Low-Latency Graph Streaming Using Compressed Purely-Functional Trees

Laxman Dhulipala, Guy Blelloch, and Julian Shun

PLDI 2019

Streaming Graph Processing

...

...
insert(e_4)

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

Update Stream

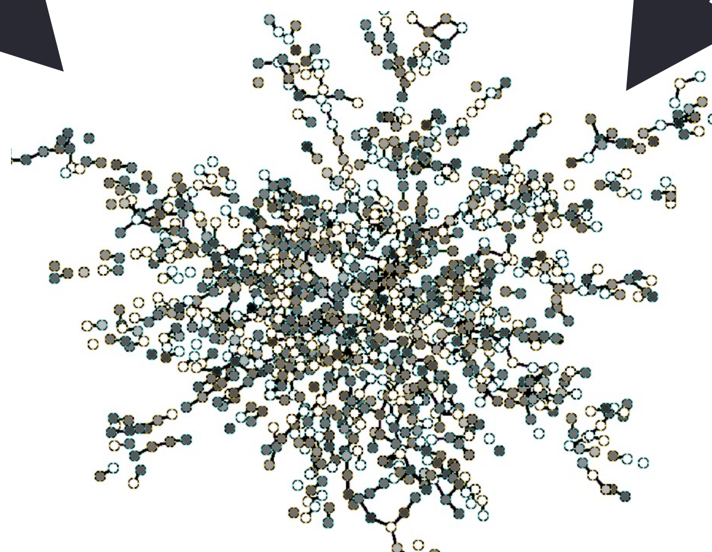
...

ClusteringCoefficient(v)

Connectivity(G)

Reachable(u—>v)?

Query Stream

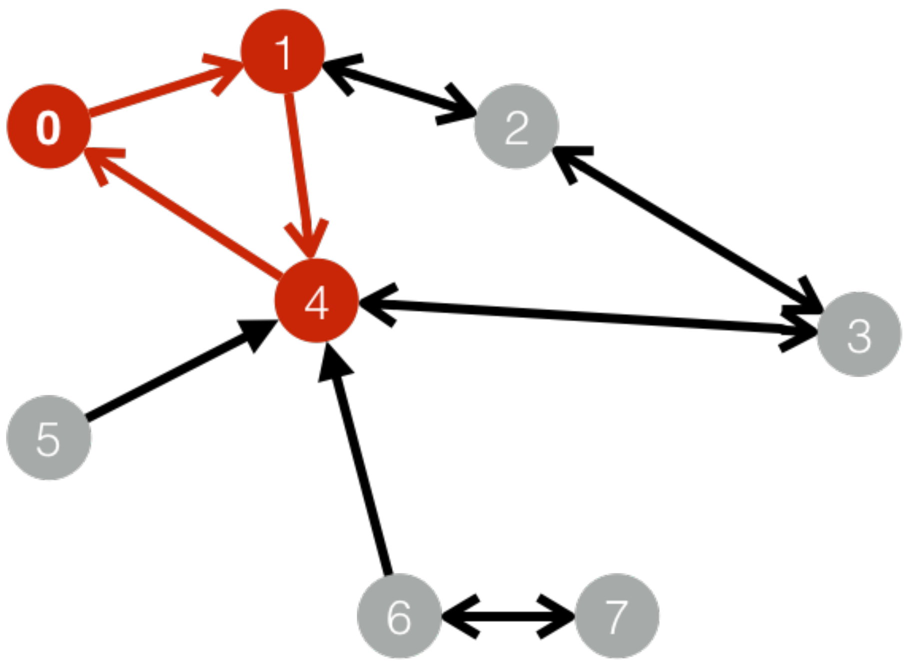


G : graph

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

Example: Fraud Detection

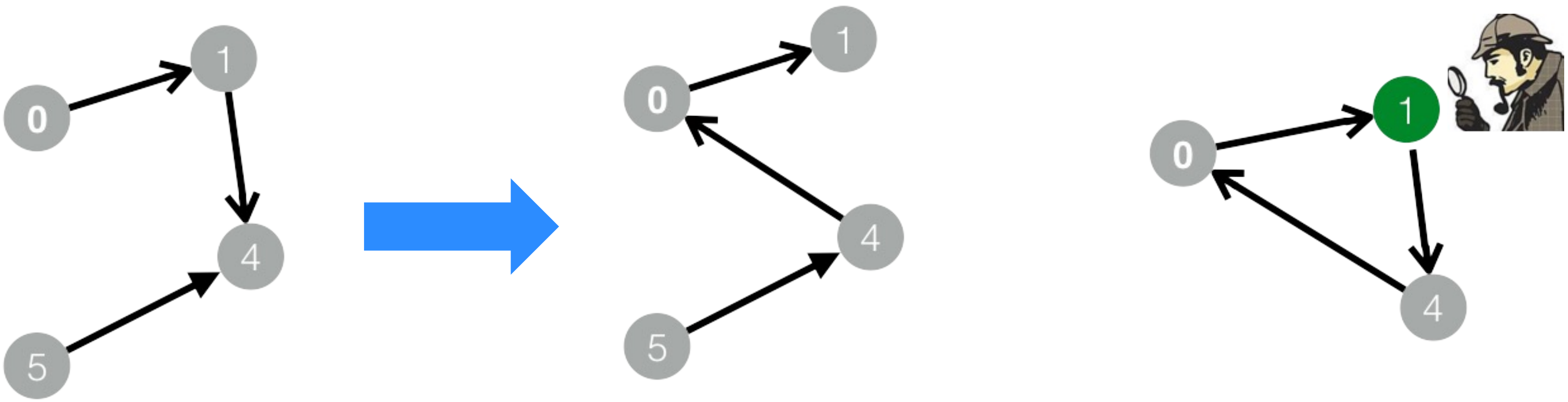
- Bank maintains a transaction graph
- Transactions occur at a high rate (1k-10k/sec)



- Goal: quickly detect anomalies in evolving transaction graph

Relaxing Serializability

- Could detect a cycle that never existed!



Evolving graph

Observed graph

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, GraphOne, and some graph databases)
  - Snapshots are not space-efficient and lead to high latency

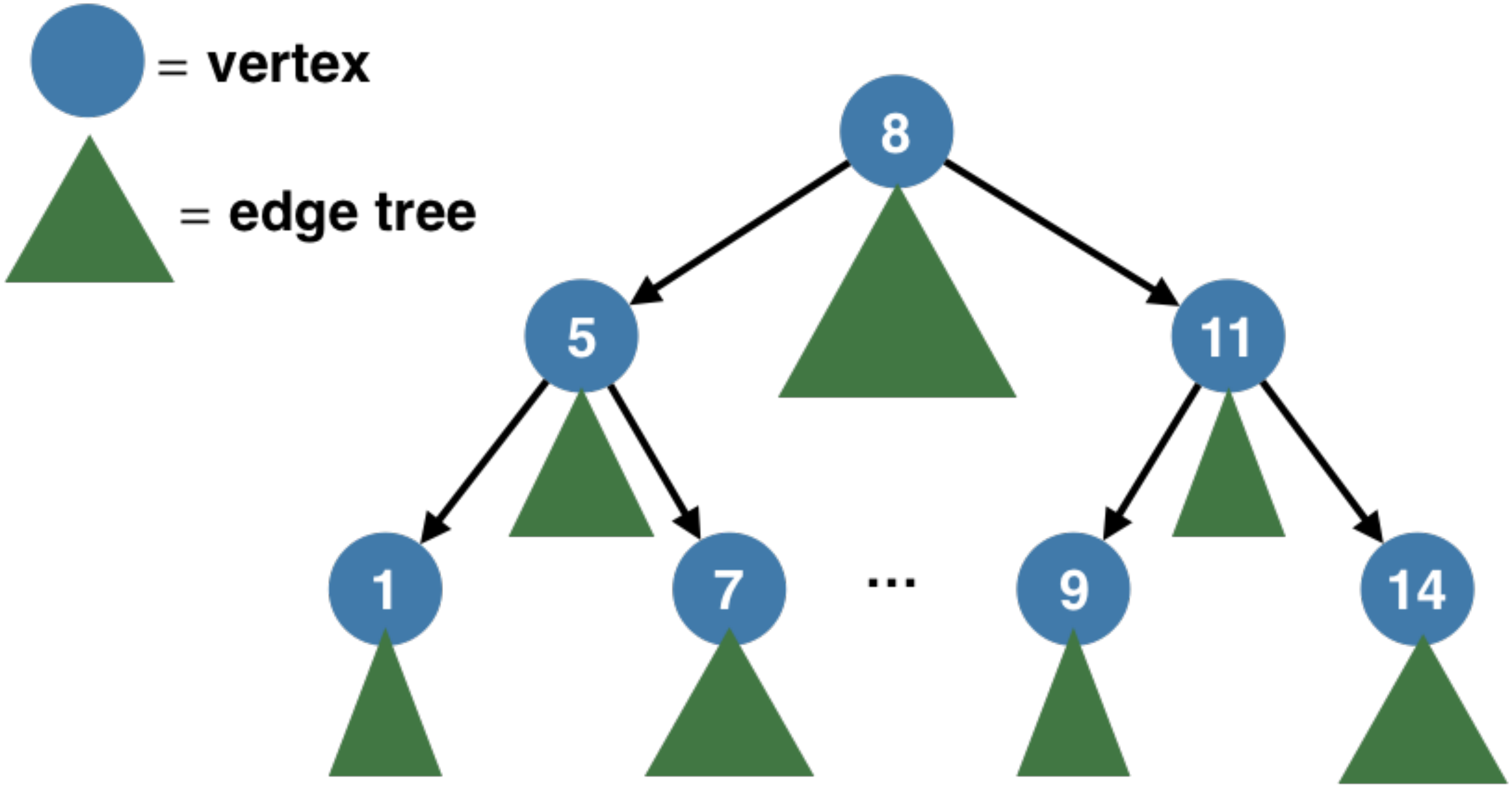
- Our framework **Aspen** uses lightweight snapshots to enable low-latency concurrent queries and updates

