IIT Hyderabad

Concurrent Wait-Free Graph Snapshots using Multi-Versioning

NETYS 2024

Gaurav Bhardwaj, Ayaz Ahmed, Dr. Sathya Peri
Indian Institute of Technology, Hyderabad.

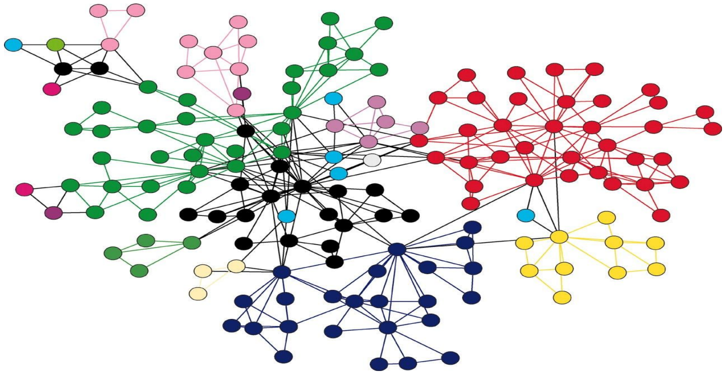
This project is funded by SERB Project CRG/2022/009391.

Content

- Background
- Motivation
- Graph Data Structure
- Graph ADT operations
- Memory Reclamation
- Results

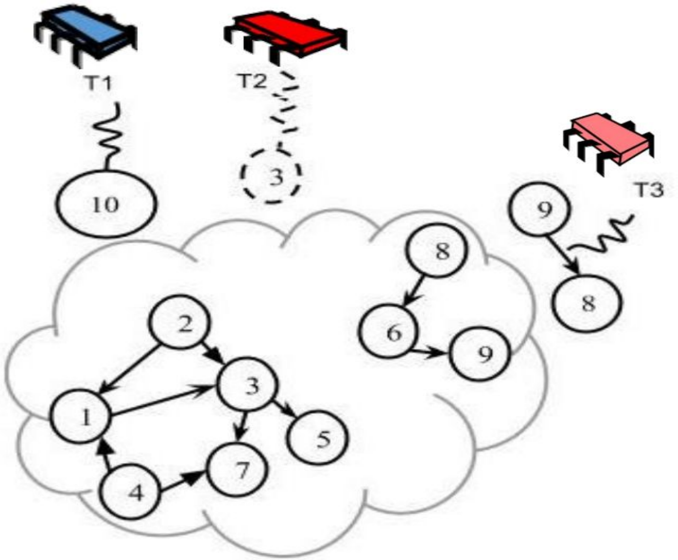
Graphs

- Graphs provide a visual representation of relationships and connections in data, making complex structures easier to understand.
- Graphs are ideal for modeling various relationships such as social networks, web links, and dependencies between tasks.
- Graphs allow for flexible data representation, enabling the modeling of diverse scenarios and real-world interactions.



Concurrent Graphs:

- In applications like social media analytics and network monitoring, concurrent graphs enable real-time analysis of dynamic data.
- Concurrent graphs are essential in large-scale systems, ensuring scalability and efficient processing of data in parallel.
- Concurrent graphs optimize resource utilization in multi-core processors, enhancing the performance of graph-based algorithms.



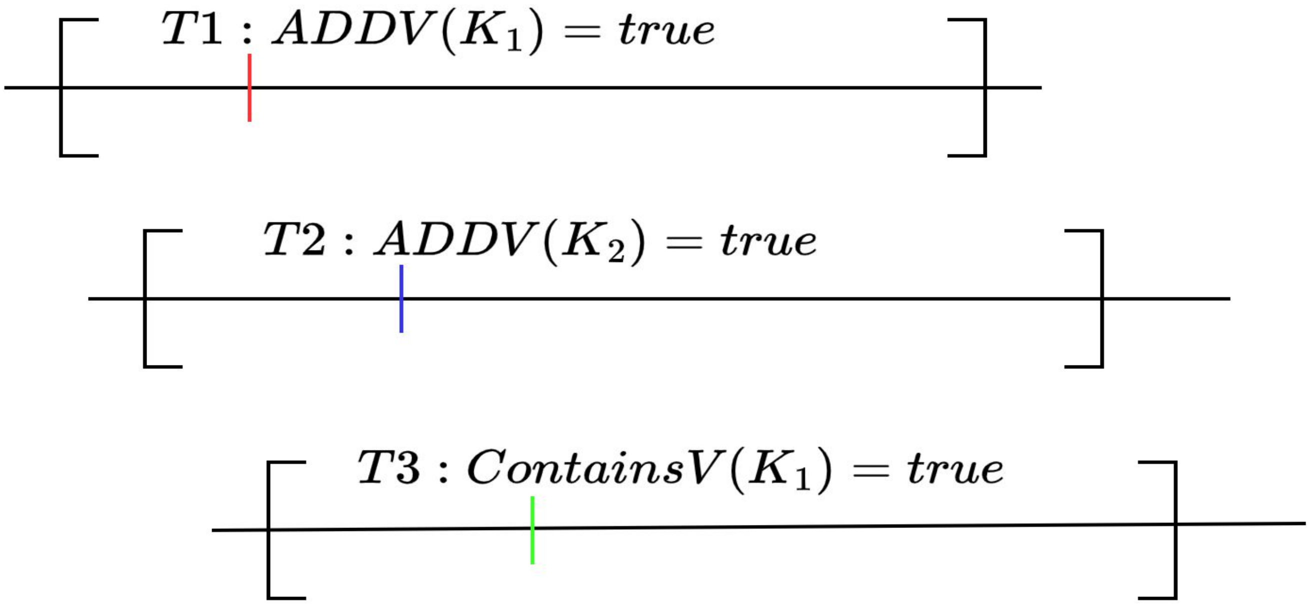
Correctness (Consistency)

- The ADT operations implemented by the data structure are represented by their invocation and return steps.
- For an arbitrary concurrent execution of a set of ADT operations should satisfy the consistency framework **Linearizability**.

Linearizability

- Assign an atomic step as a linearization point (LP) inside the execution interval of each of the operations and show that the data structure invariants are maintained across the LPs.
- An arbitrary concurrent execution is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.

Linearizability Example



Non-Blocking Progress Condition

An execution is said to be Non-Blocking if it doesn't blocks the execution of other threads.

- **Obstruction Free:** A thread is guaranteed to finish in a finite number of steps in isolation.
- **Lock-Freedom:** Atleast one thread should be able to finish in the finite number of steps.
- **Wait-Freedom:** All threads should be able to finish in a finite number of steps.

Related Work

IIT Hyderabad

- Chatterjee et al.[a]
    - Proposed a Lock-Free dynamic unbounded unweighted graph.
    - Used Lock-Free Linked Lists to store vertices and edges.
    - Used obstruction free method to perform the partial snapshot on the graph.
- Chatterjee et al.[b]
    - Extended the previous work for speed up.
    - Proposed a Lock-Free dynamic unbounded weighted graph.
    - Used Lock-Free Hash Tables to store vertices and Binary Search tree to store edges.
    - Used obstruction free method to perform the partial snapshot on the graph.
- Bhardwaj et al.[c]
    - Extended the previous work with a wait-free snapshot operation.
    - Used wait-free snapshot algorithm to perform the complete snapshot of the graph.
    - Snapshot of the graph is used to perform the graph analytics operation.

