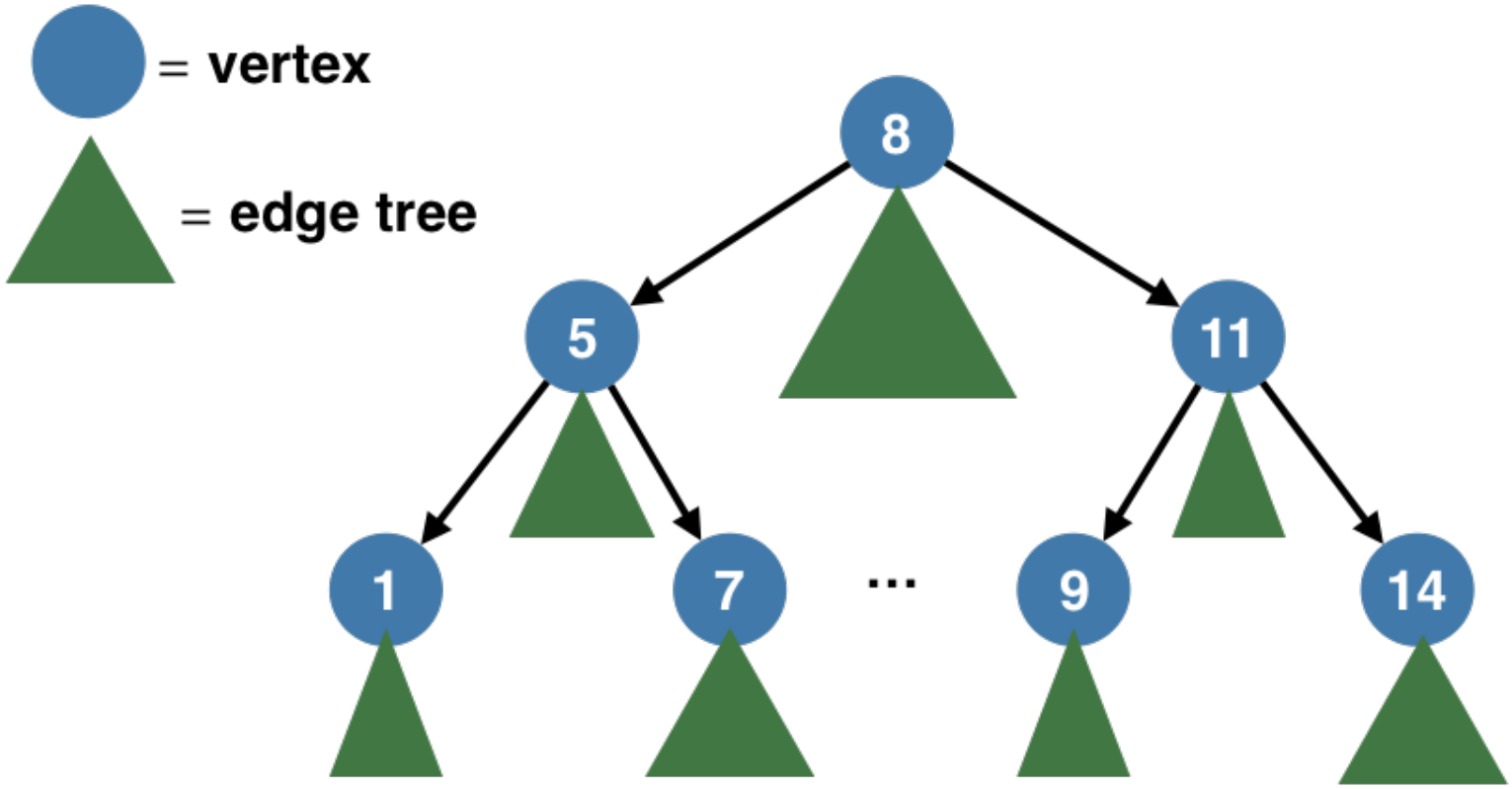
**Goals: Serializability for updates/queries, achieve low latency and high throughput**

# Existing Work

- Single Version Systems
  - Maintain a **single** version of the graph
  - Common approach in graph streaming (e.g., STINGER, cuSTINGER, and KickStarter)
  - Need to separate queries from updates for serializability
- Multi-Version Systems
  - Support multiple graph snapshots (e.g., LLAMA, Kineograph, and some graph databases)
  - Snapshots are not space-efficient and lead to high latency
- **Aspen** uses lightweight snapshots to enable low-latency concurrent queries and updates

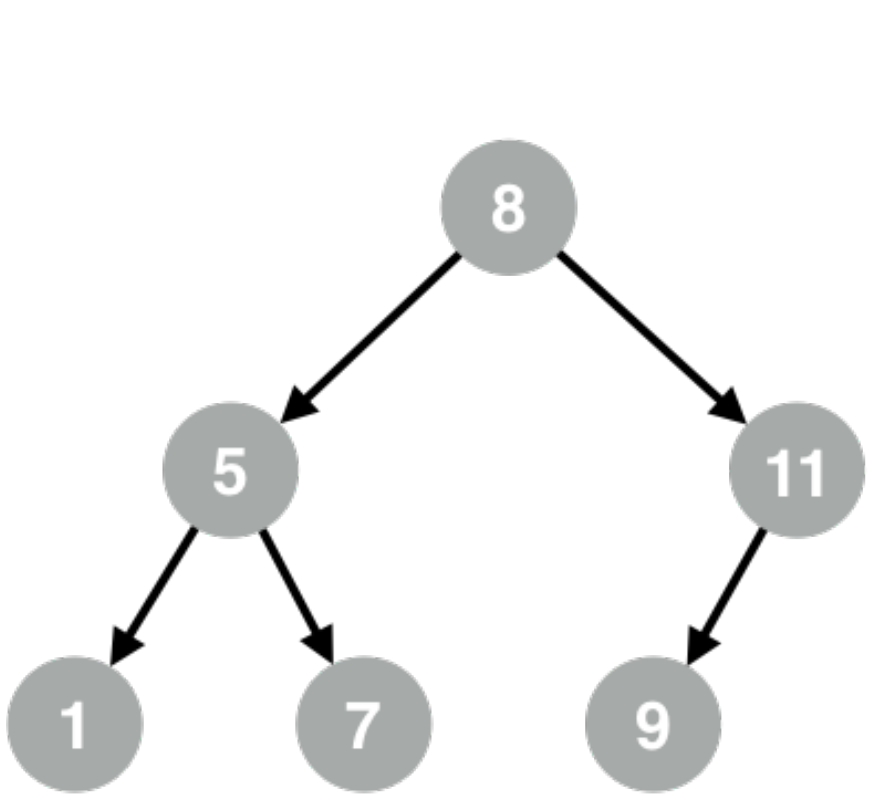
# Graphs Using Purely Functional Trees

- Purely functional trees can be updated efficiently (in logarithmic time/space) while retaining old copy of tree
- Aspen uses tree of **vertices**, where each vertex stores a tree of its incident **edges**