Terminology: Streaming vs. Dynamic

- **Streaming graph processing**: Goal is to run algorithms on a graph that is changing in real-time while obtaining serializable results
  - Need to process updates concurrently with algorithm execution
- **Dynamic graph algorithms**: Goal is to update the result of an algorithm based on updates to the graph itself
  - Should be more efficient than recomputing answer from scratch
  - Allows for barriers between algorithm execution and processing updates
- This talk is about streaming graph processing

Graphs Using Purely Functional Trees

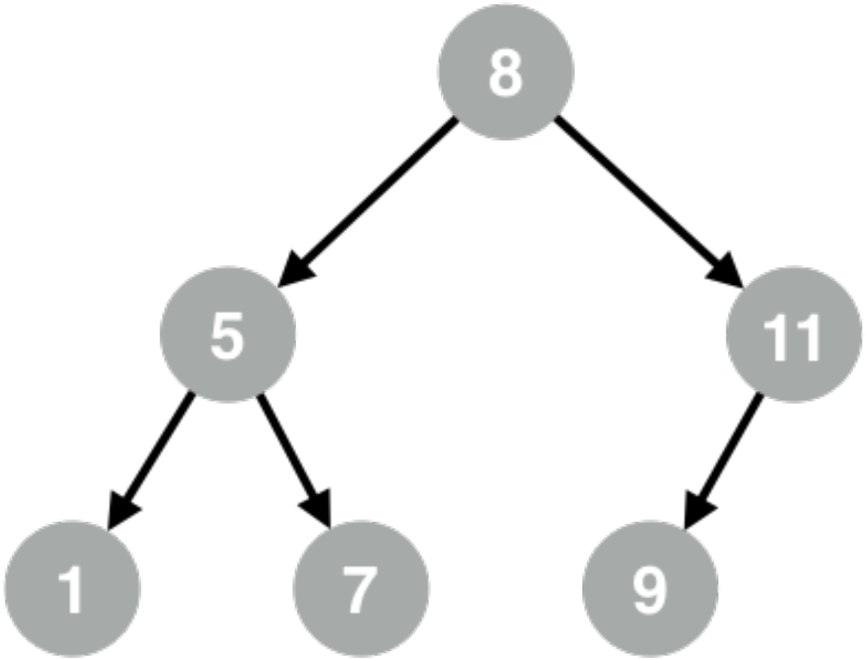
- Purely functional trees can be updated efficiently (in logarithmic time/space) while retaining old copy of tree
- Aspen uses tree of **vertices**, where each vertex stores a tree of its incident **edges**



Updates via Path Copying

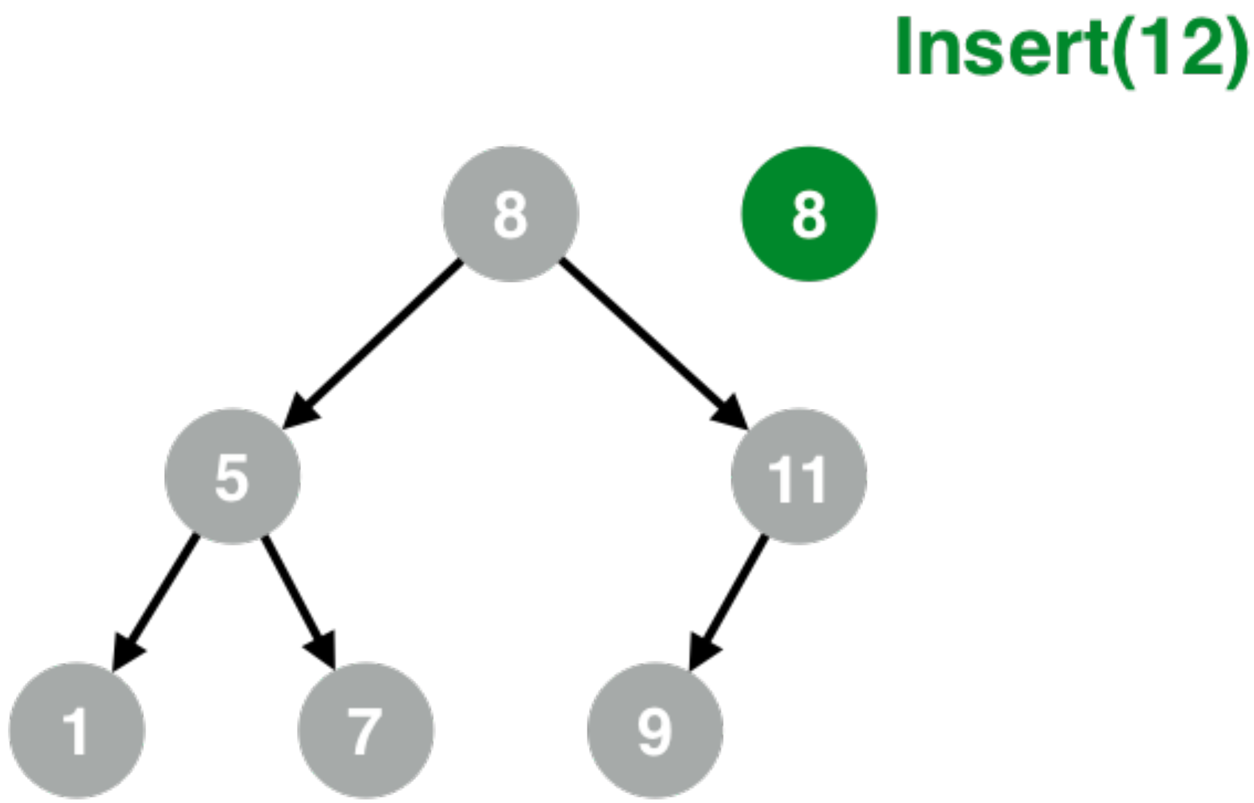
- Easy to generate new versions via path copying

Insert(12)