[a] **"A Simple and Practical Concurrent Non-blocking Unbounded Graph with Reachability Queries",** Bapi Chatterjee, Sathya Peri, Muktikanta Sa, Nandini Singhal in the 20th International Conference on Distributed Computing and Networking (ICDCN), Bangalore, India, January 2019.

[b] **"Non-blocking dynamic unbounded graphs with worst-case amortized bounds",** Bapi Chatterjee, Sathya Peri, Muktikanta Sa, Manogana In: International Conference on Principles of Distributed Systems (2021)

[c] **"Non-blocking Dynamic Unbounded Graphs with Wait-Free Snapshot",** Gaurav Bhardwaj, Sathya Peri, and Pratik Shetty: 25th International Symposium on Stabilization, Safety, and Security (2023)

Drawback:

- Current Literature supports obstruction free partial snapshot or wait-free complete snapshot.
- As a result, a partial snapshot operation may continuously get interrupted
  - Hence may never terminate.
- Complete Snapshot is costly as compared to the partial snapshot.
  - Graph analytics operation which requires partial snapshot still has to undergo complete snapshot.

Motivation

- To Construct a **wait-free partial** Snapshot for an Unbounded graph while allowing concurrent non-blocking (lock-free) *update* methods.

Multiversioning:

- A multiversion object maintains its previous versions, so threads can have access to the history of the object.
- Widely used in:
  - Database
  - STM
  - Concurrent Data structure
- Threads can have access to the history of the object.
- Allows to update the data structure without hindering the snapshot.

Graph Data Structure:

- Each vertex in the graph is stored as **Vnode** in a concurrent lock-free resizable hash table.
- Each **Vnode** has a key and a pointer to the **vertex version** list.
- Each node (**Vnode_version**) in the vertex version list contains the pointer to the corresponding edge list stored in a lock-free linked list.
- Each **Vnode_version** node contains the insert timestamp and a delete timestamp.

```
int global_ts

class VNode{
  int k
  Vnode_version* vhead
}
```

```
class Vnode_version{
  int k
  int insert_ts
  int delete_ts
  Enode* elist
  Vnode_vesion* next_ver
}
```

Graph Data Structure:

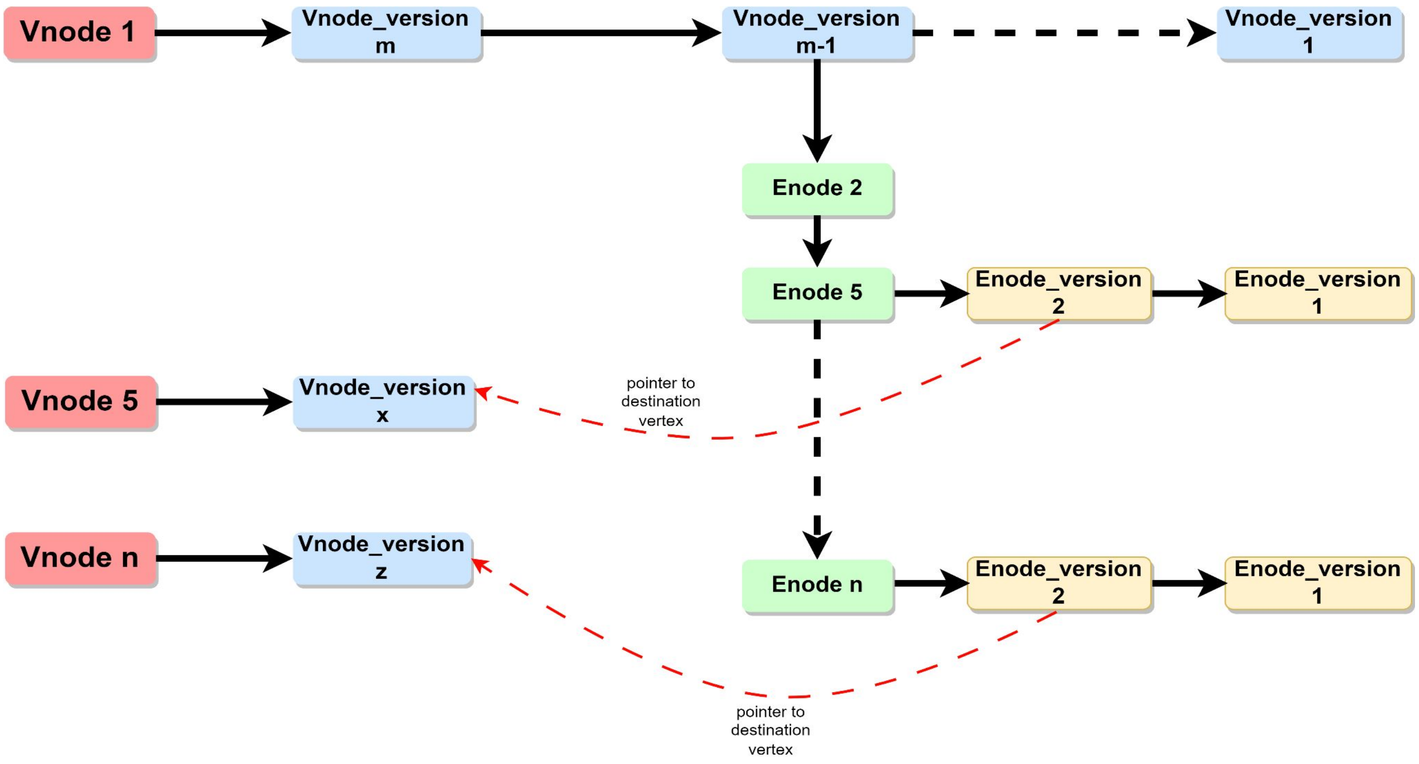
IIT Hyderabad

- Each edge is stored as **Enode** in a concurrent lock-free linked list (edge list) associated with each version node.
- Enode stores the key of the destination vertex and the pointer to the edge version list.
- Each node (Enode_version) in the edge version list contains weight, insert timestamp and delete timestamp.
- Enode_version points to the next version and also stores the reference to the destination Vnode_version.
-

```
class ENode{
  int l
  ENode* enext
  Enode_version* ehead
}
```

```
class Enode_version{
  int weight
  int insert_ts
  int delete_ts
  Vnode_version* dest_ver_node
  Enode_version* next_ver
}
```