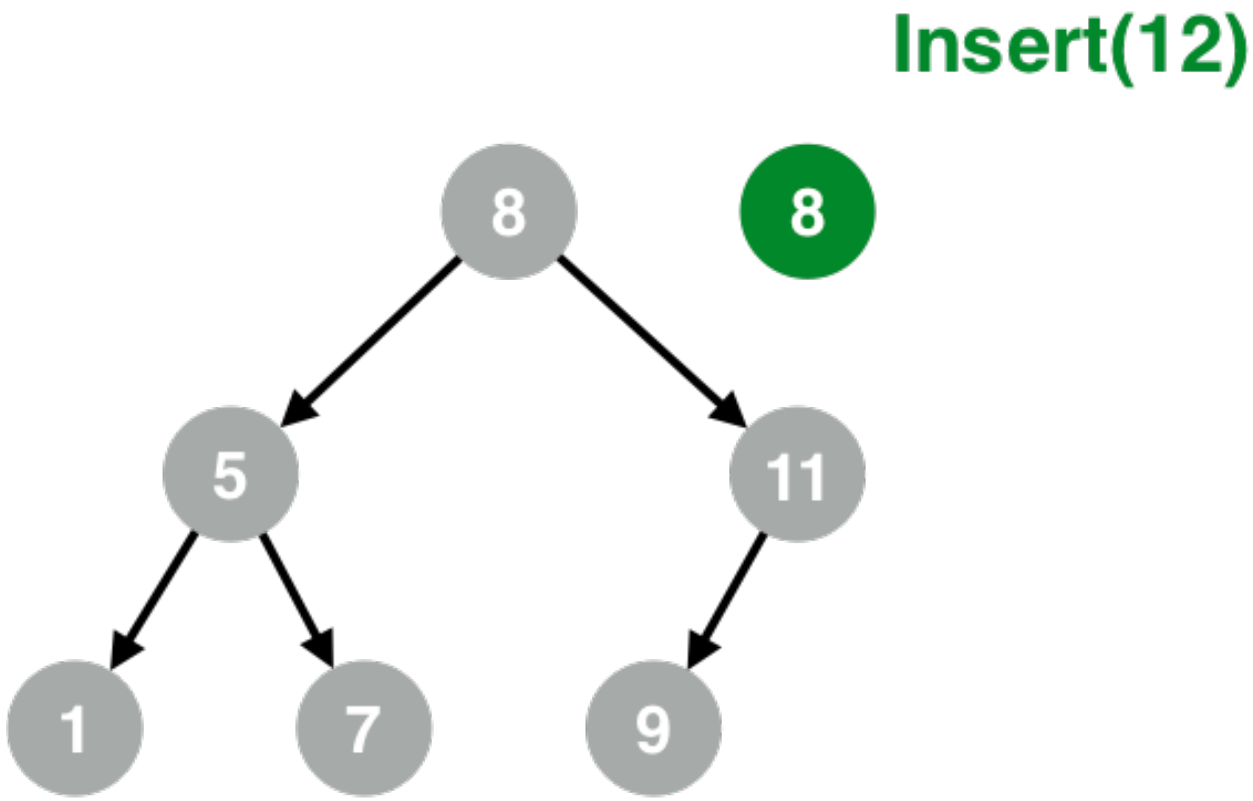
# Updates via Path Copying

- Easy to generate new versions via path copying



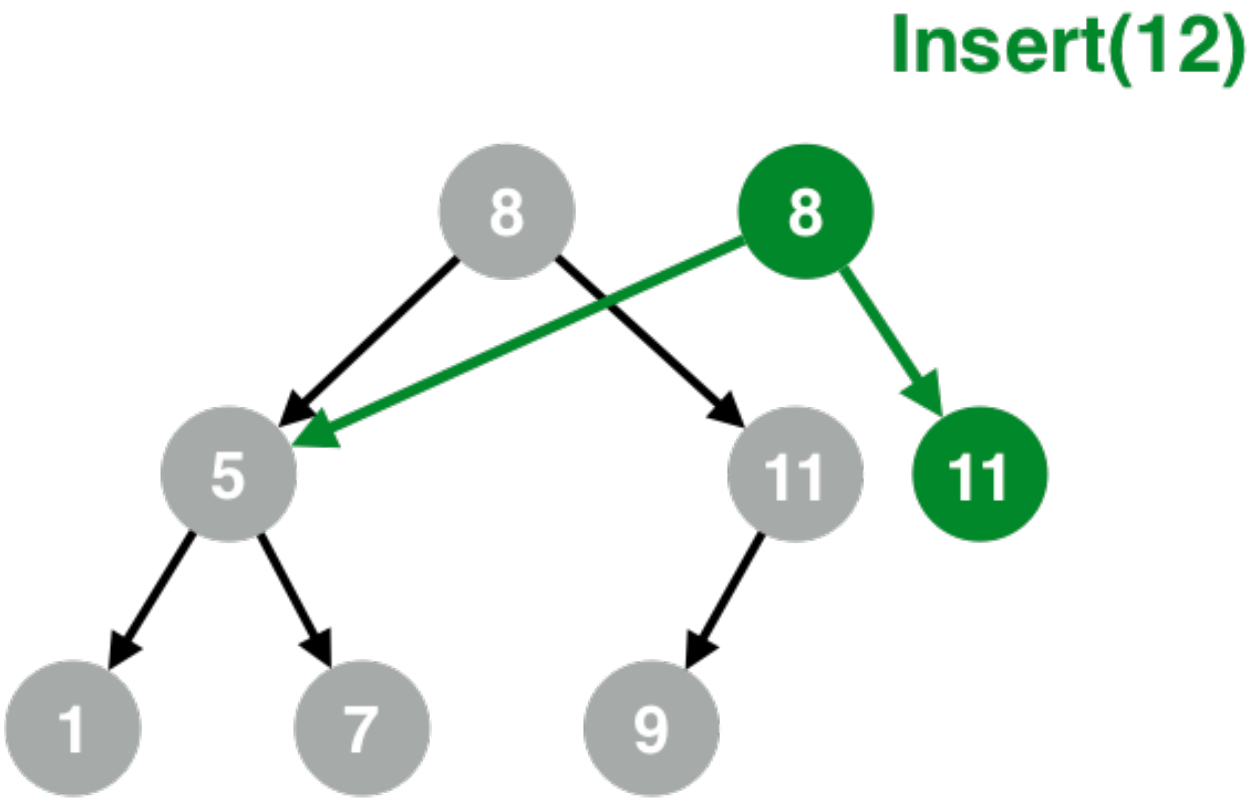
# Updates via Path Copying

- Easy to generate new versions via path copying



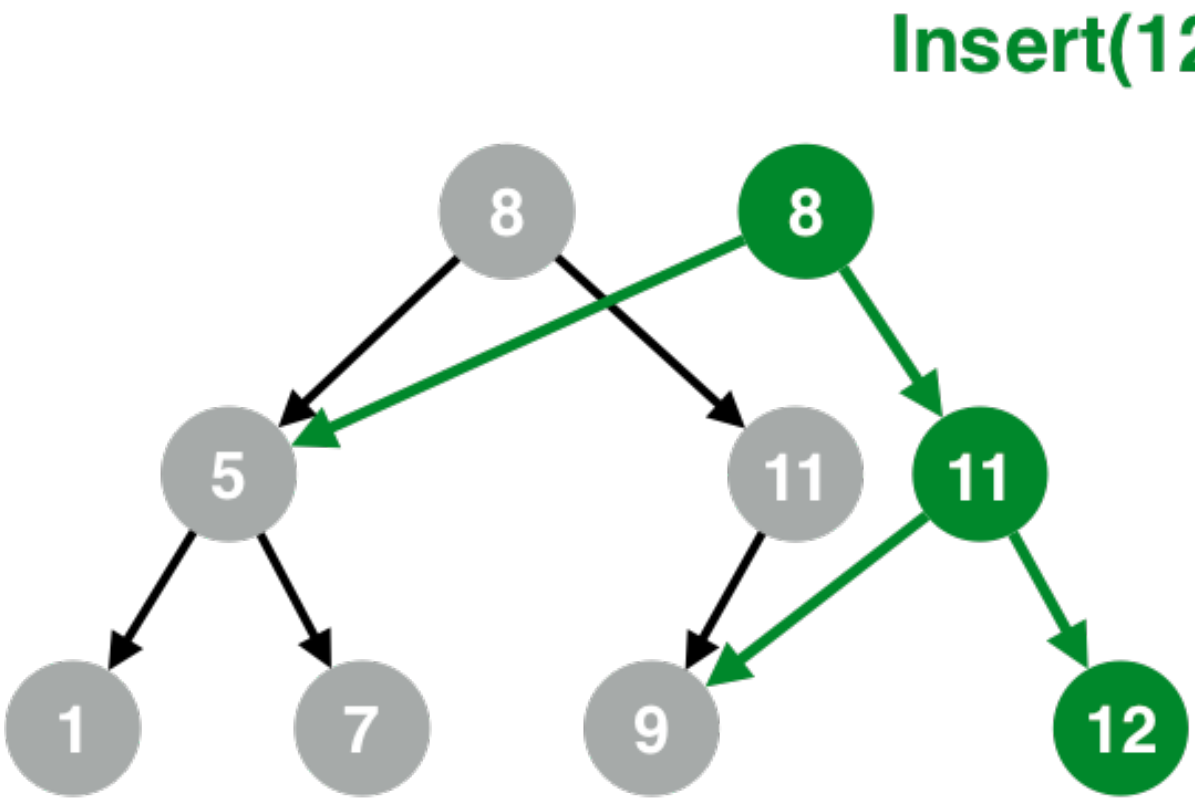
# Updates via Path Copying

- Easy to generate new versions via path copying

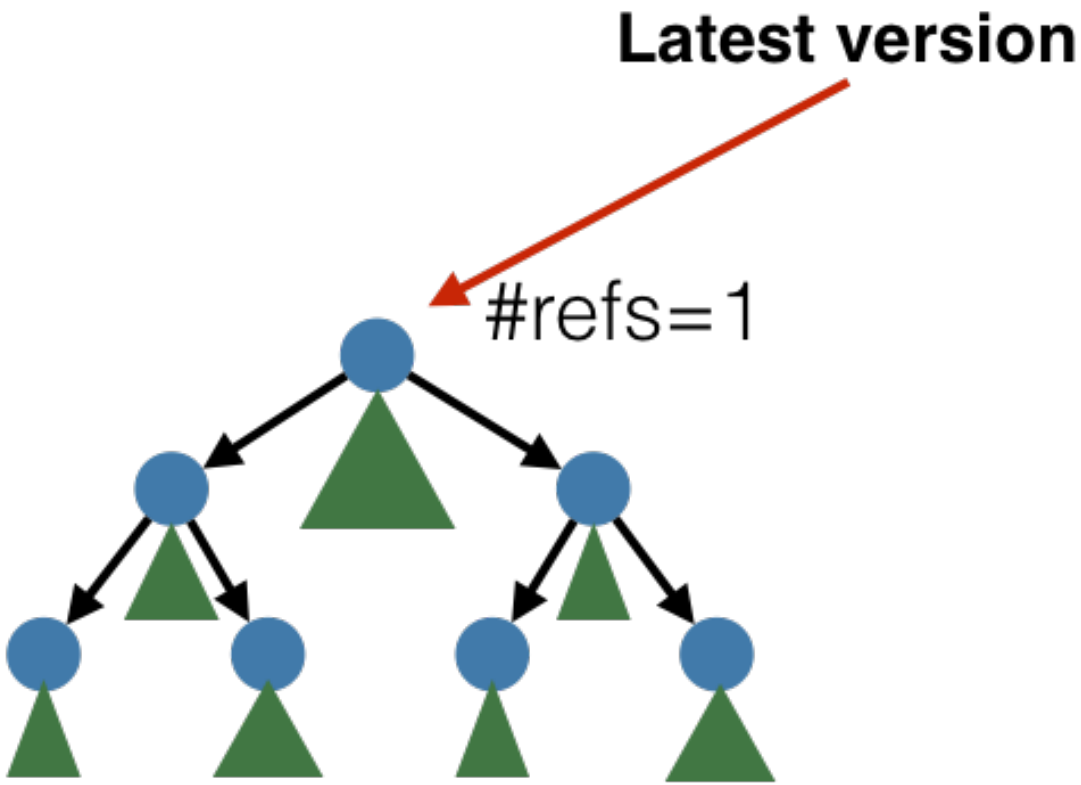