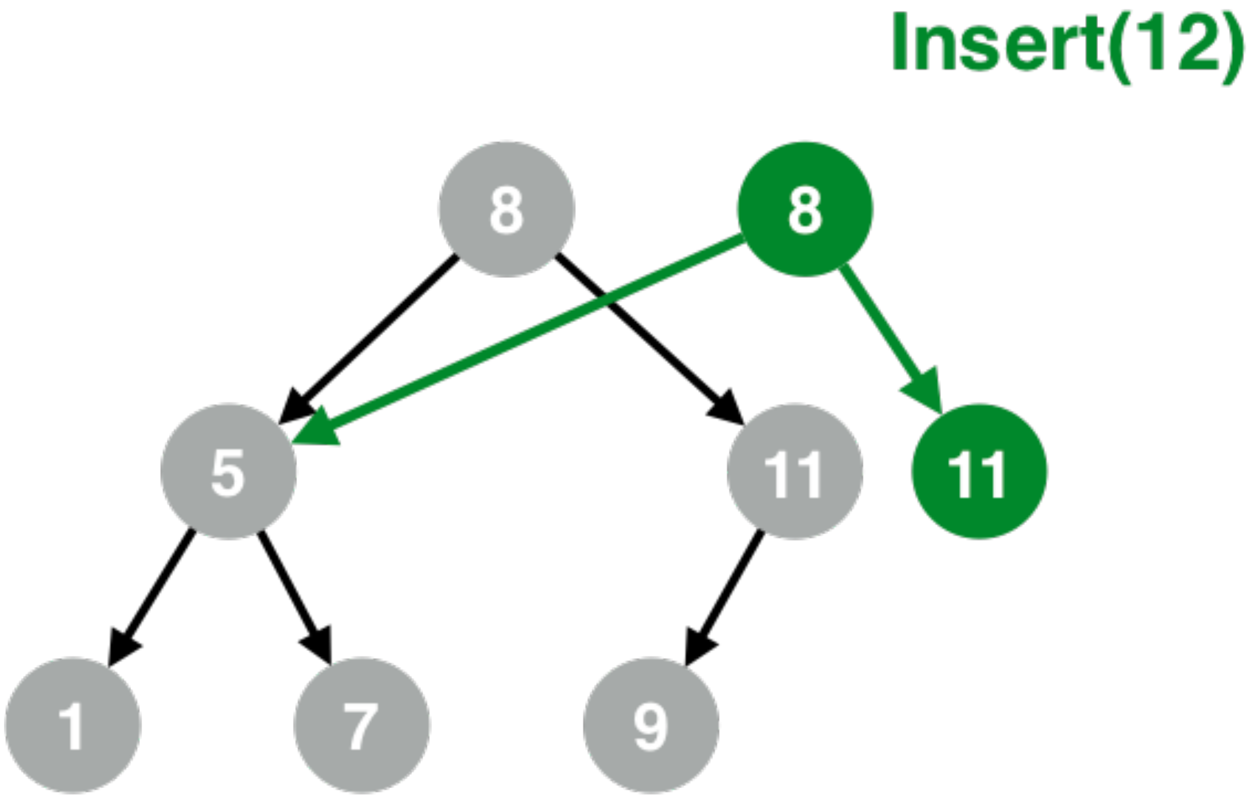
Updates via Path Copying

- Easy to generate new versions via path copying



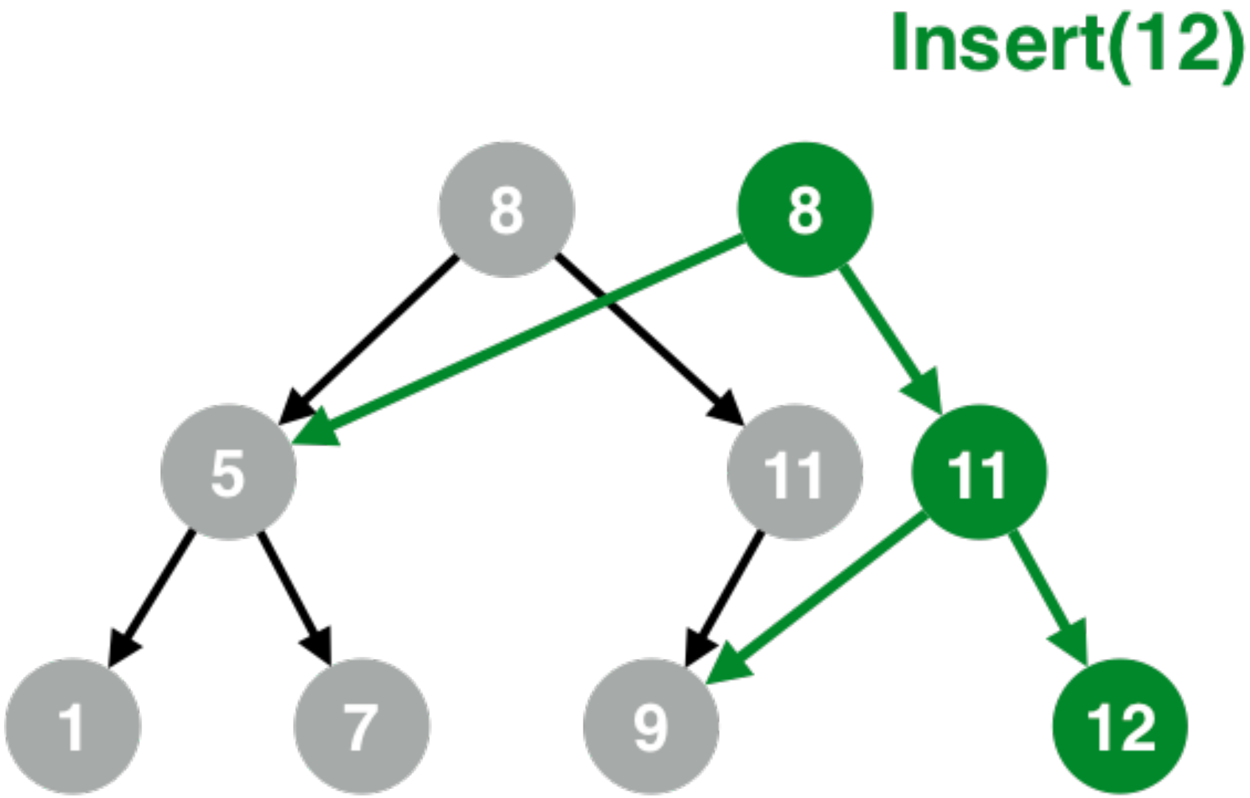
Updates via Path Copying

- Easy to generate new versions via path copying



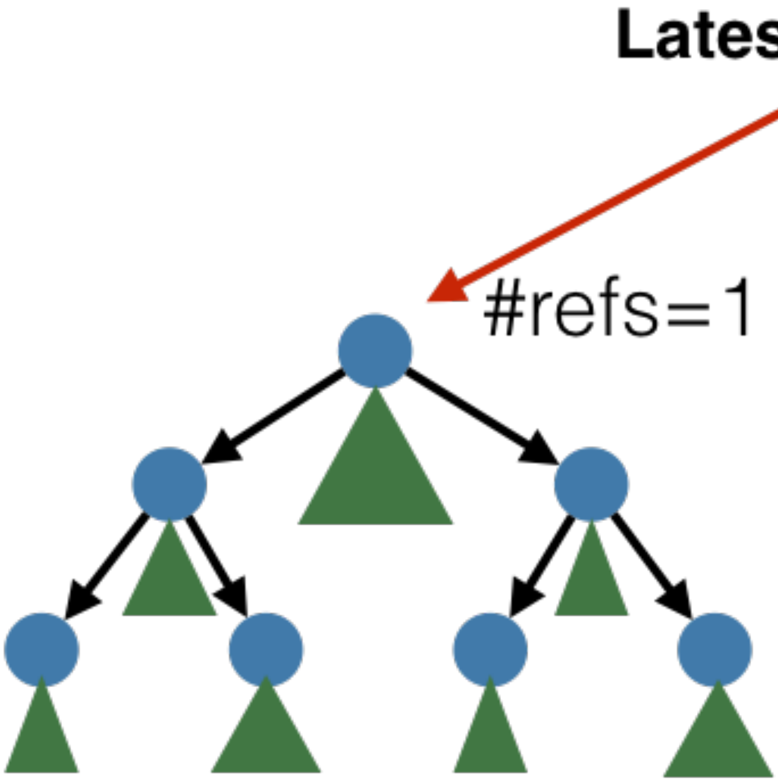
Updates via Path Copying

- Easy to generate new versions via path copying

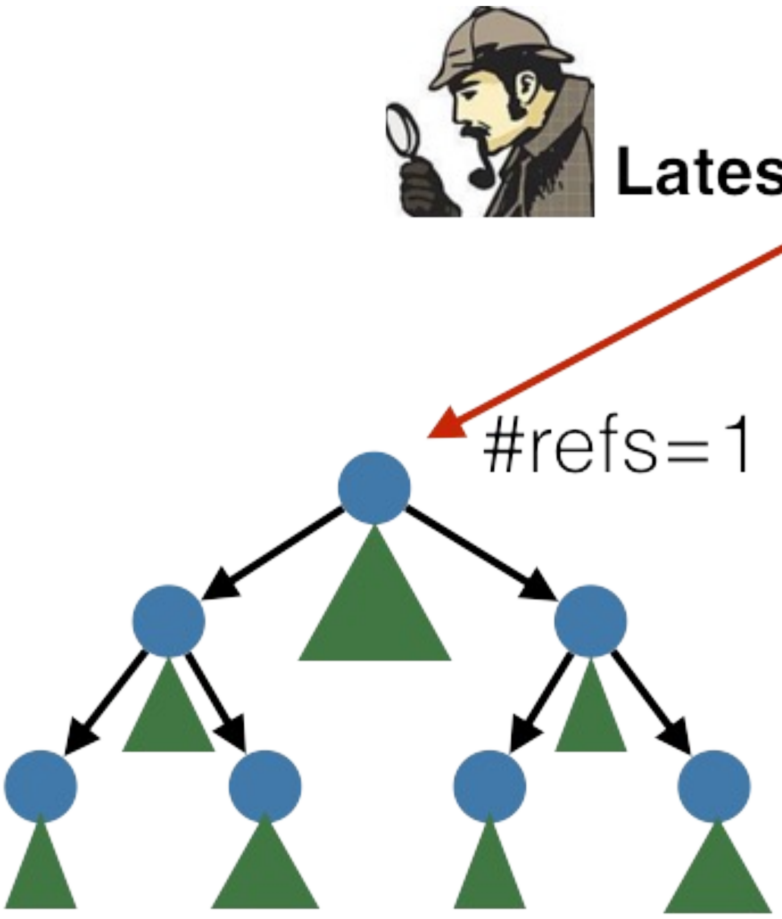