Graph Data Structure

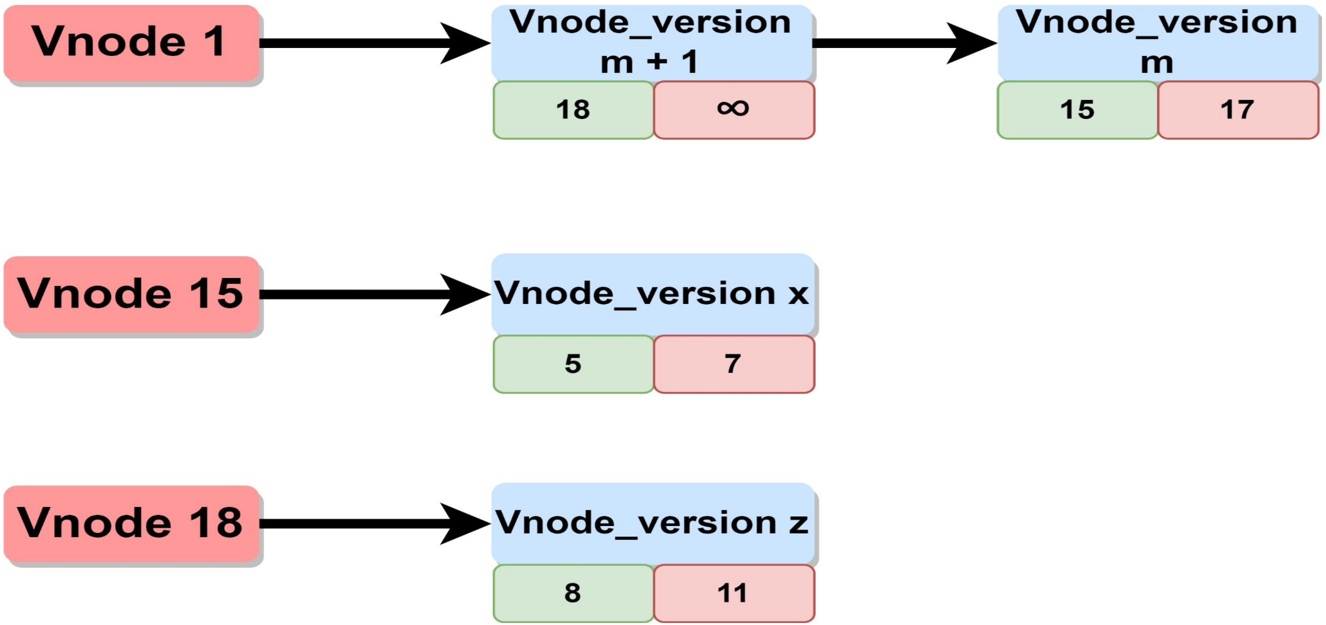


ADT:

- AddVertex(k)
- RemoveVertex(k)
- AddEdge(i, j,w)
- RemoveEdge(i, j)
- ContainsVertex(k)
- ContainsEdge(i, j)
- BFS(v)
- SSSP(v)
- Snapshot()