# Updates via Path Copying

- Easy to generate new versions via path copying



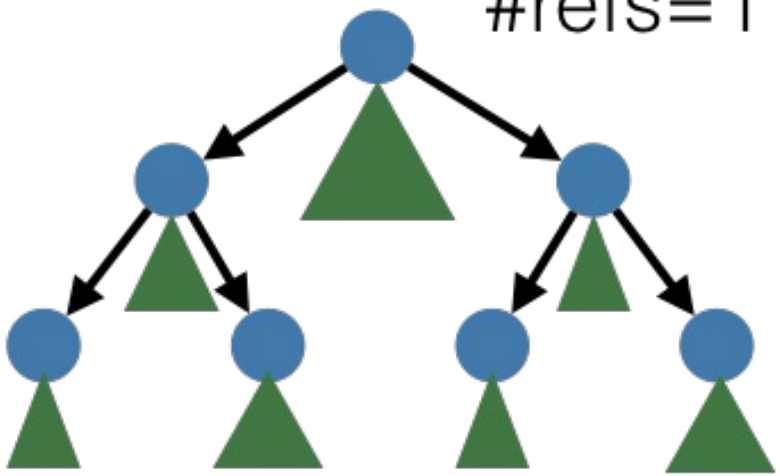
# Immutability Enables Concurrency



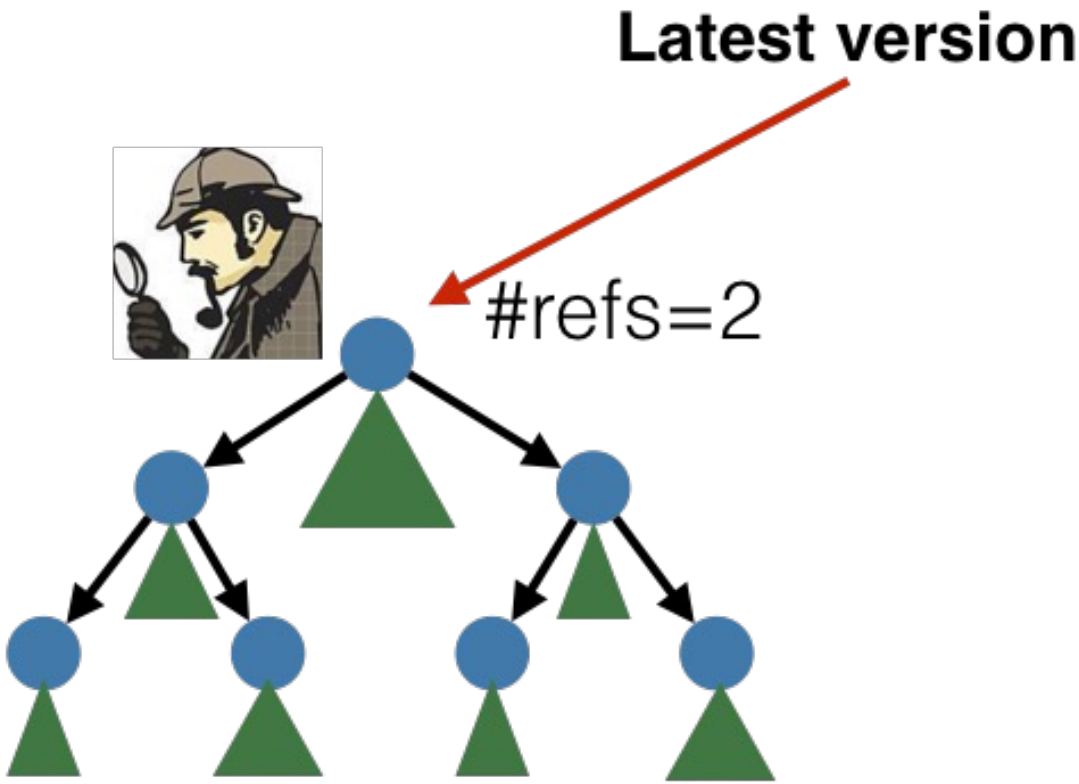
# Immutability Enables Concurrency

**Latest version**

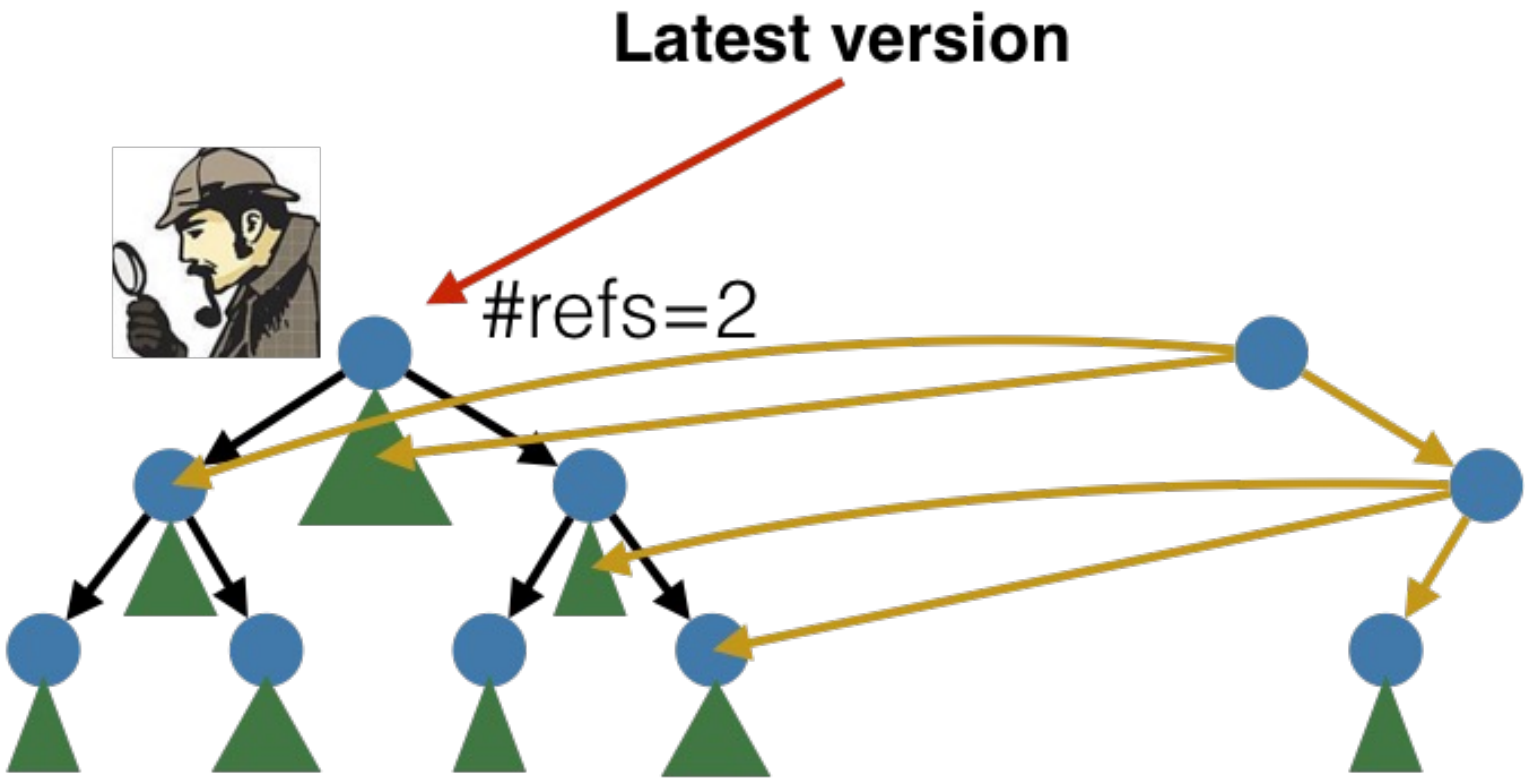
#refs=1



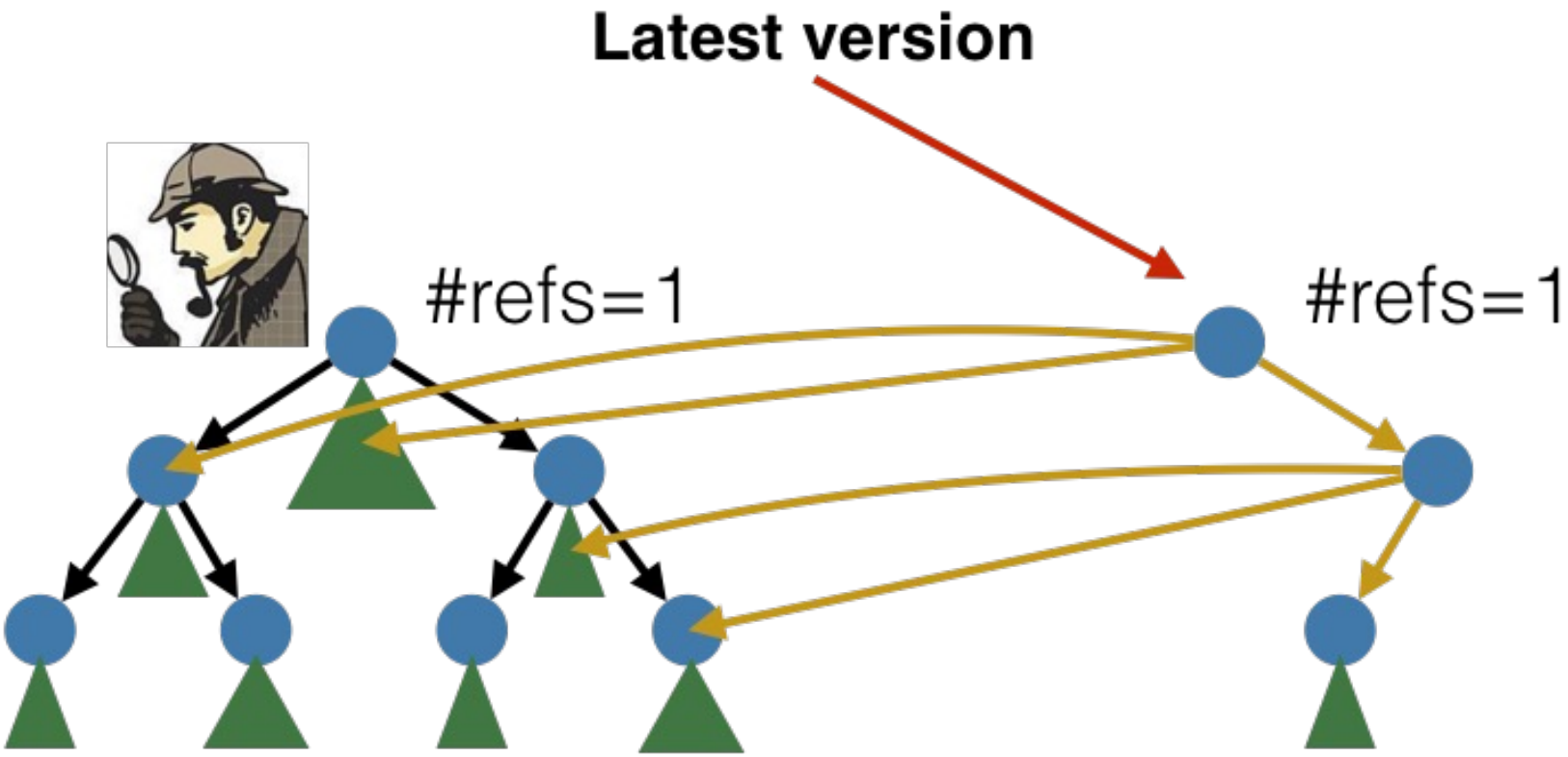
# Immutability Enables Concurrency



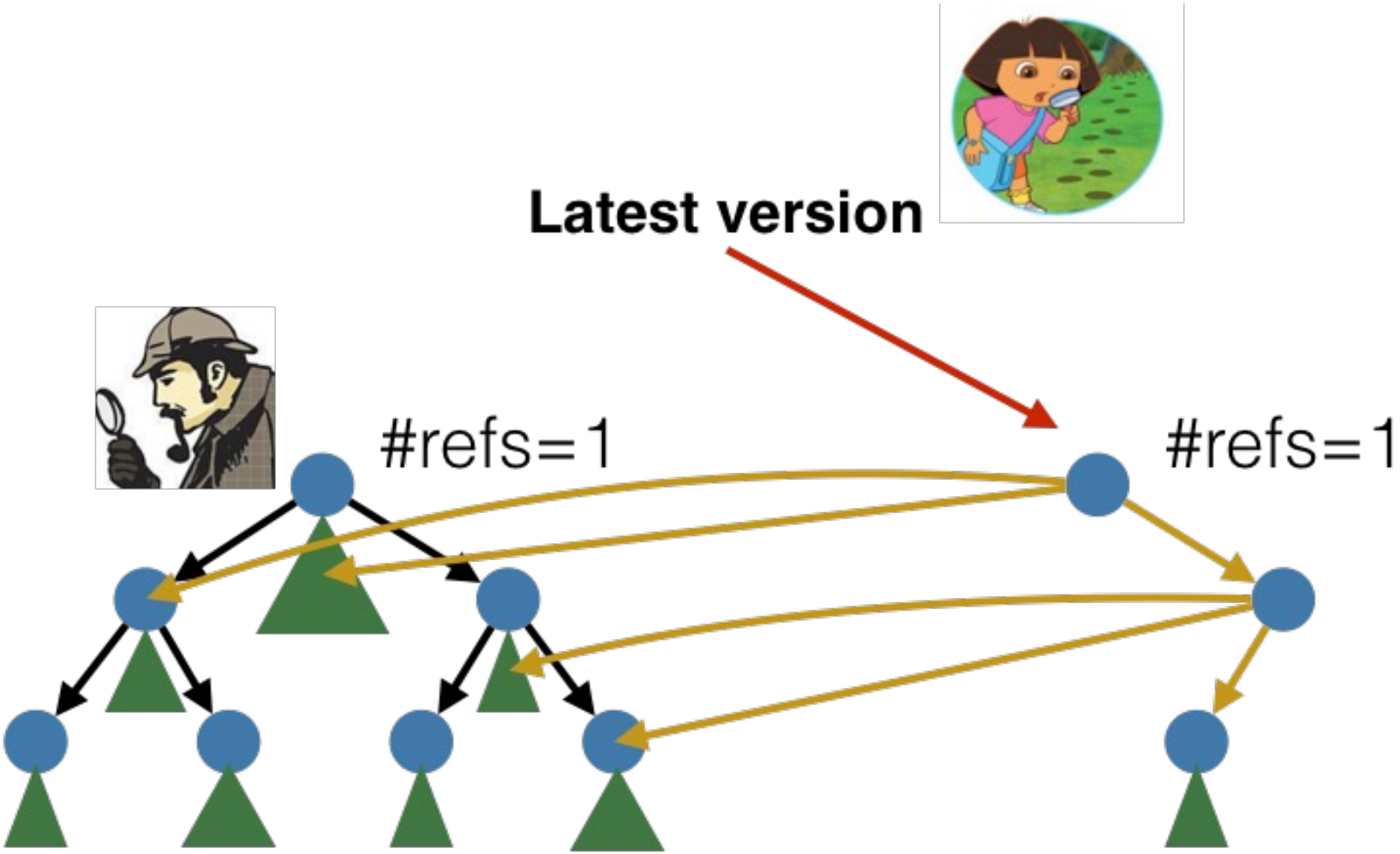