- We can obtain immutability versions of the tree

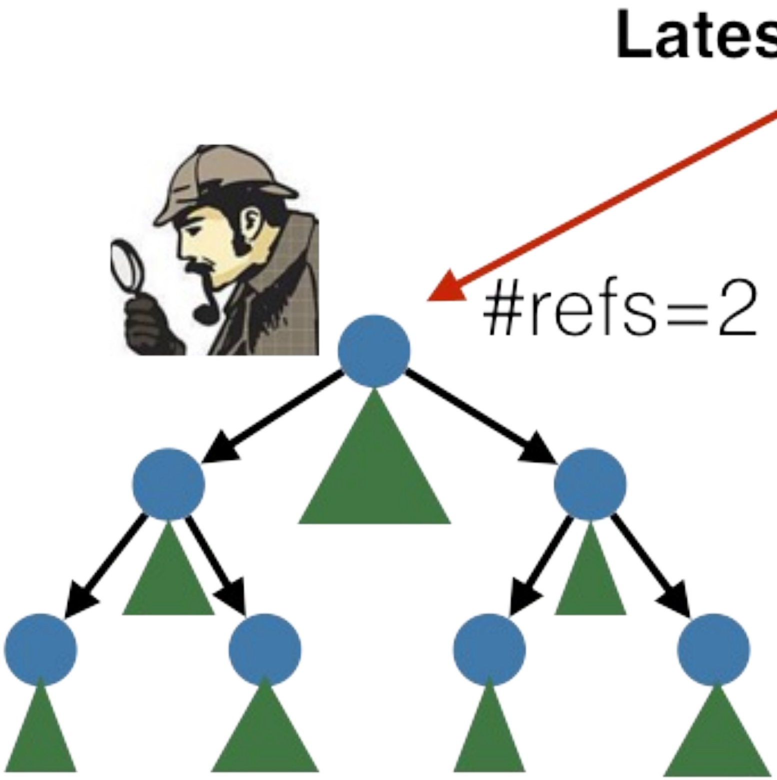
Immutability Enables Concurrency



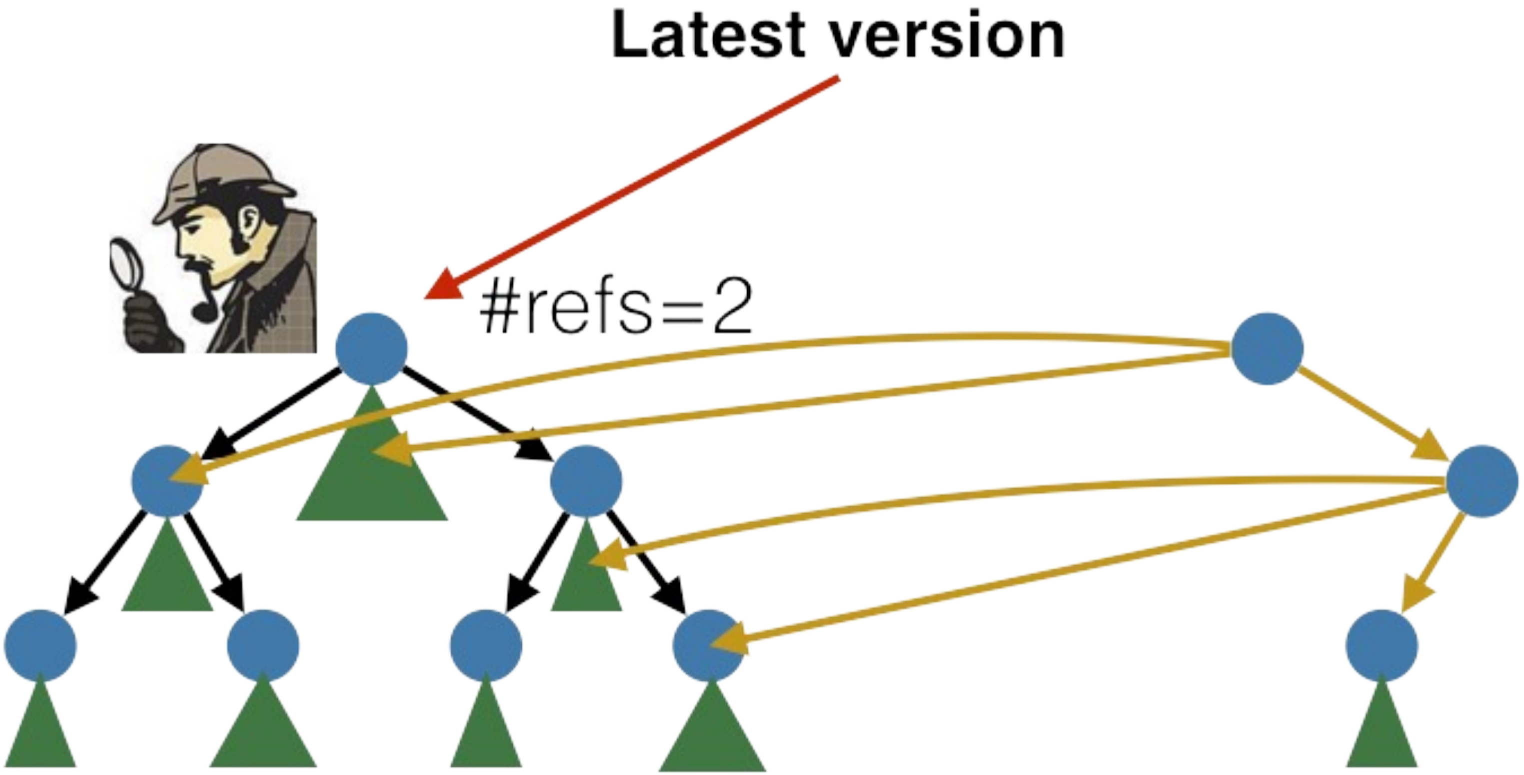
Immutability Enables Concurrency



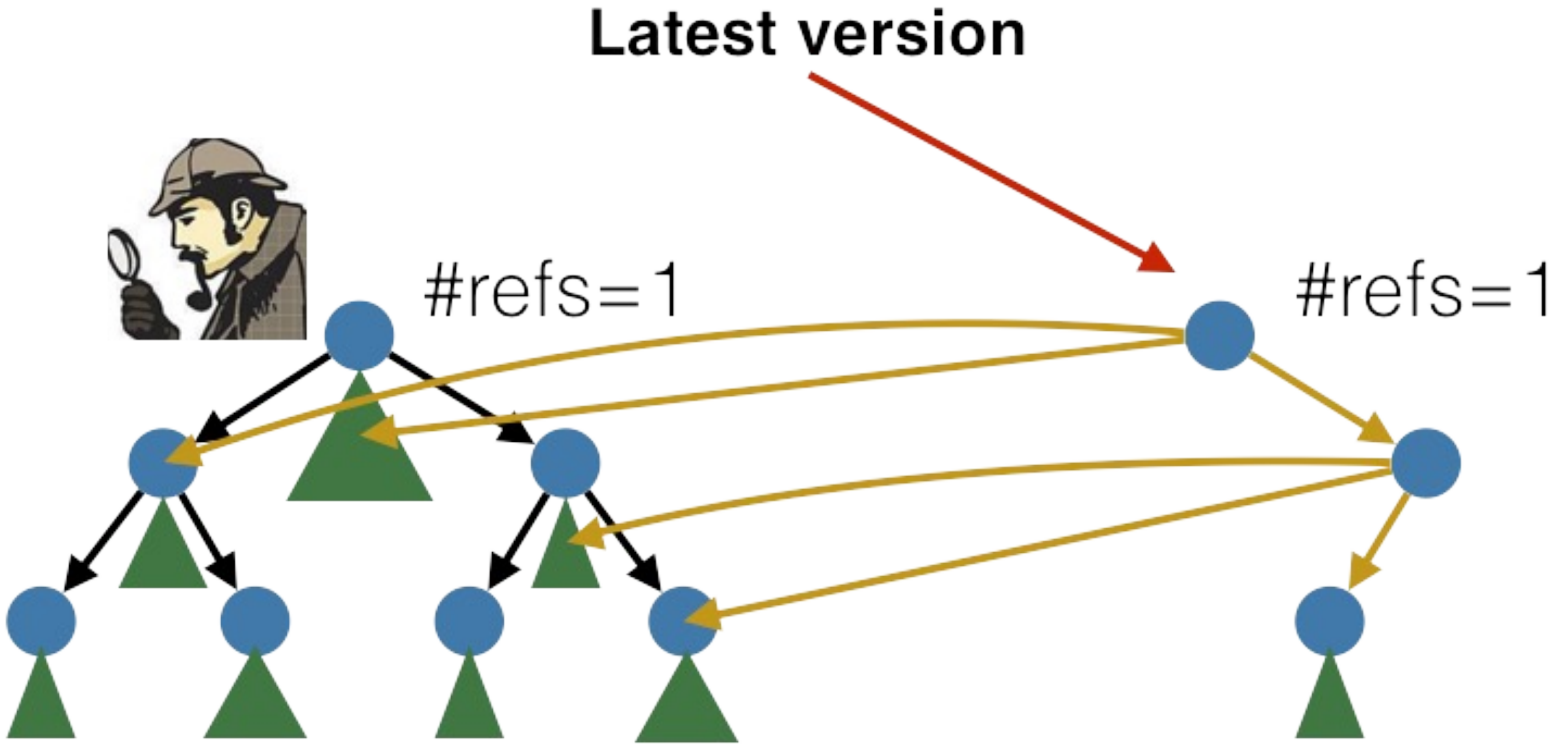
Immutability Enables Concurrency



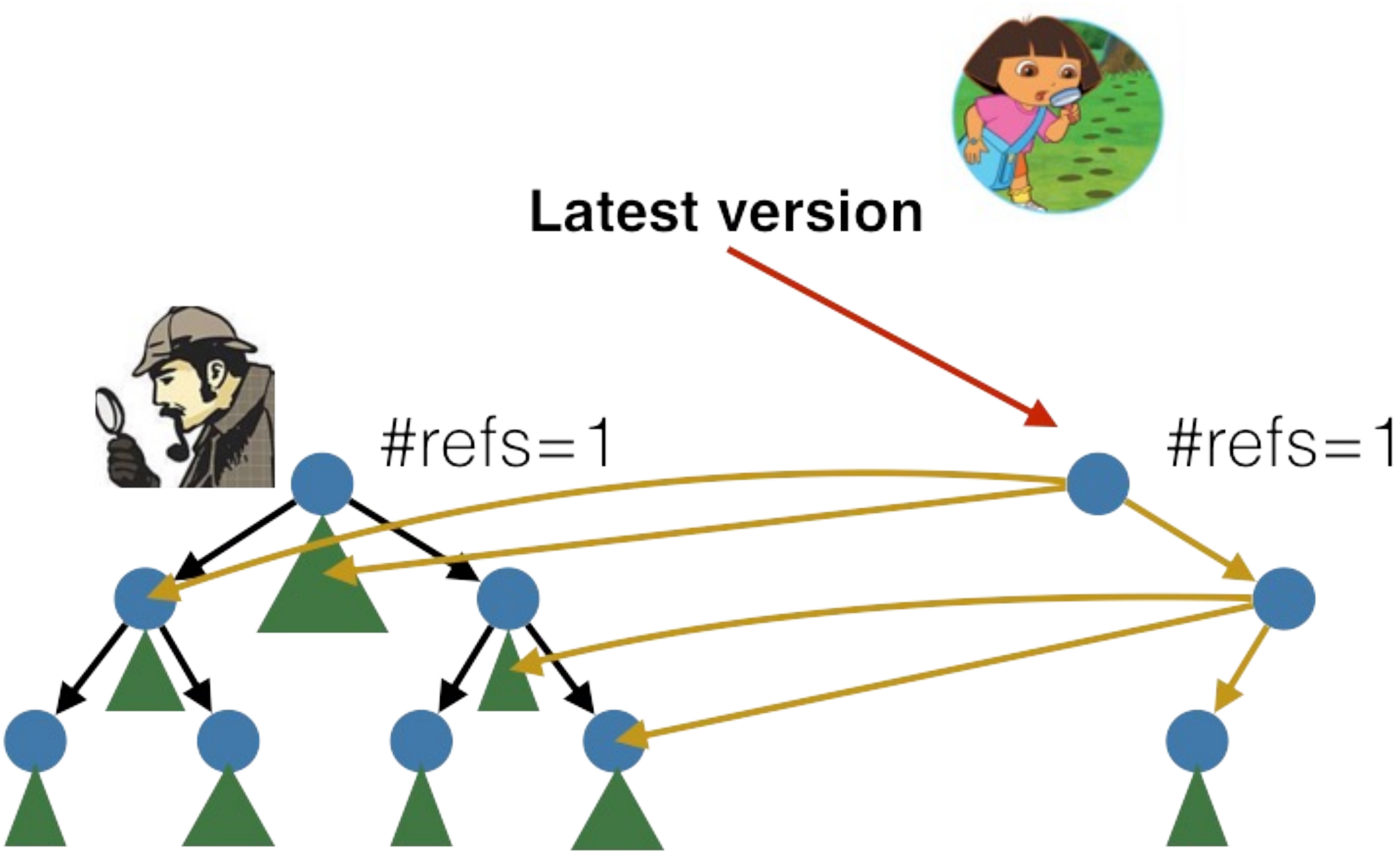
Immutability Enables Concurrency