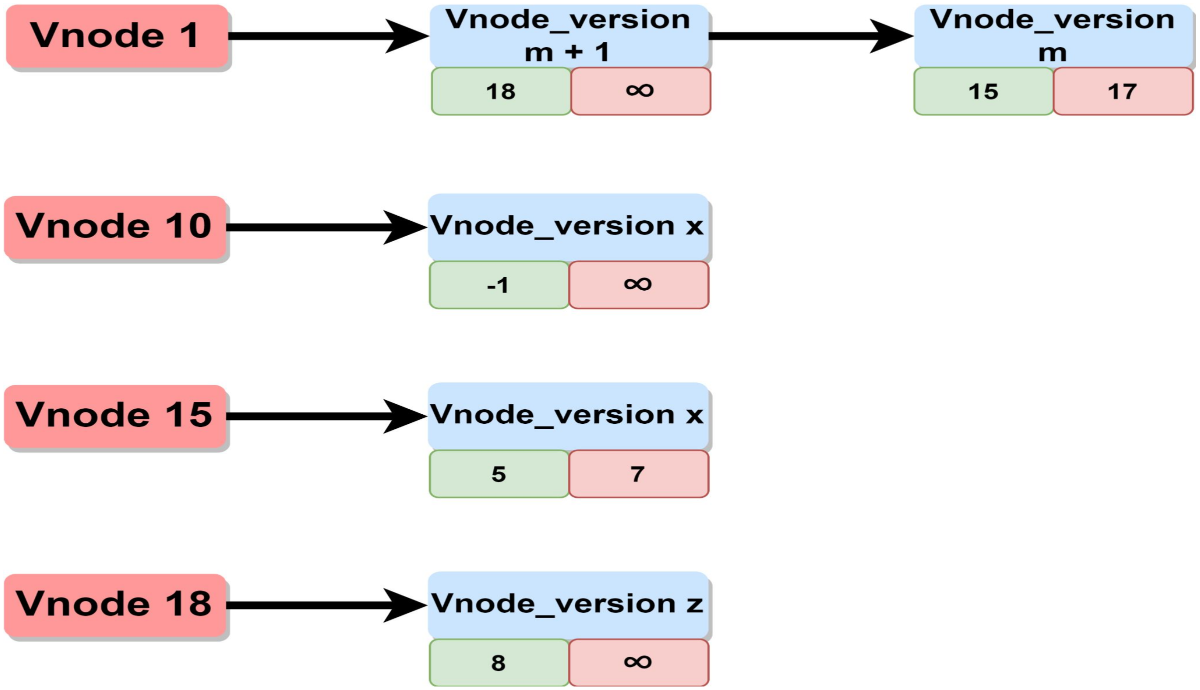
Addvertex(10) : Case-1

- Vnode 10 doesn't Exists



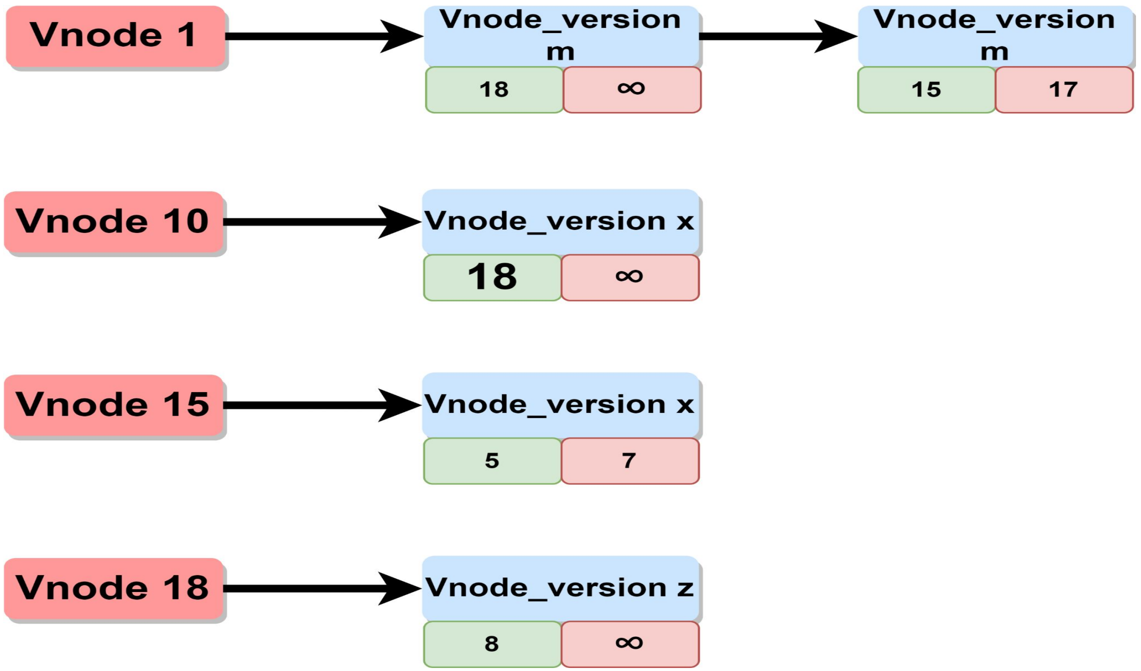
Addvertex(10) : Case-1

- Vnode 10 is inserted with insert_ts initialized to -1



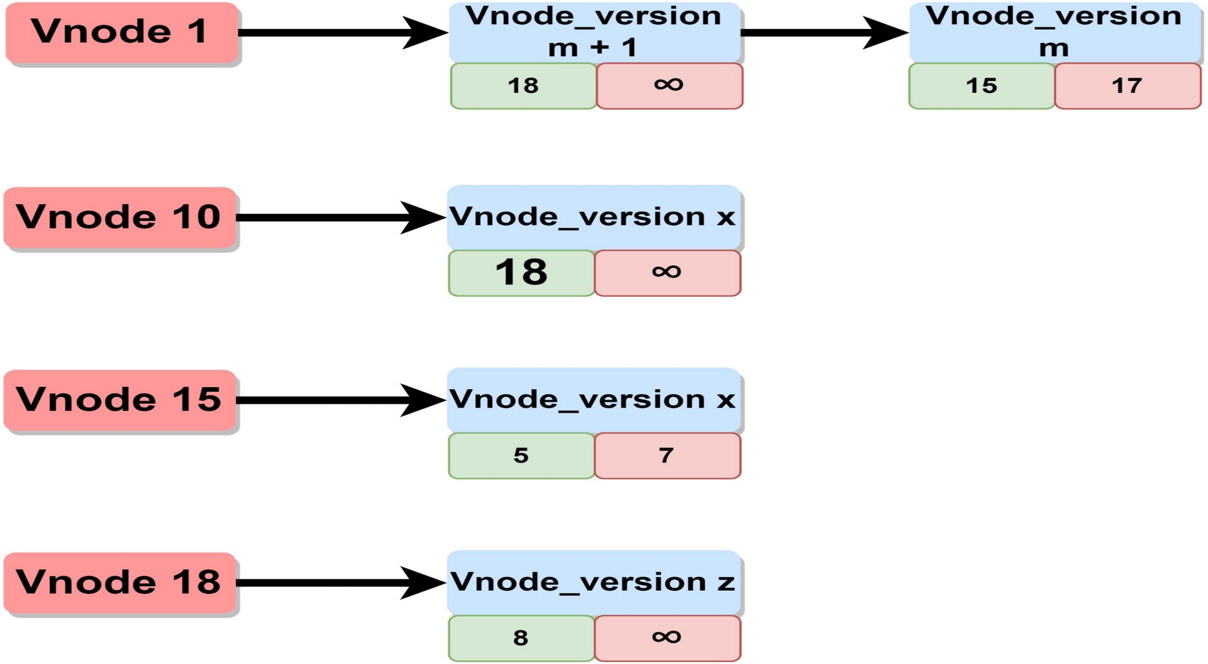
Addvertex(10) : Case-1

- Insert timestamp is set to current global timestamp



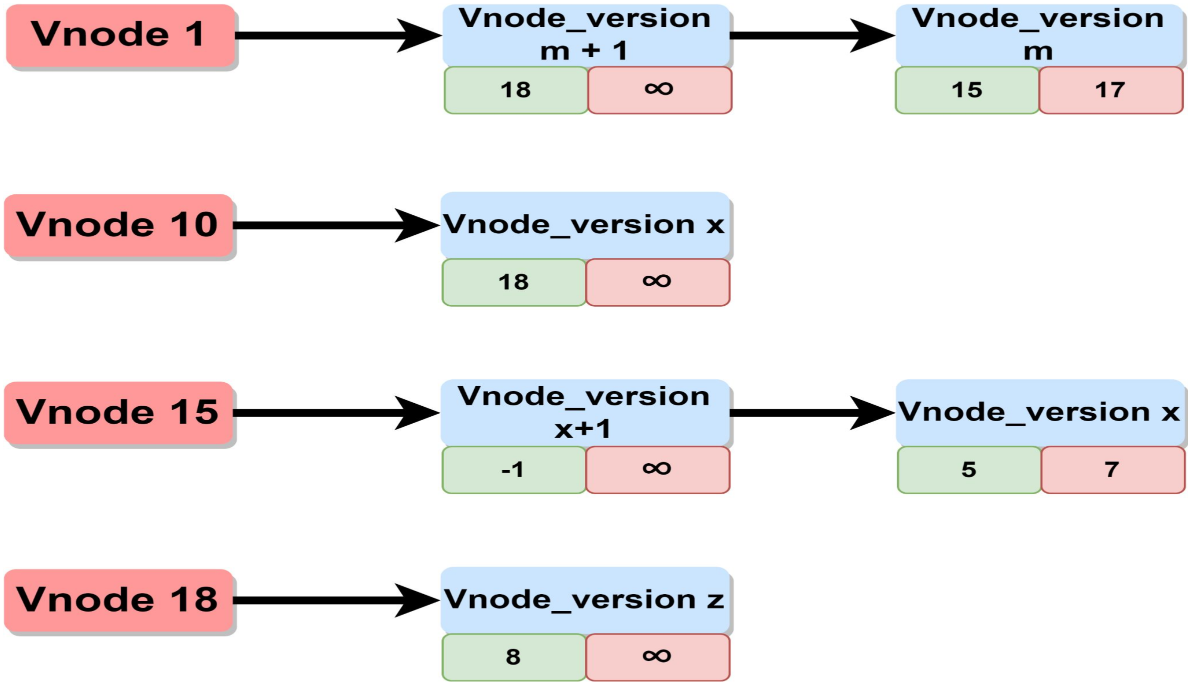
Addvertex(15) : Case-2

- Case2: When Vnode exists but there is no active version.



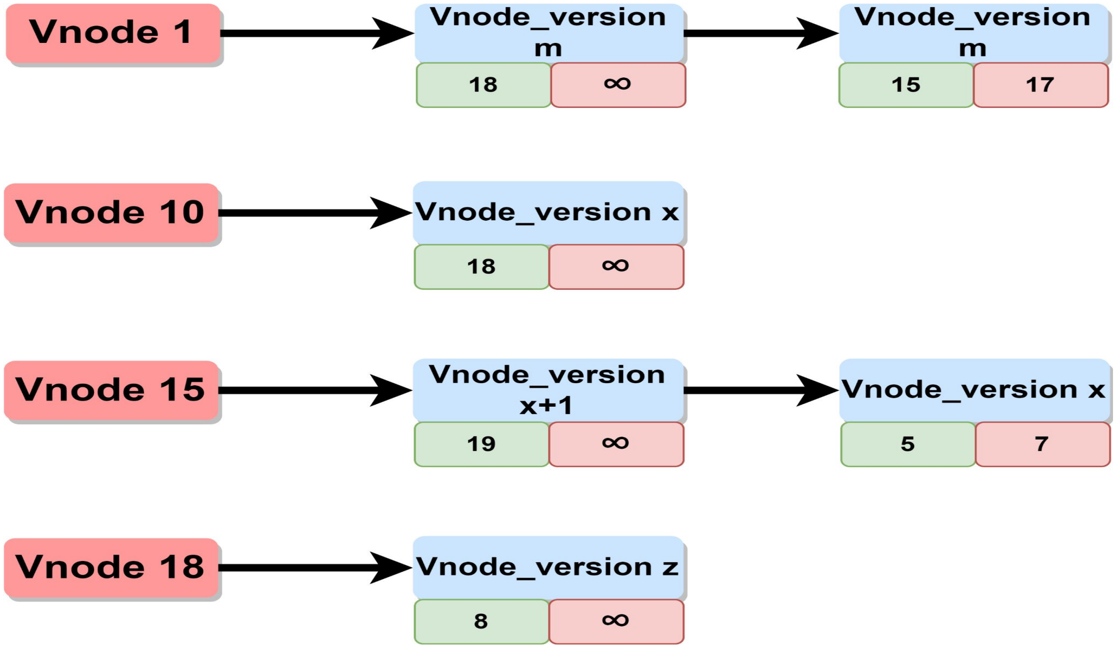
Addvertex(15) : Case-2

- A new Vnode_version is added on the top of the list with insert timestamp as -1