# Immutability Enables Concurrency



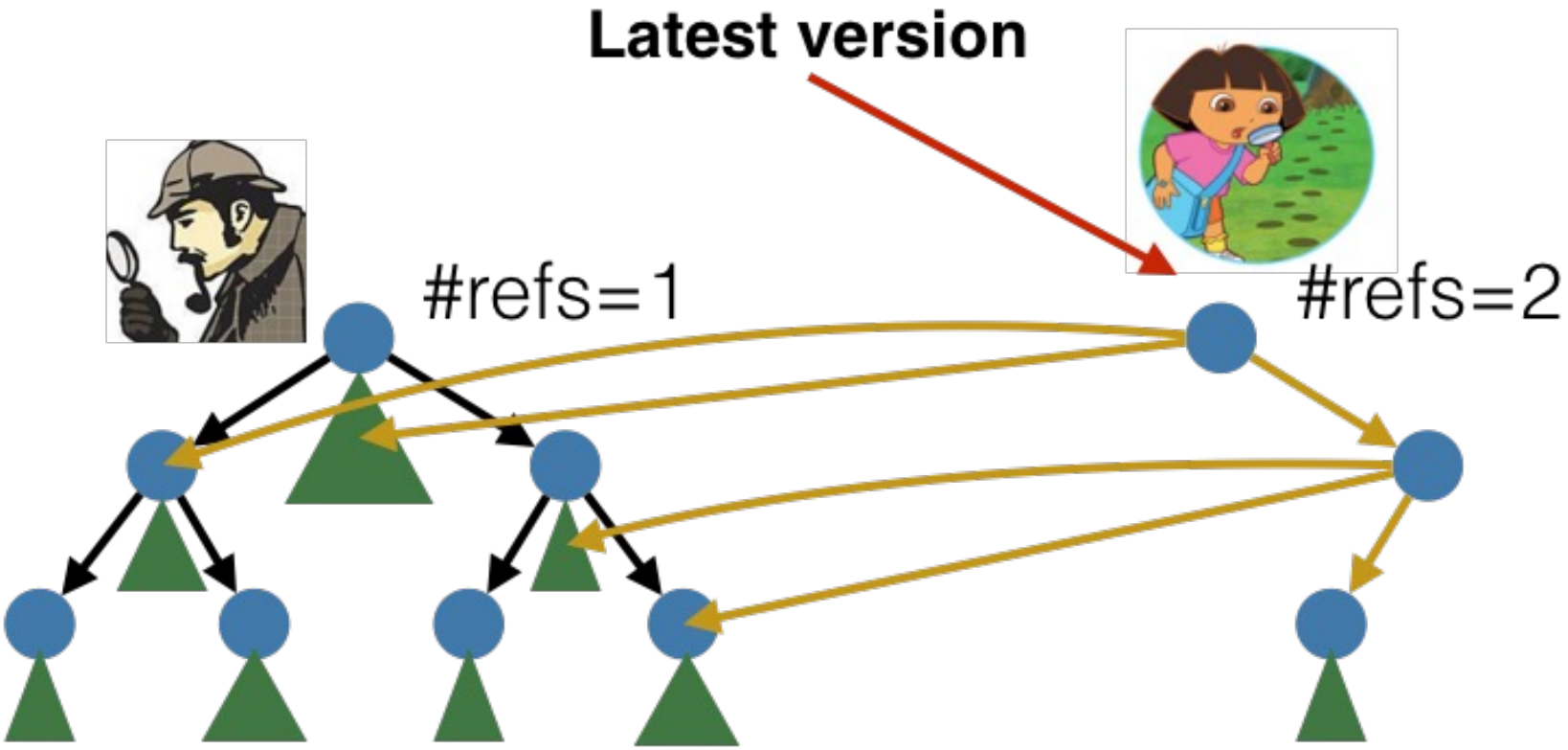
# Immutability Enables Concurrency



# Immutability Enables Concurrency

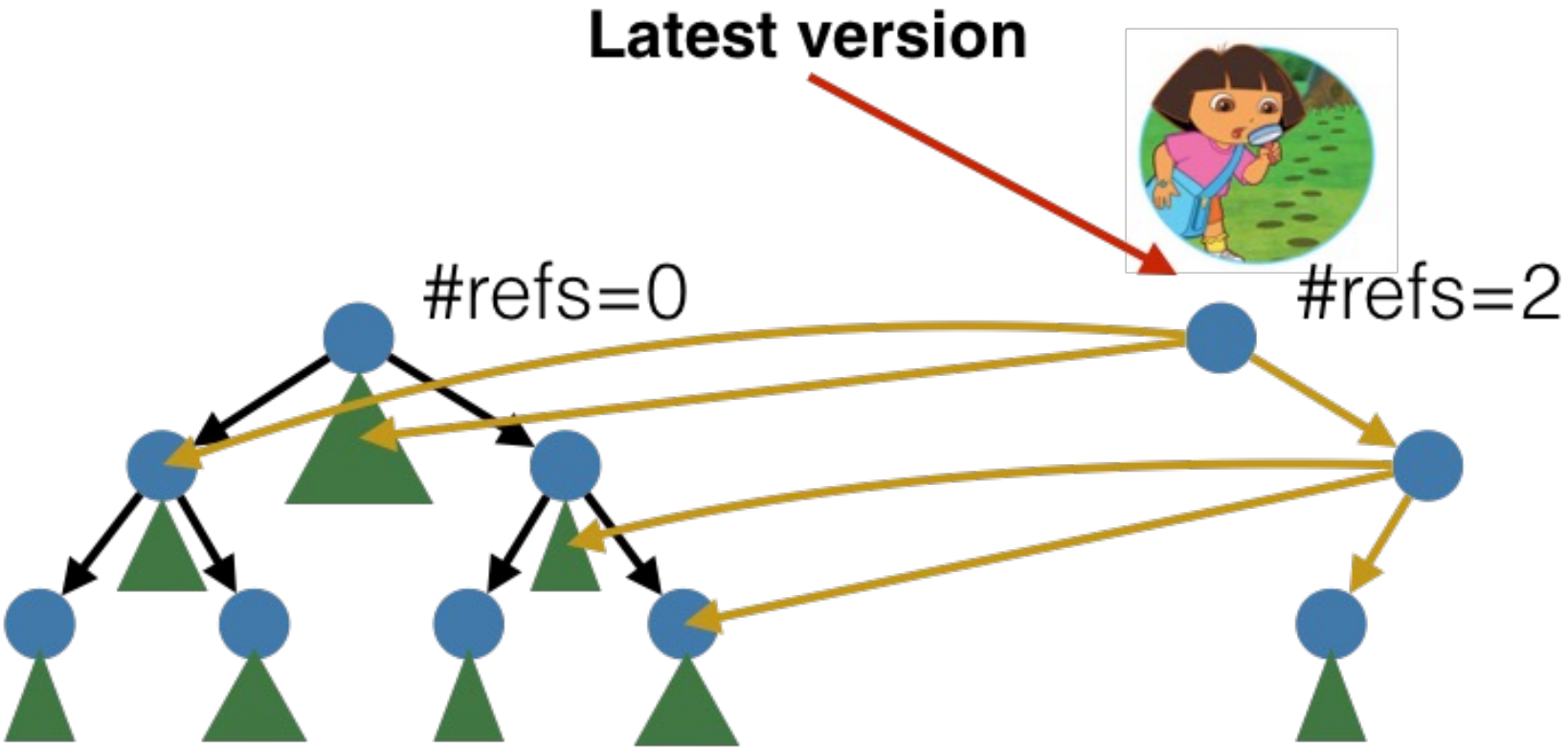


# Immutability Enables Concurrency



# Immutability Enables Concurrency

*Garbage collect all tree nodes whose reference count is decremented to 0*

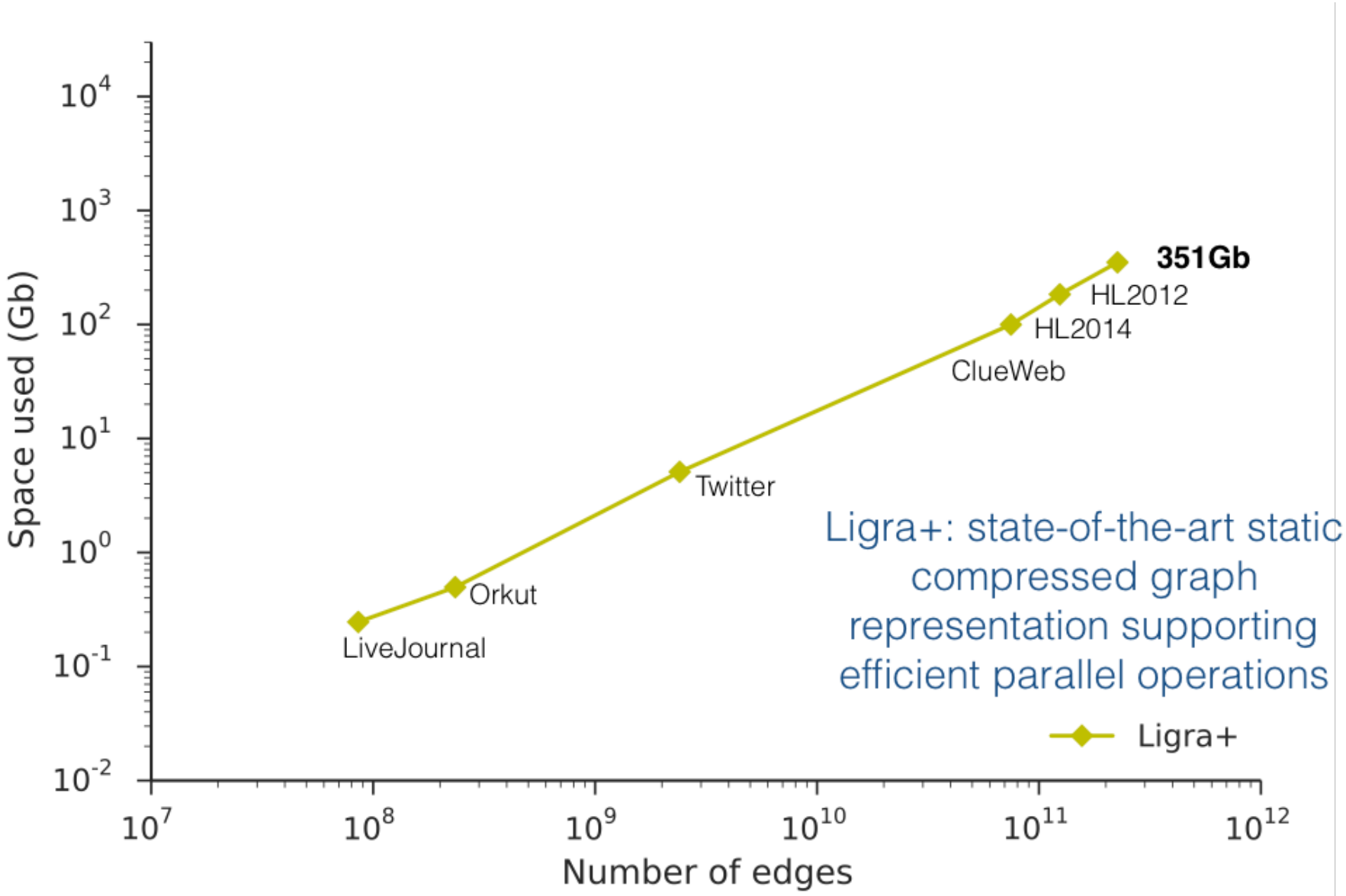