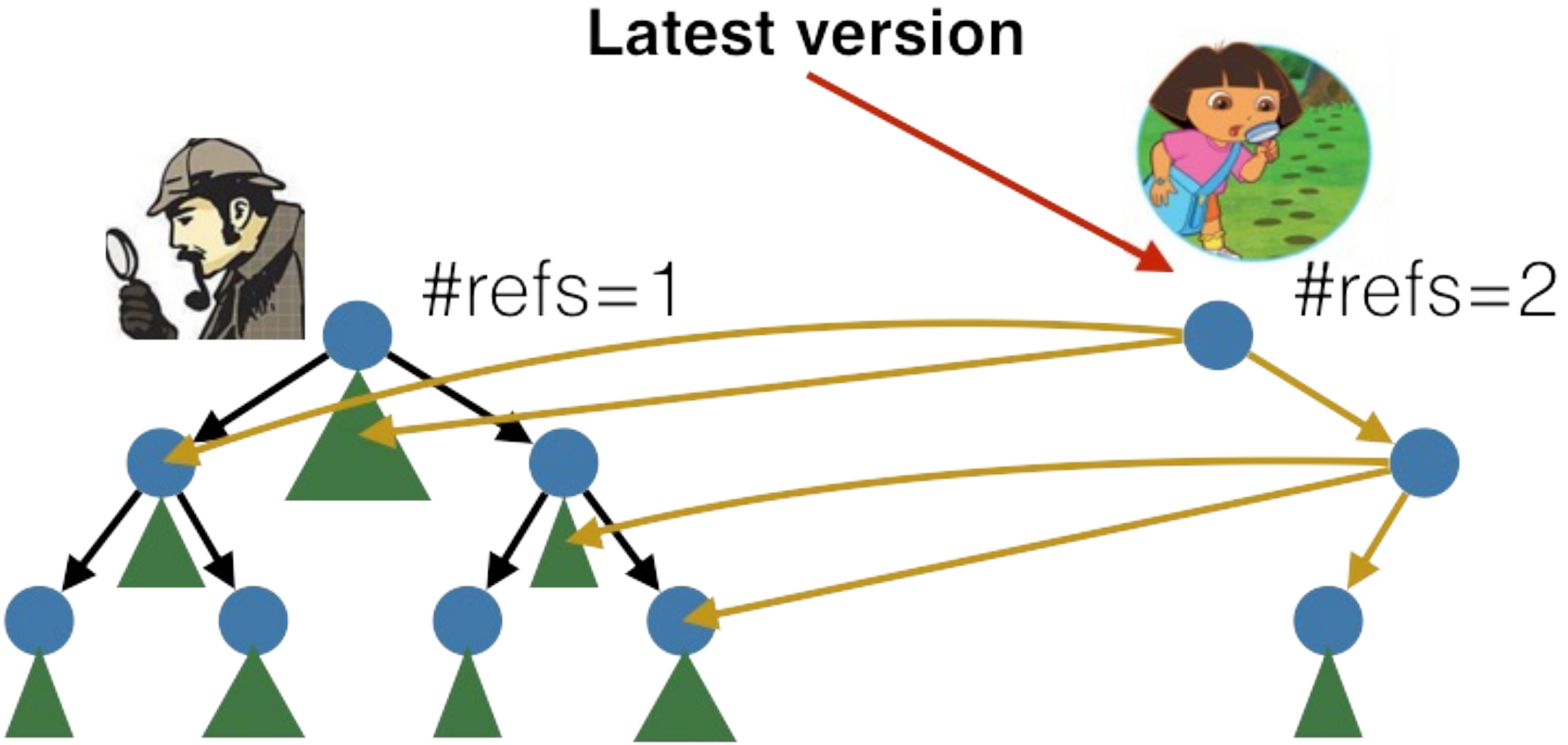
Immutability Enables Concurrency



Immutability Enables Concurrency

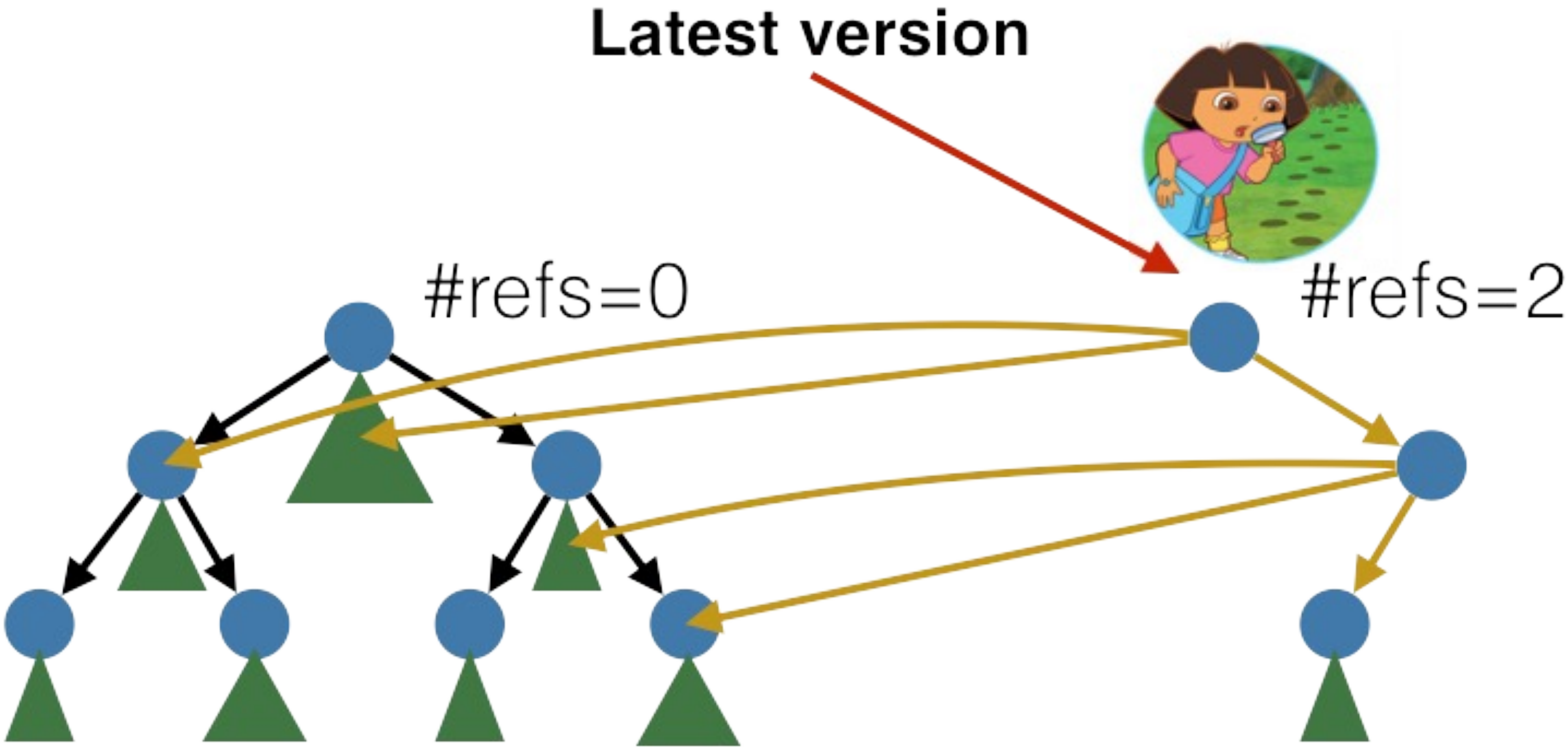


Immutability Enables Concurrency



Immutability Enables Concurrency

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



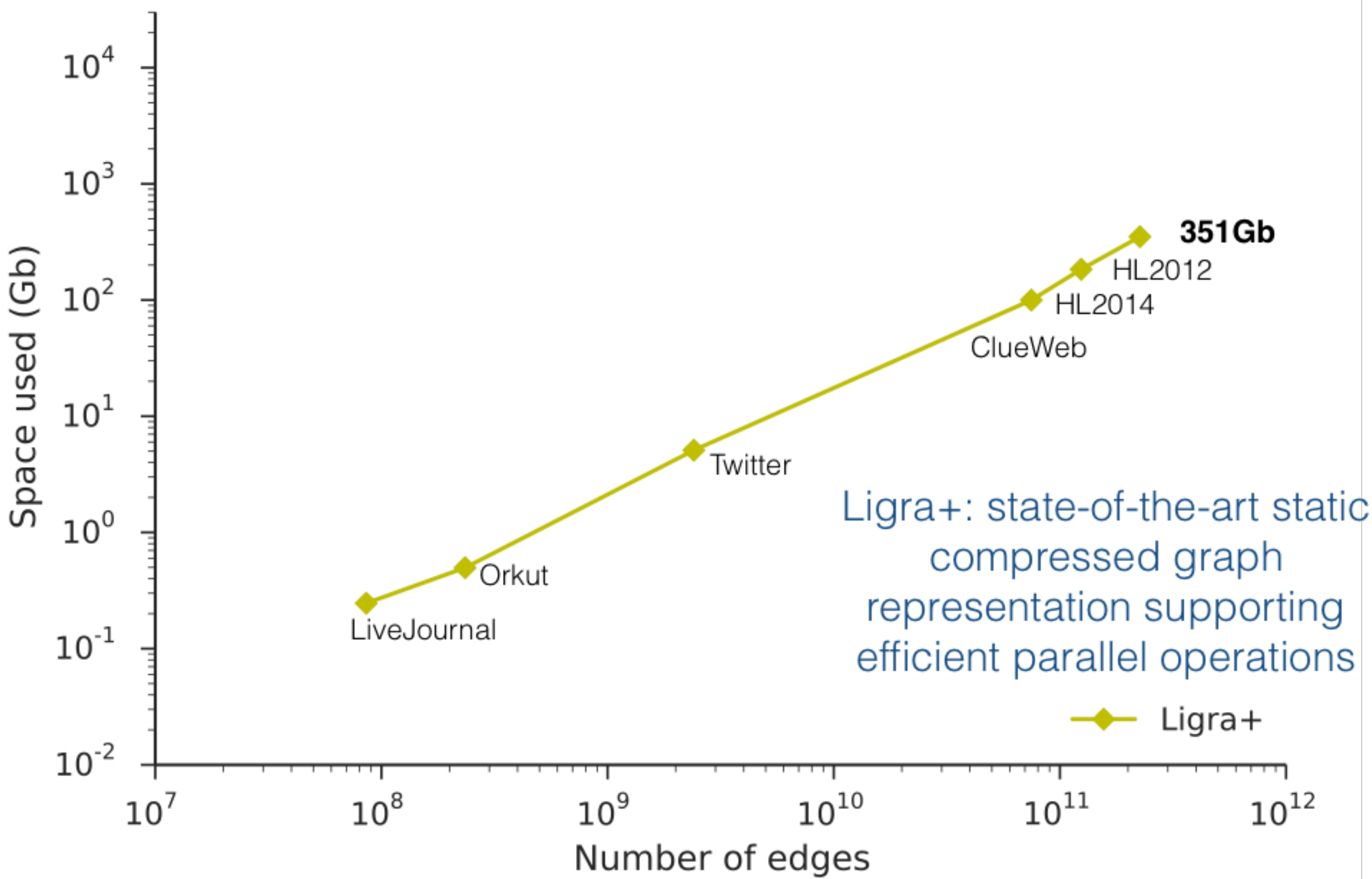
Disadvantages of representing graphs using trees