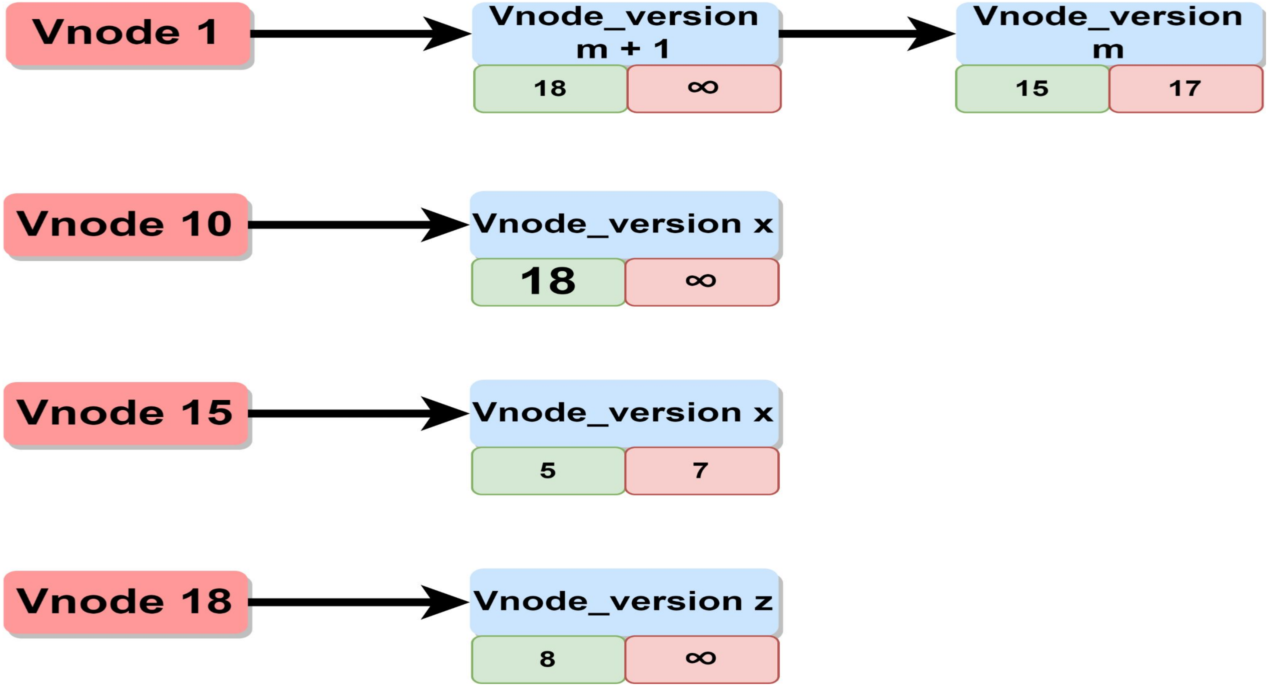
Addvertex(15) : Case-2

- Insert timestamp is set to current global timestamp



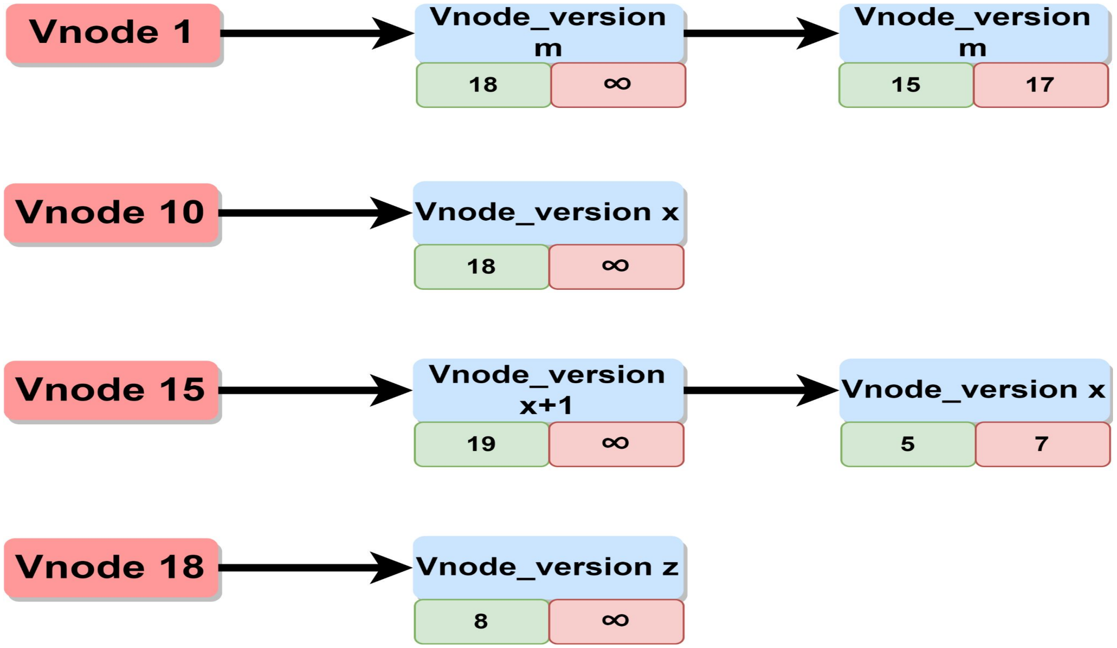
Addvertex(18) : Case-3

- Already exists.



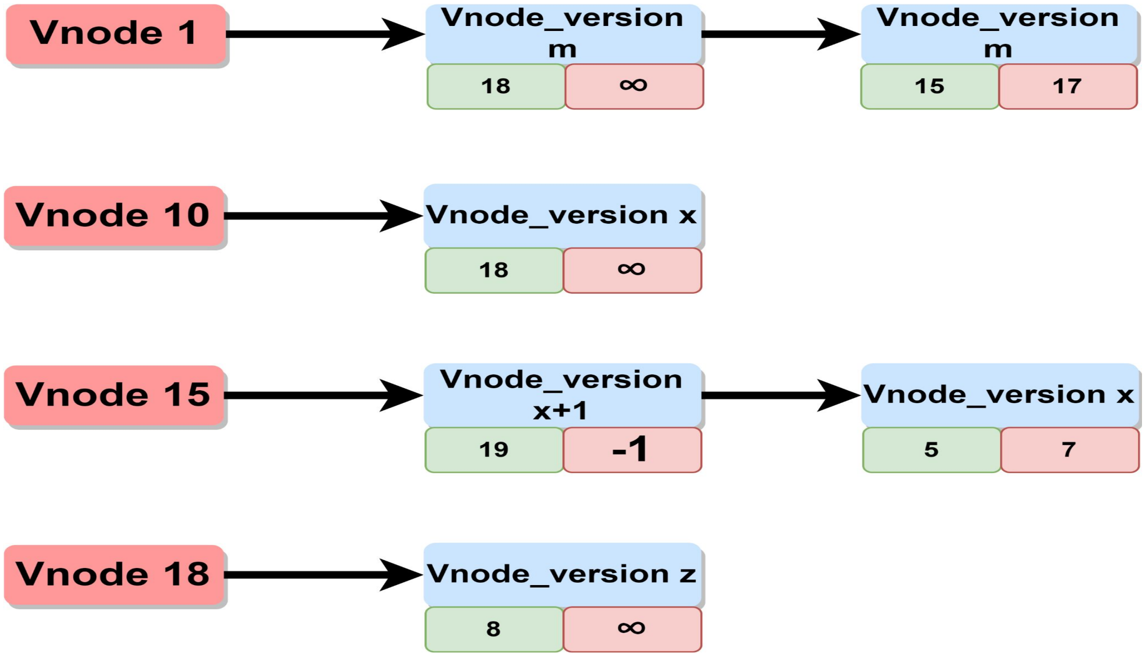
Remove Vertex(15) : Case-1

- Active version_vertex exists.