# Disadvantages of representing graphs using trees

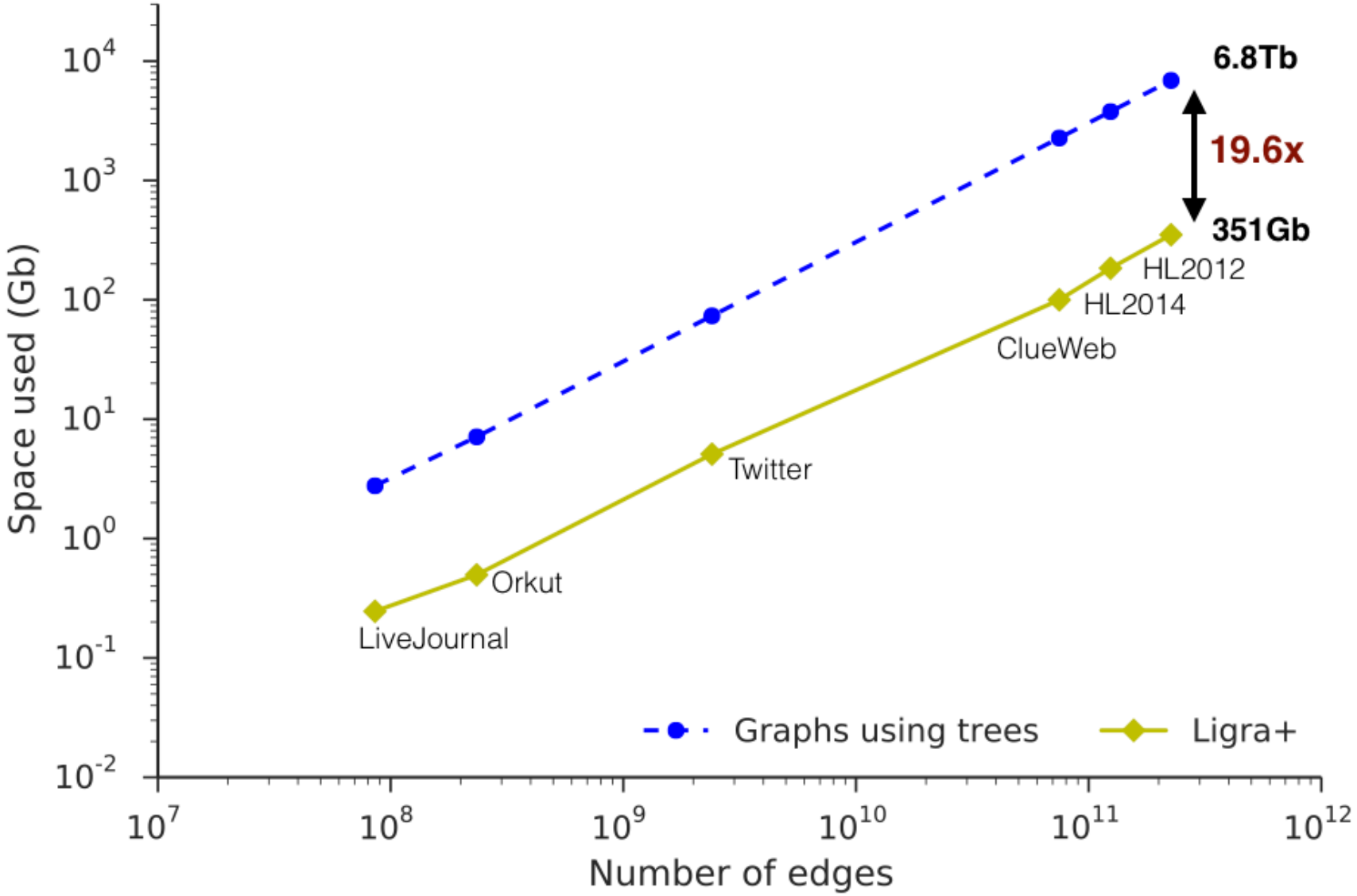
- Poor Cache Usage
  - One tree node per vertex and edge
  - One cache miss per edge access in the worst case
- Space Inefficiency
  - Need to store children pointers and metadata on tree nodes
  - Lose ability to perform integer compression