- Poor Cache Usage
  - One tree node per vertex and edge
  - One cache miss per edge access in the worst case

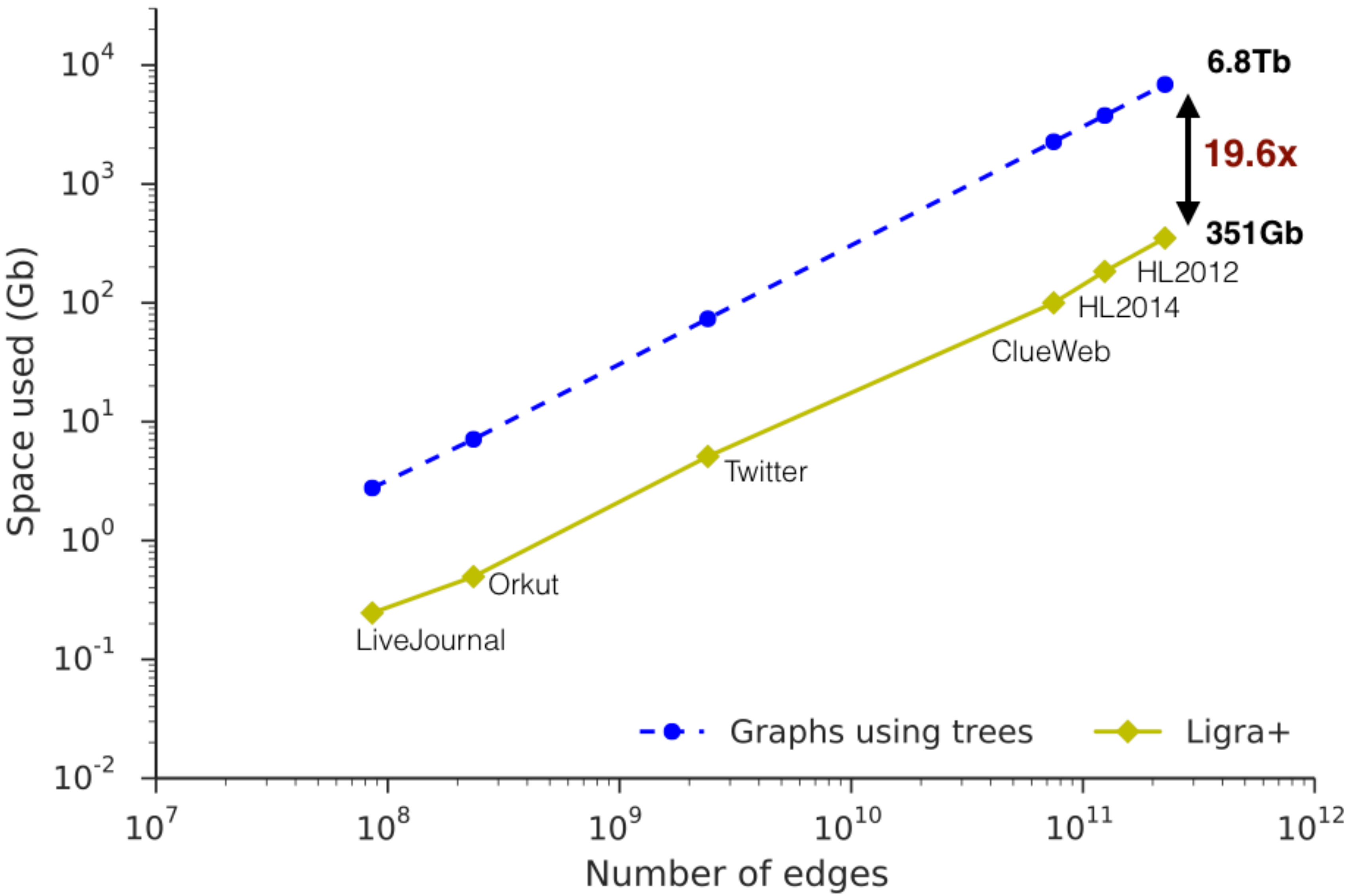
- Space Inefficiency
  - Need to store children pointers and metadata on tree nodes
  - Lose ability to perform integer compression

Requires close to 7TB of memory to store the symmetrized Hyperlink 2012 graph (225B edges)!