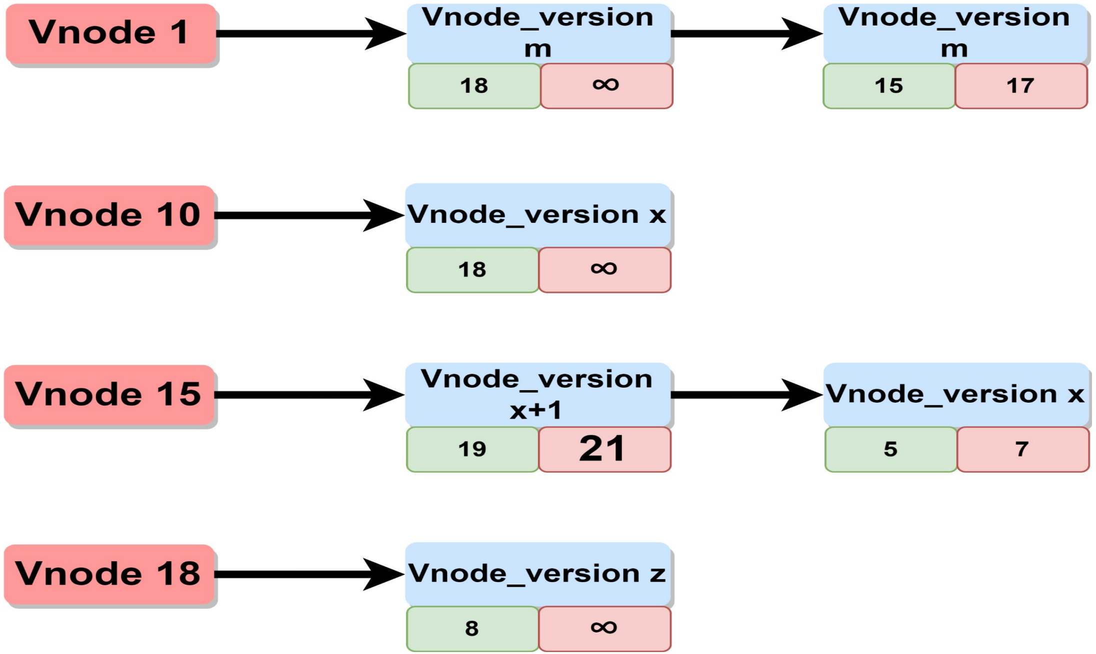
Remove Vertex(15) : Case-1

- Set the delete timestamp to - 1.



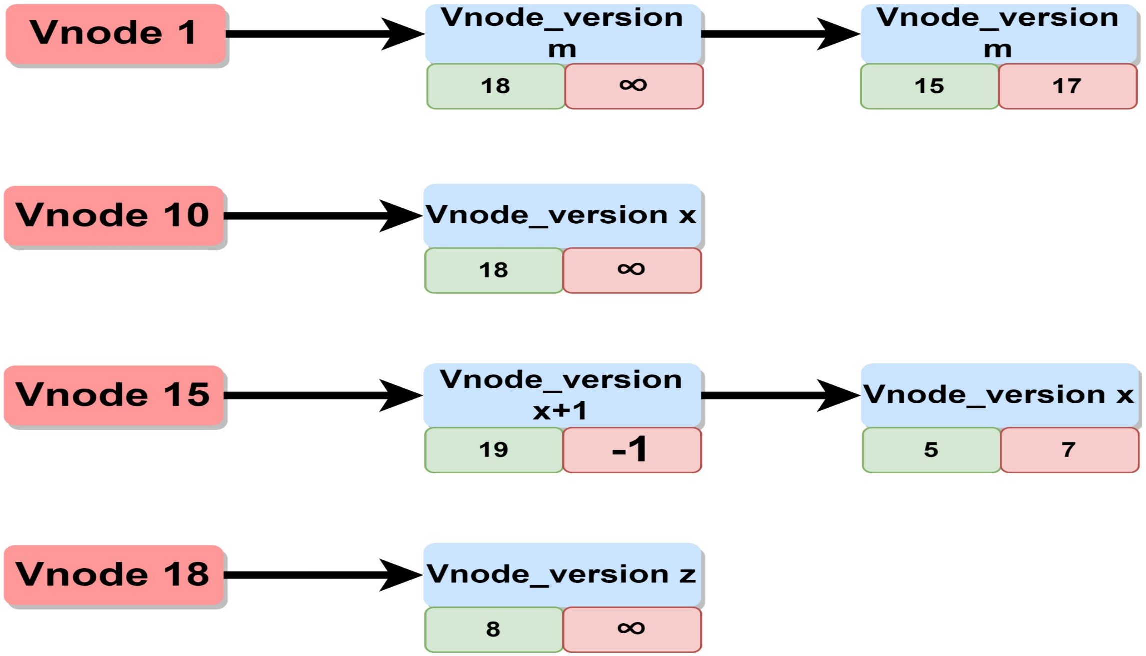
Remove Vertex(15) : Case-1

- Set the delete timestamp to current global timestamp.



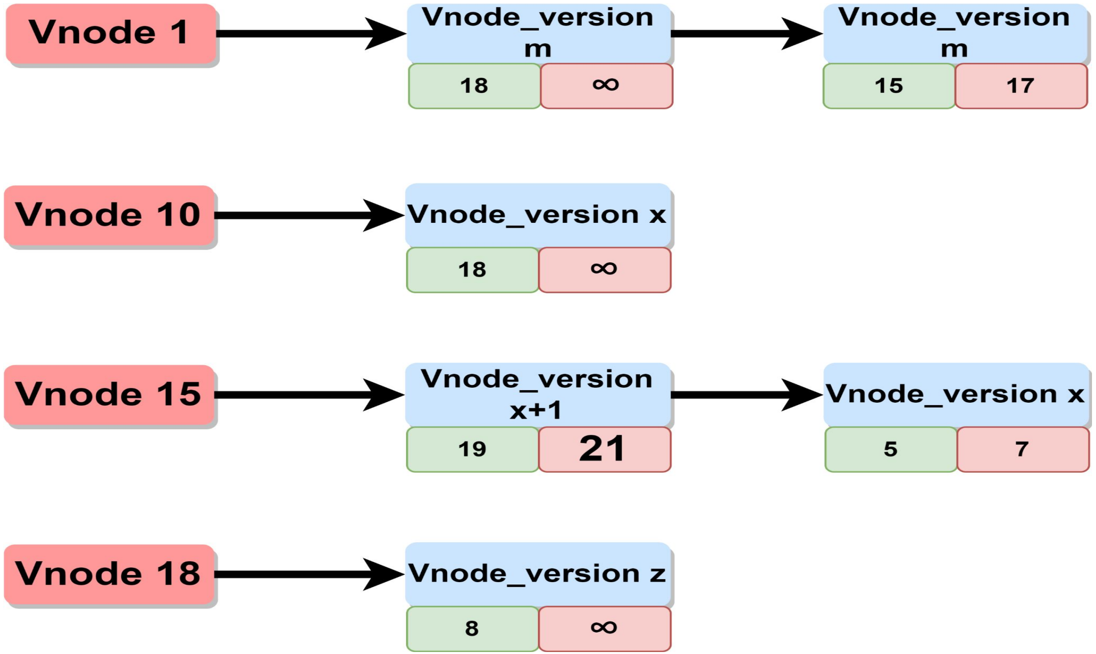
Remove Vertex(15) : Case-2

- Set TS if -1.