Requires close to 7TB of memory to store the symmetrized Common Crawl graph (225B edges)!

# Space Overhead of Graphs using Trees

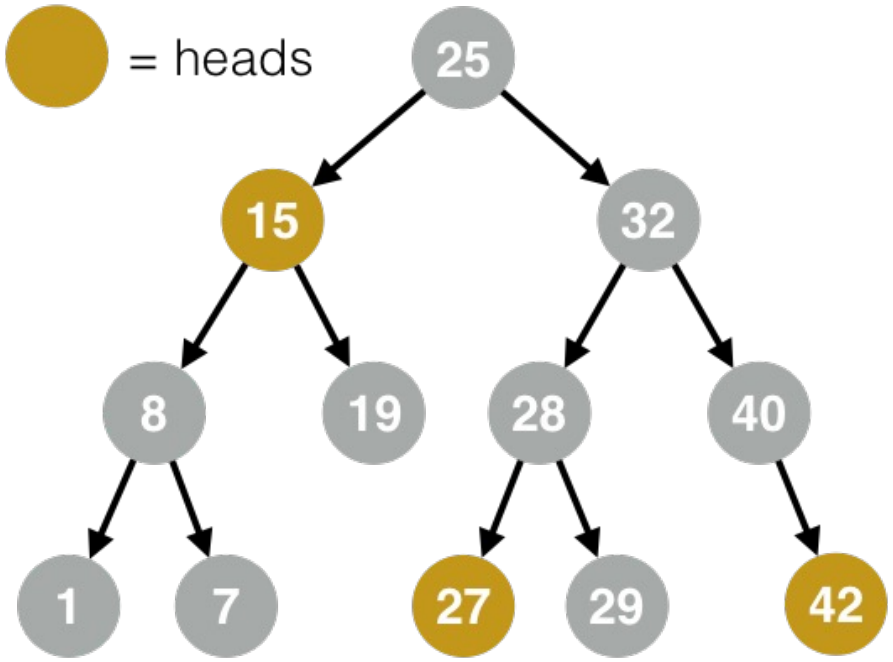


# Space Overhead of Graphs using Trees



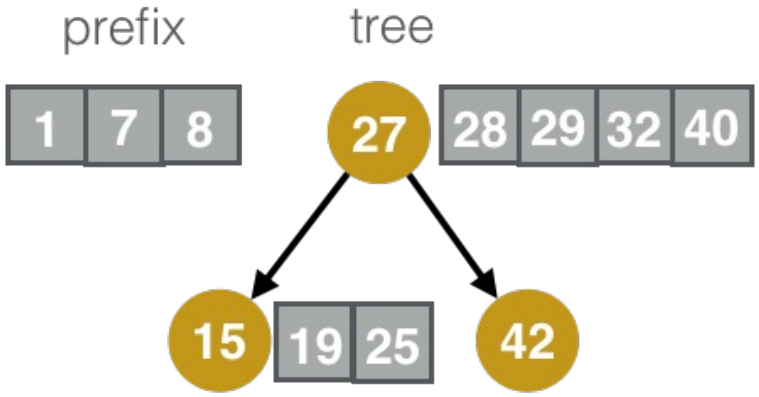
# C-tree

- Purely functional **compressed** tree data structure
- Chunking parameter = B. Fix a hash function *h*.
- Select elements as **heads** with probability **1/B** using *h*.