Space Overhead of Graphs using Trees

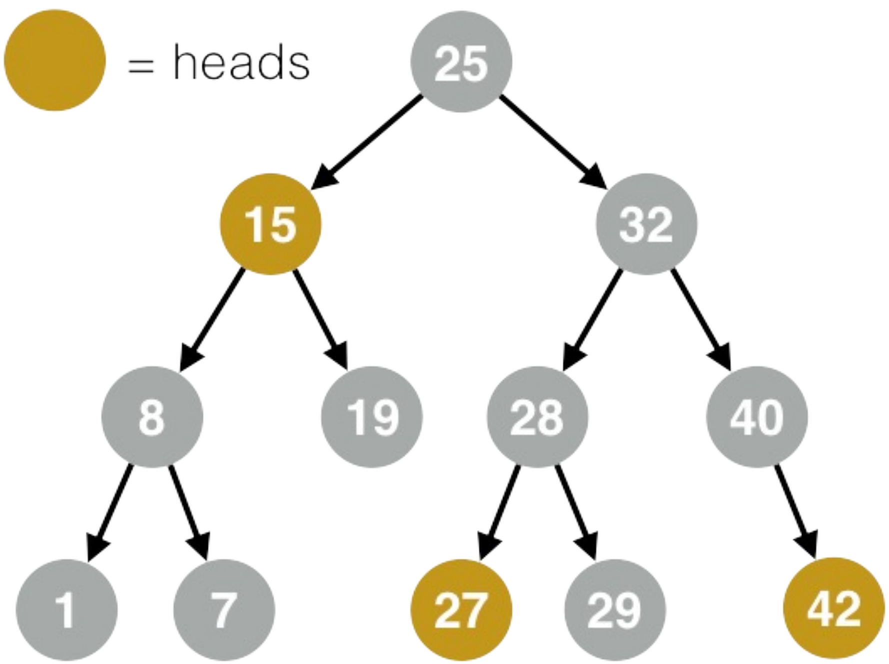


Space Overhead of Graphs using Trees



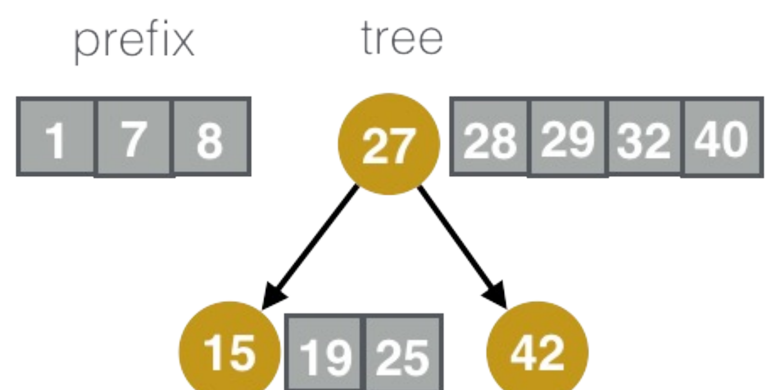
C-tree

- 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

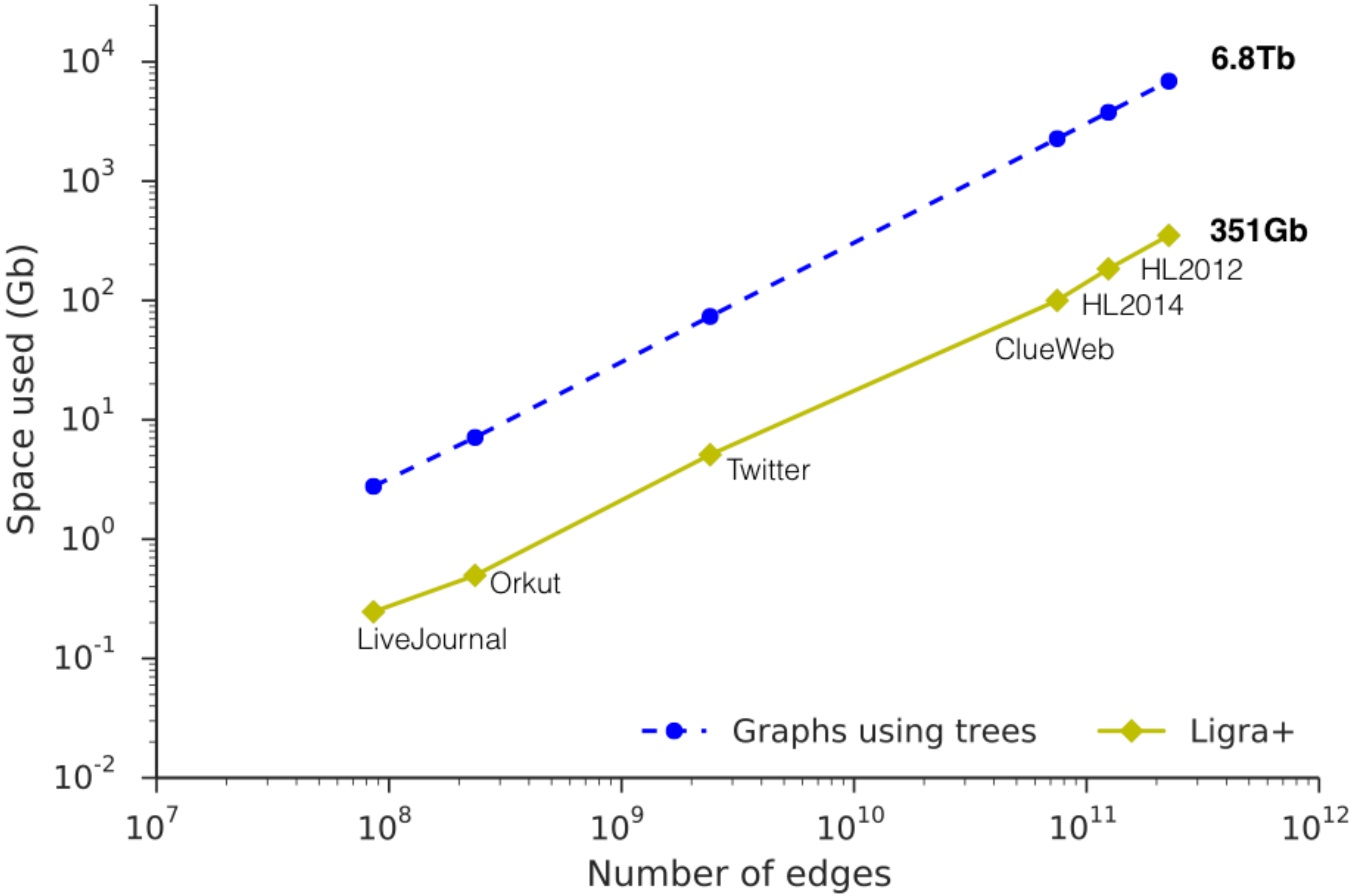
Further improve space usage for integer C-trees by difference encoding chunks