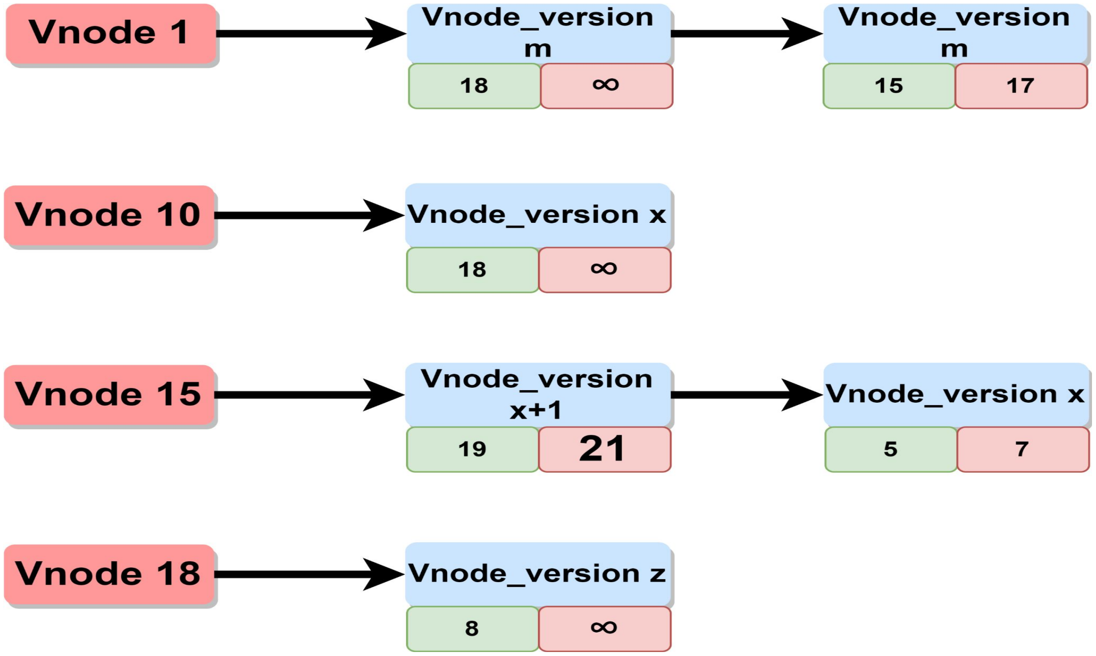
Remove Vertex(15) : Case-2

- Set the delete timestamp to current global timestamp.



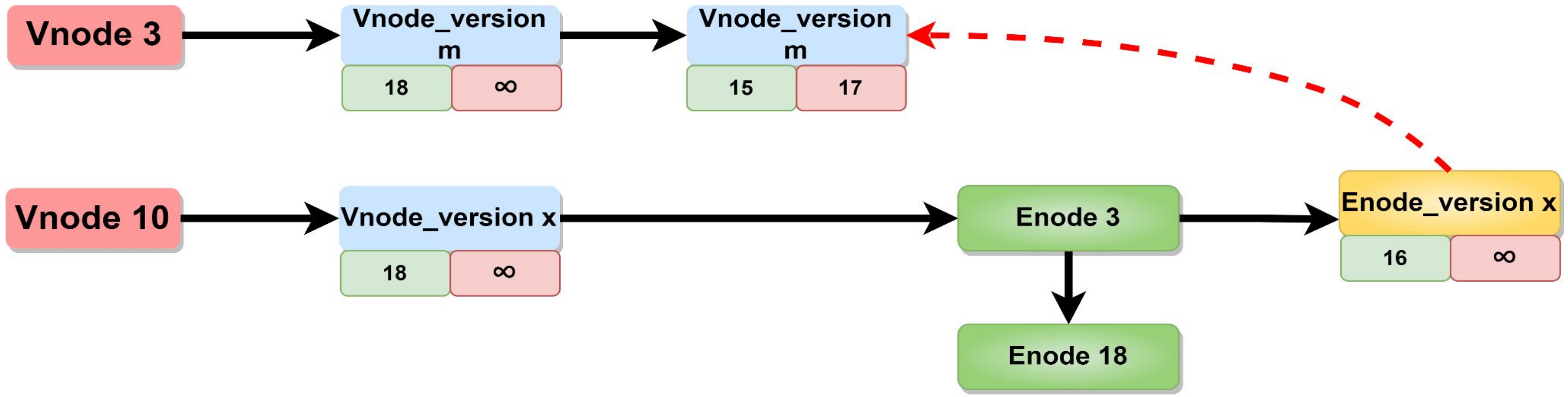
Remove Vertex(15) : Case-3

- Delete Timestamp of the vertex is already set



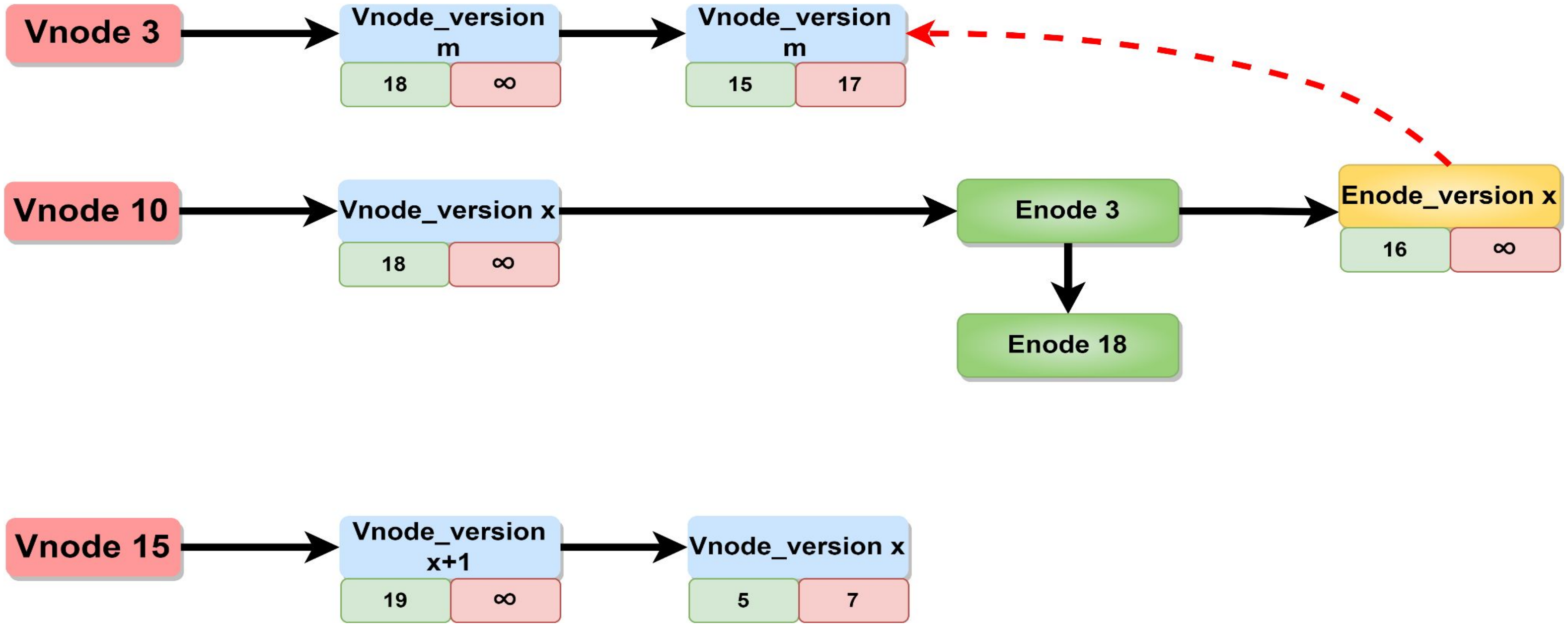
Add Edge(10,15) : Case-1

Case1: Vertex 15 doesn’t exists.