Tree

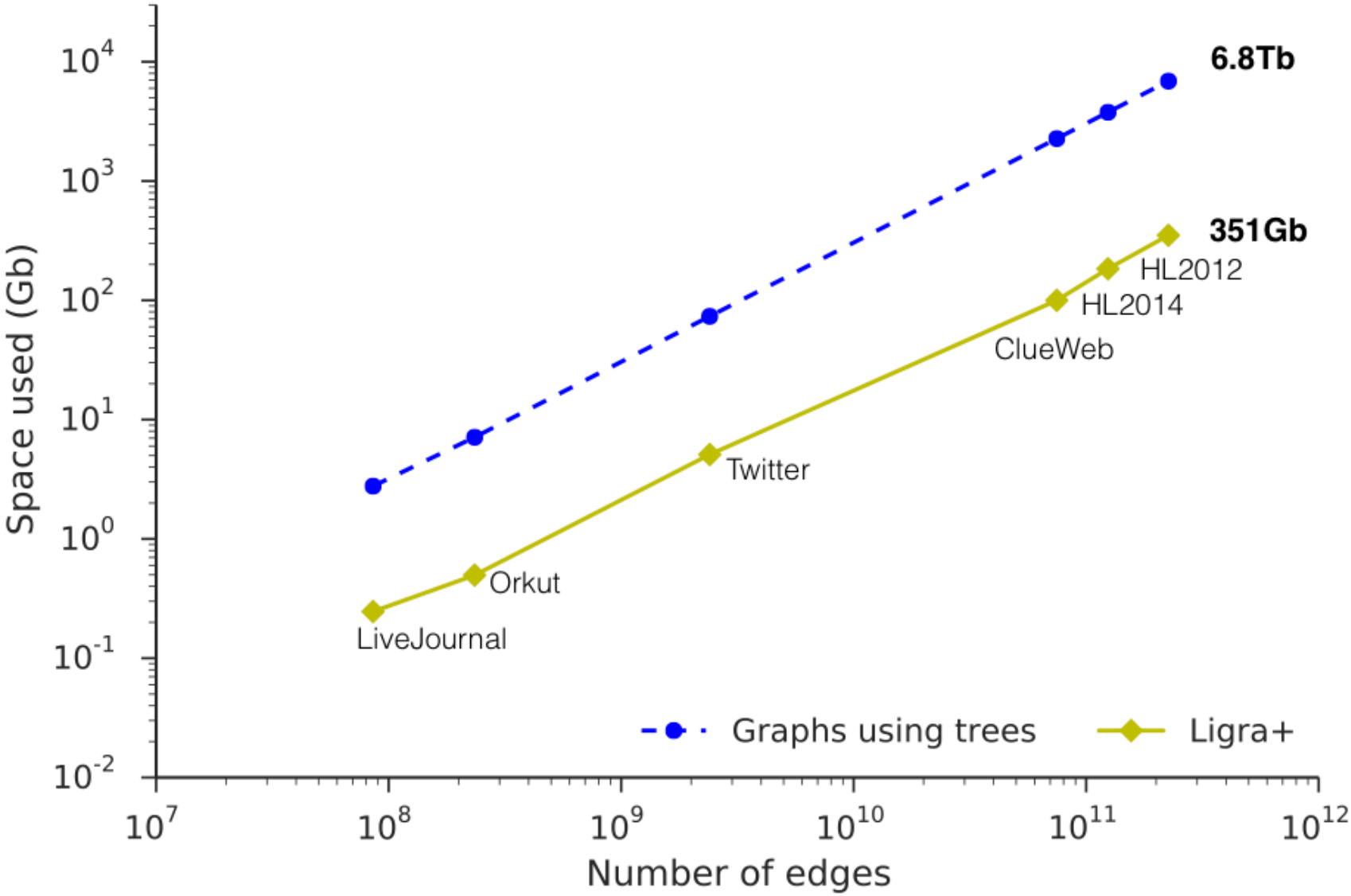
*More cache-efficient traversals*
*Further improve space usage for integer C-trees by difference encoding chunks*