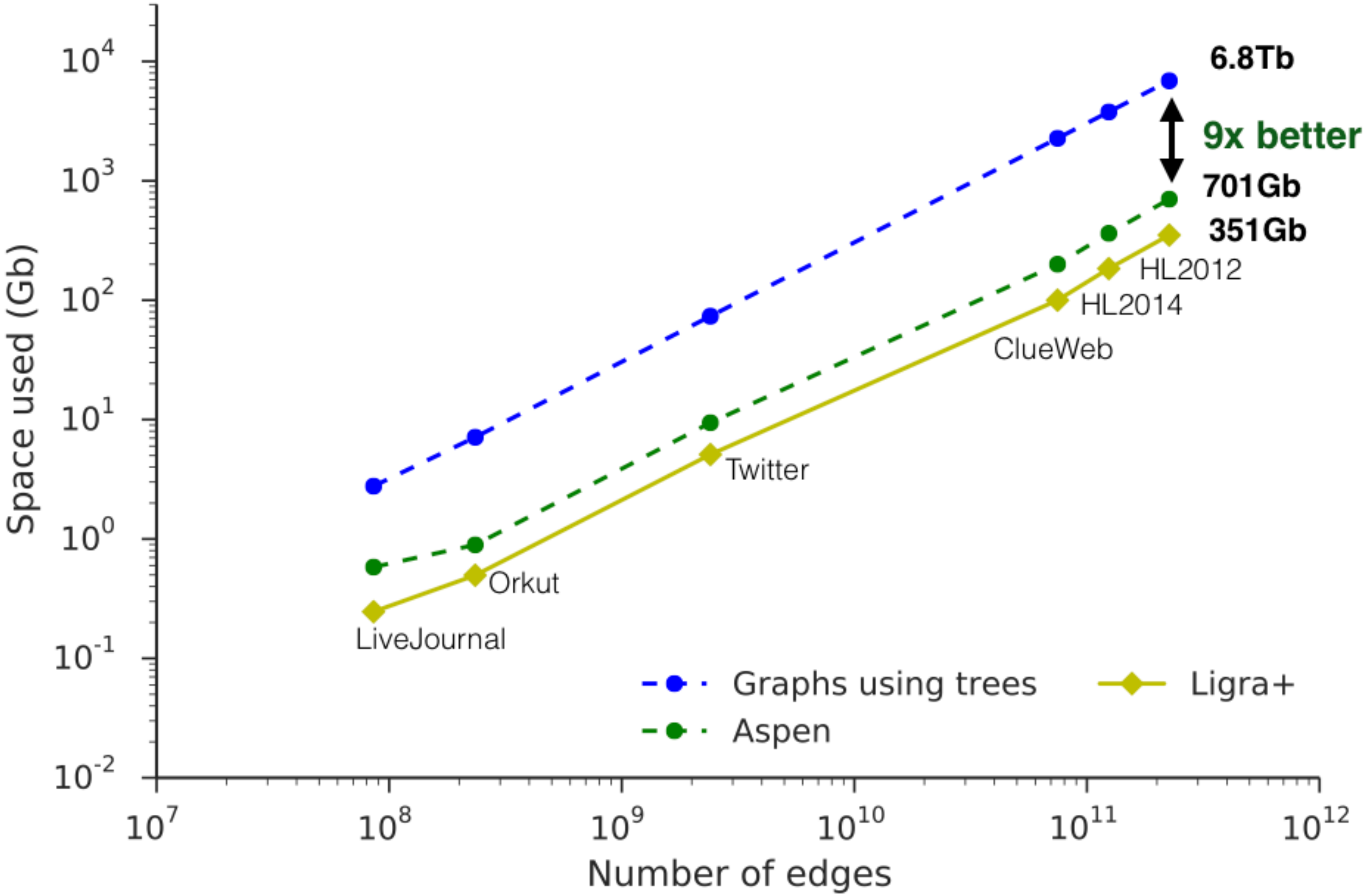
C-tree

- Supports parallel bulk insertions and deletions efficiently

Space Usage of Graphs using C-trees

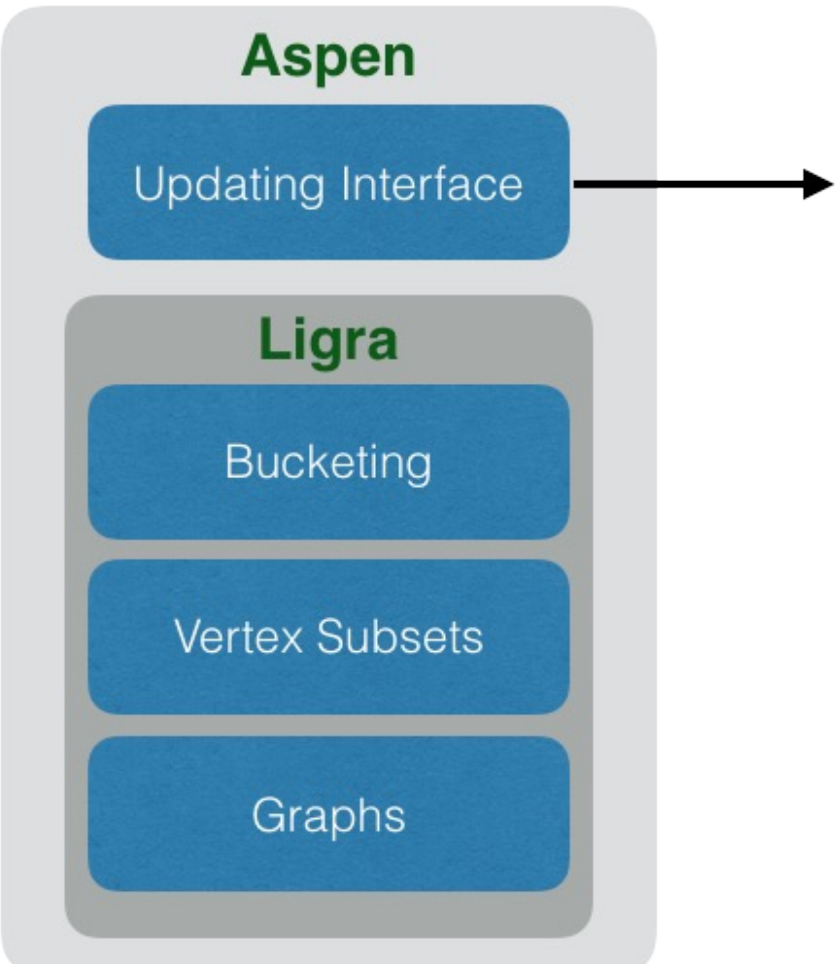


Space Usage of Graphs using C-trees



Aspen Framework

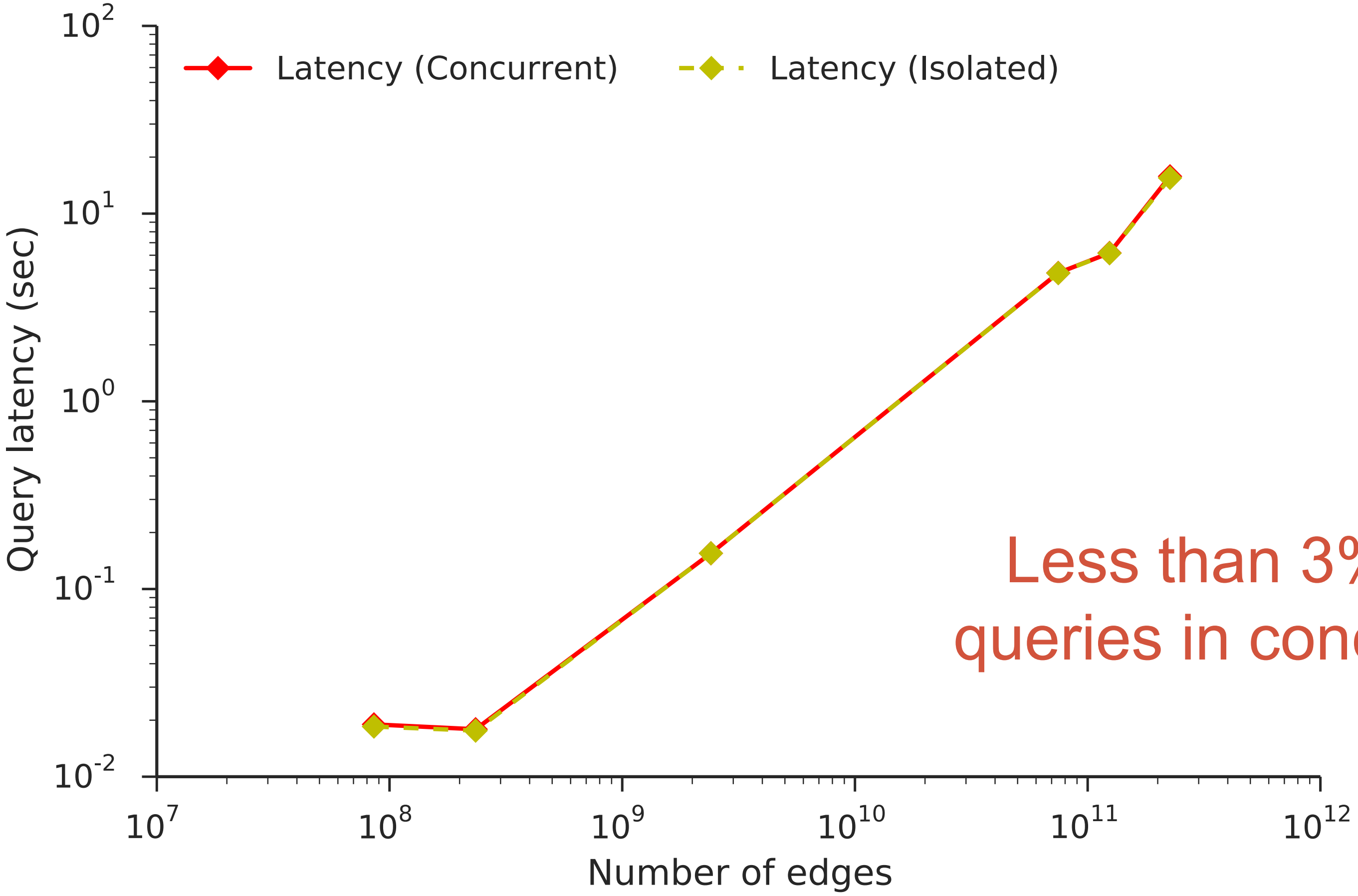
- Extension of Ligra with primitives for **updating graphs**
- Supports single-writer multi-reader concurrency



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

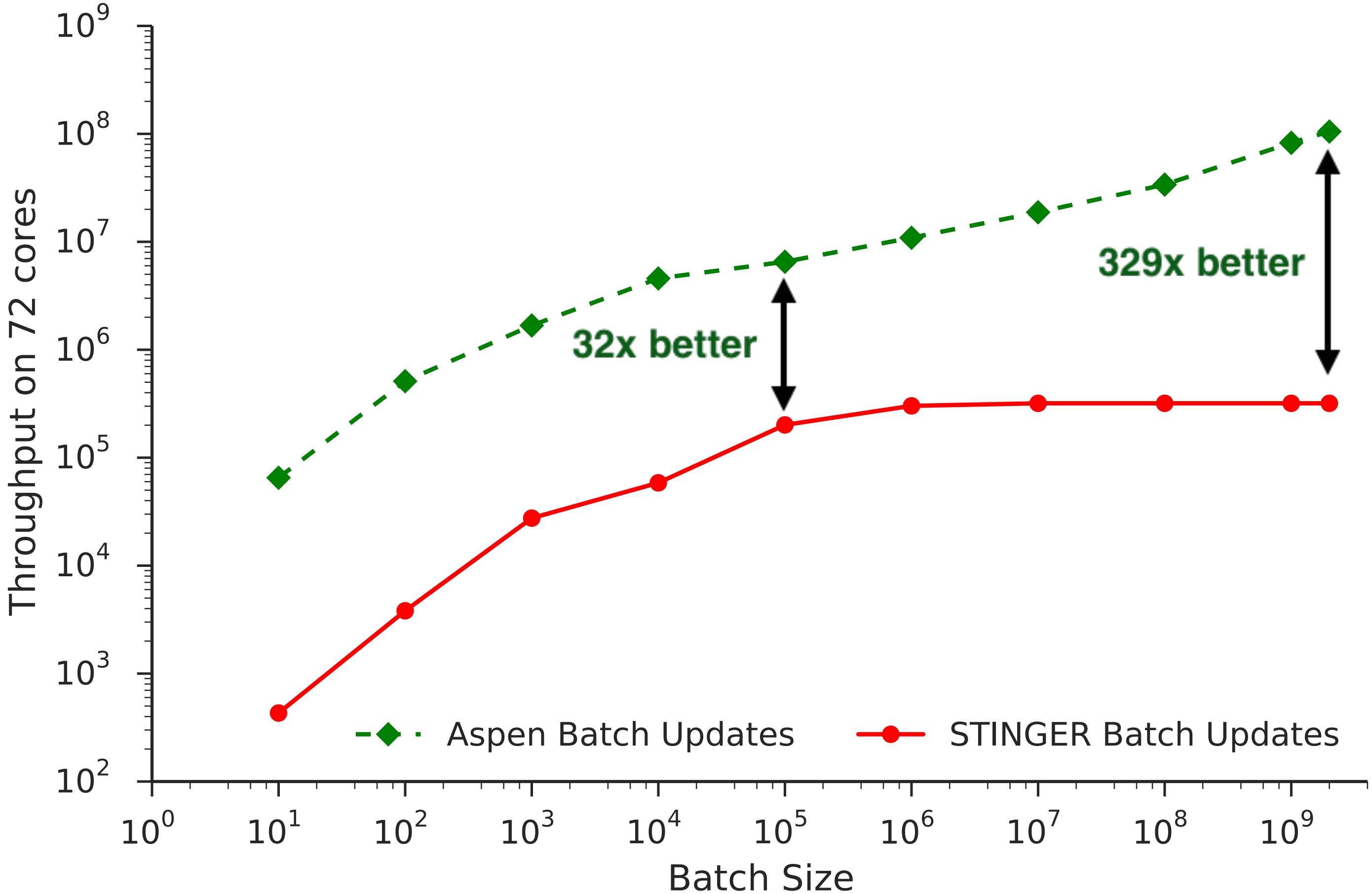
Concurrent Queries and Updates

- 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 Hyperlink 2012 graph at over 100M edge updates per second
- About 1.4x faster than GraphOne (developed concurrently and independently) based on a rough comparison