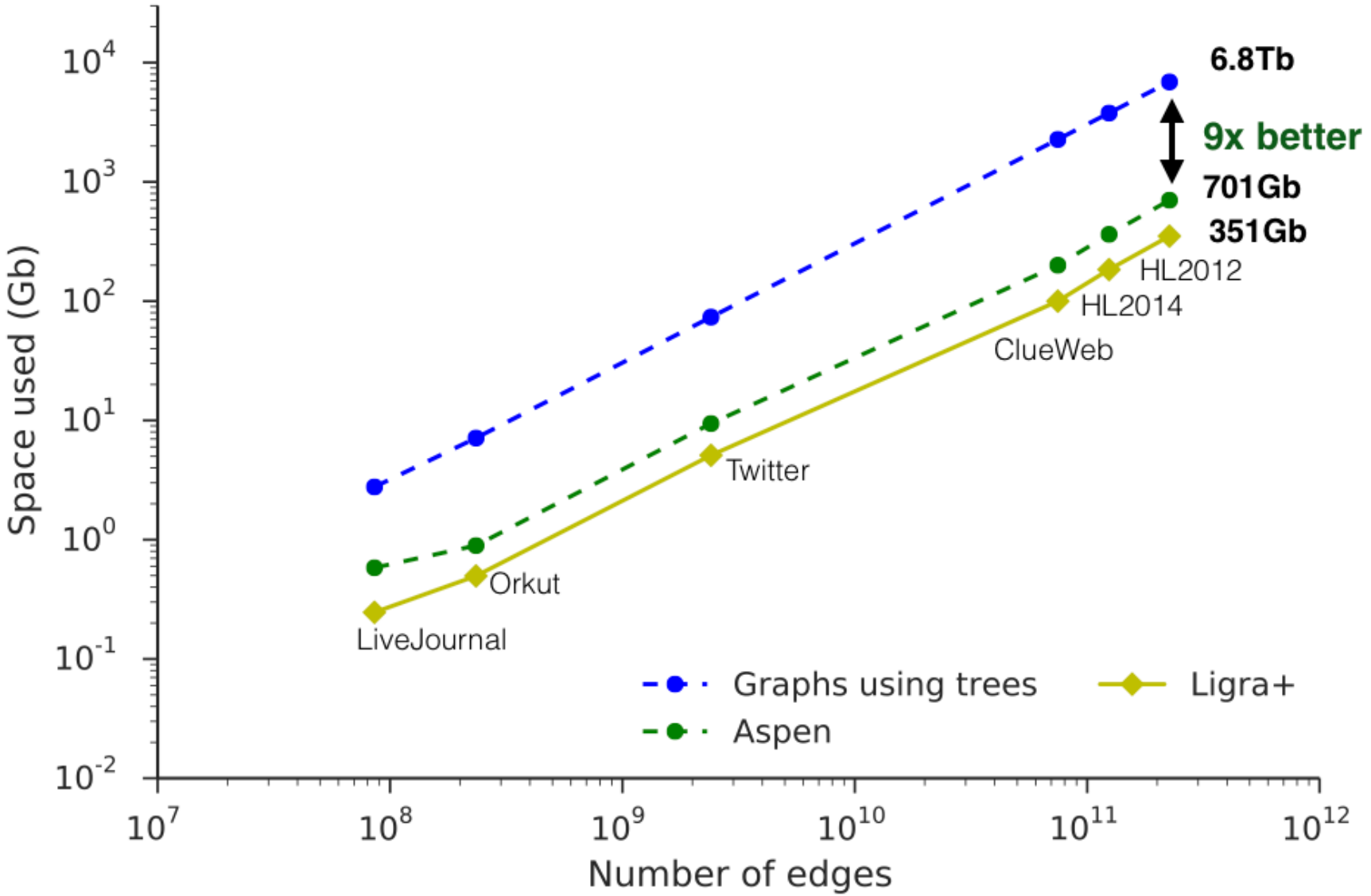

C-tree

- Supports parallel bulk insertions and deletions efficiently

# Space Usage of Graphs using C-trees

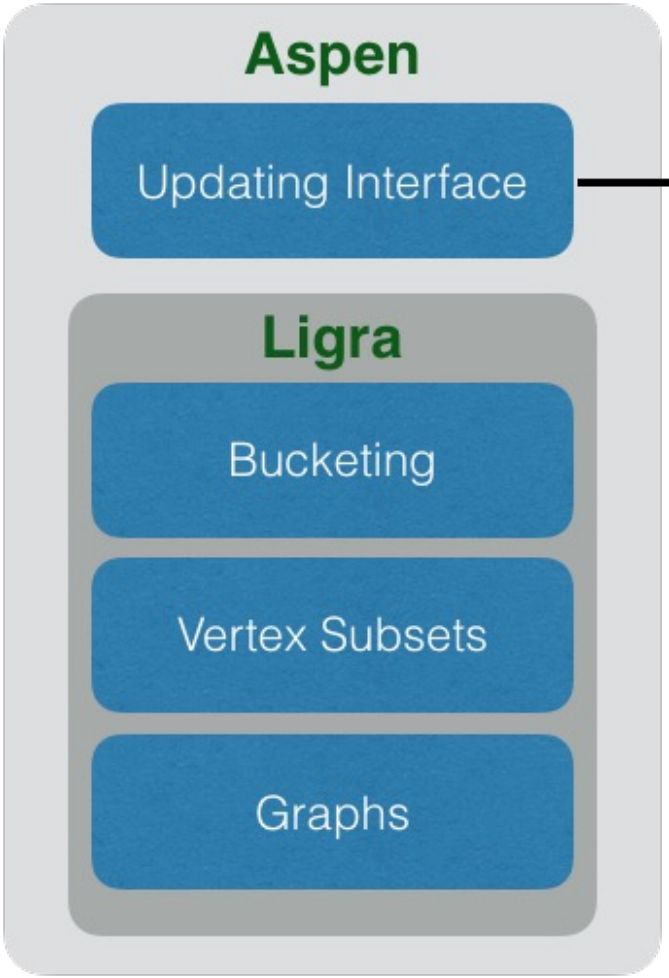


# Space Usage of Graphs using C-trees



# Aspen Framework

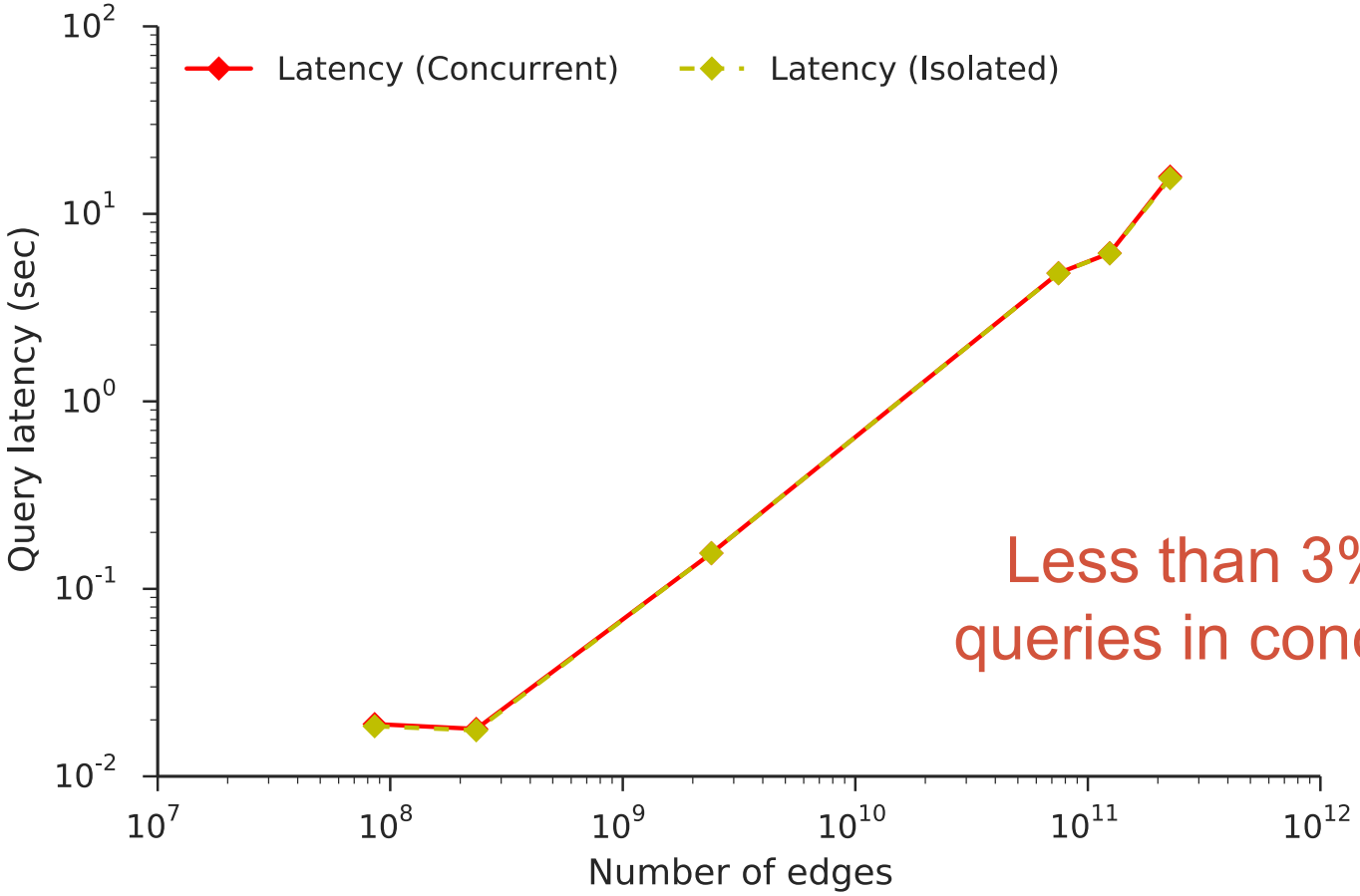
- Extension of Ligra with primitives for **updating graphs**
- Supports single-writer multi-reader concurrency




versioned graph
acquire()
release($G_s$)
multi_insert($E_B$), multi_delete($E_B$)
...

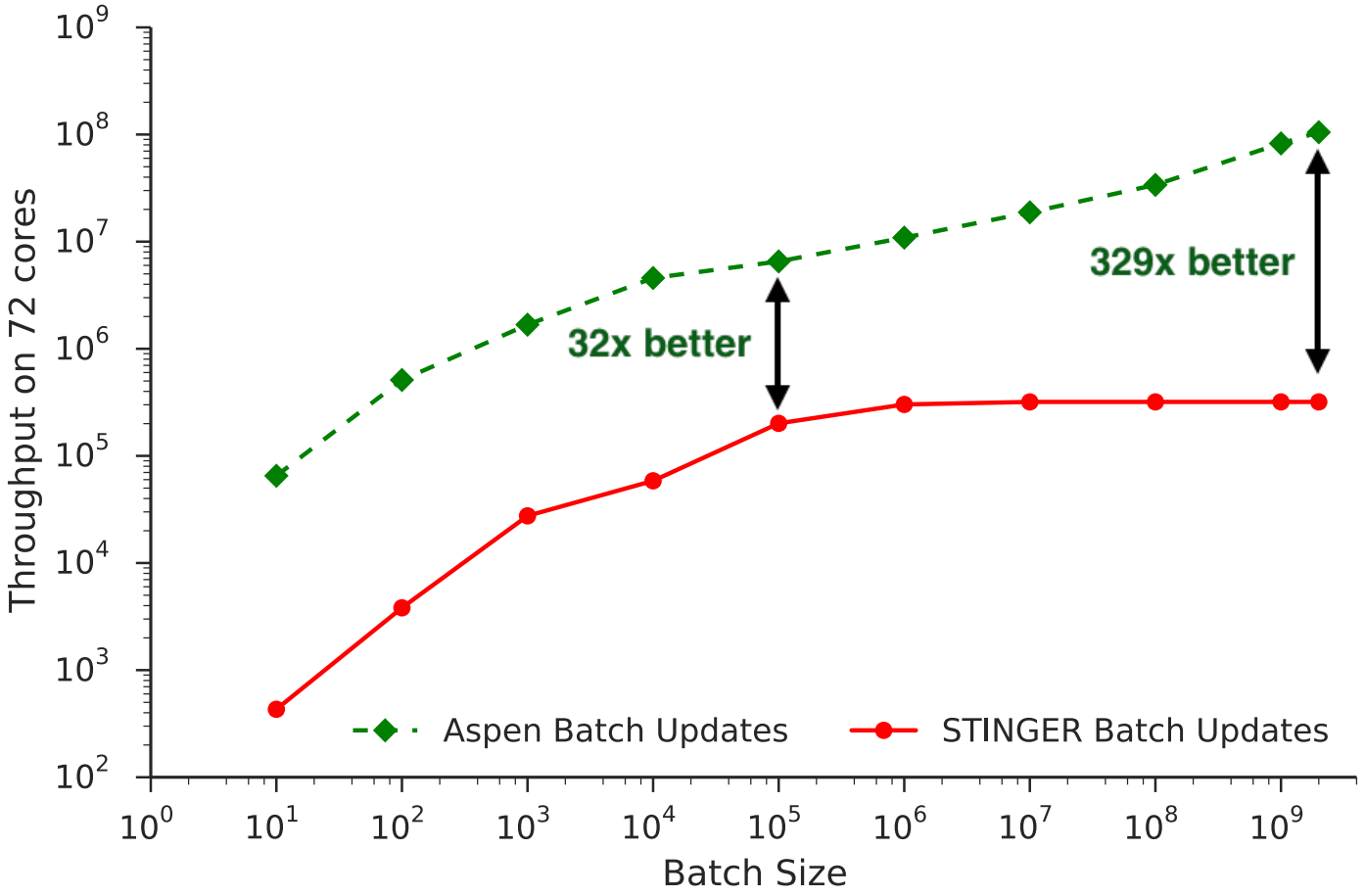
# Concurrent Queries and Update Task

- 72-core hyper-threaded machine with 1TB RAM
- 1 hyper-thread updating graph while remaining hyper-threads running parallel BFS




Less than 3% impact on queries in concurrent setting

# Parallel Batch Updates




- Aspen processes the Common Crawl graph at over 100M edge updates per second

# Conclusion

- GraphIt: graph DSL that enables easy exploration of large graph optimization space
  - https://graphit-lang.org/

- Aspen: streaming graph framework that enables concurrent queries and updates with low latency
  - https://github.com/ldhulipala/aspen/