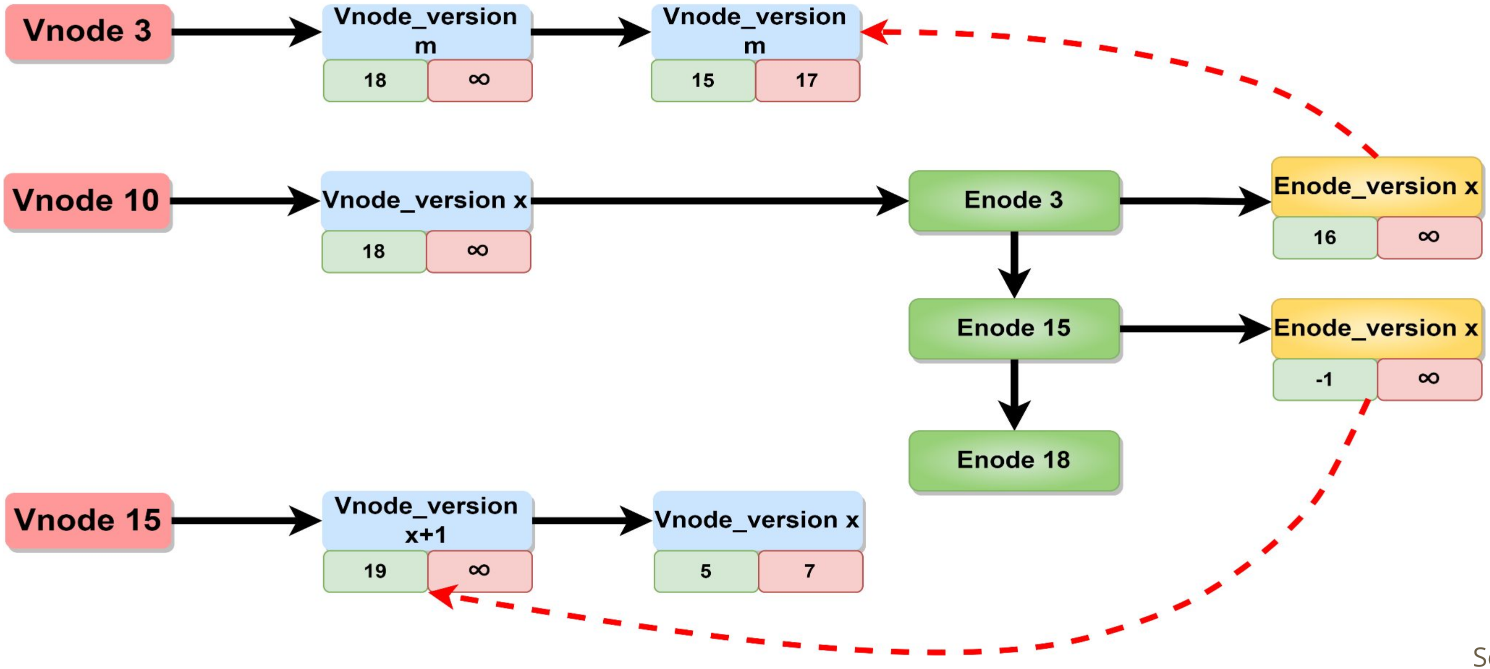
Add Edge(10,15) : Case-2

Case2: Enode(10,15) doesn’t exists.



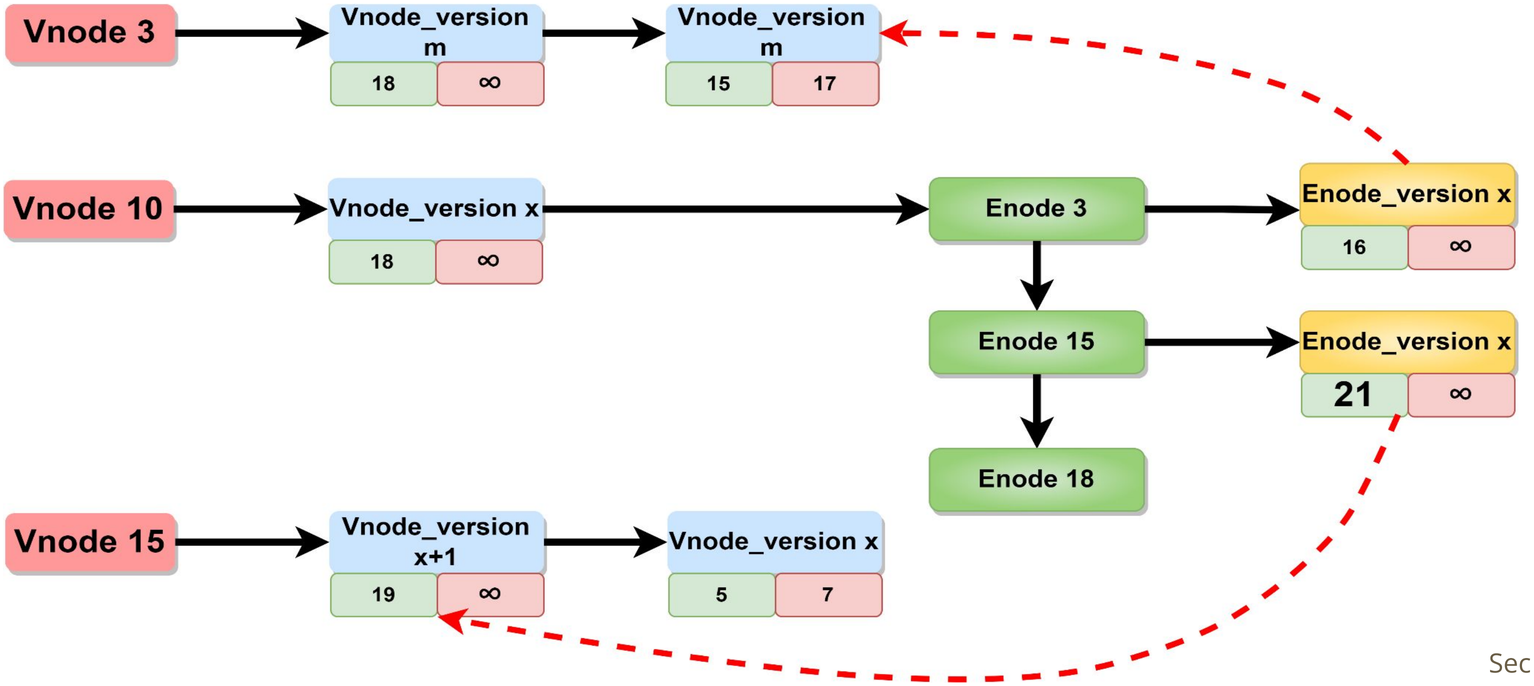
Add Edge(10,15) : Case-2

Case2: New Enode is created with insert timestamp as -1.



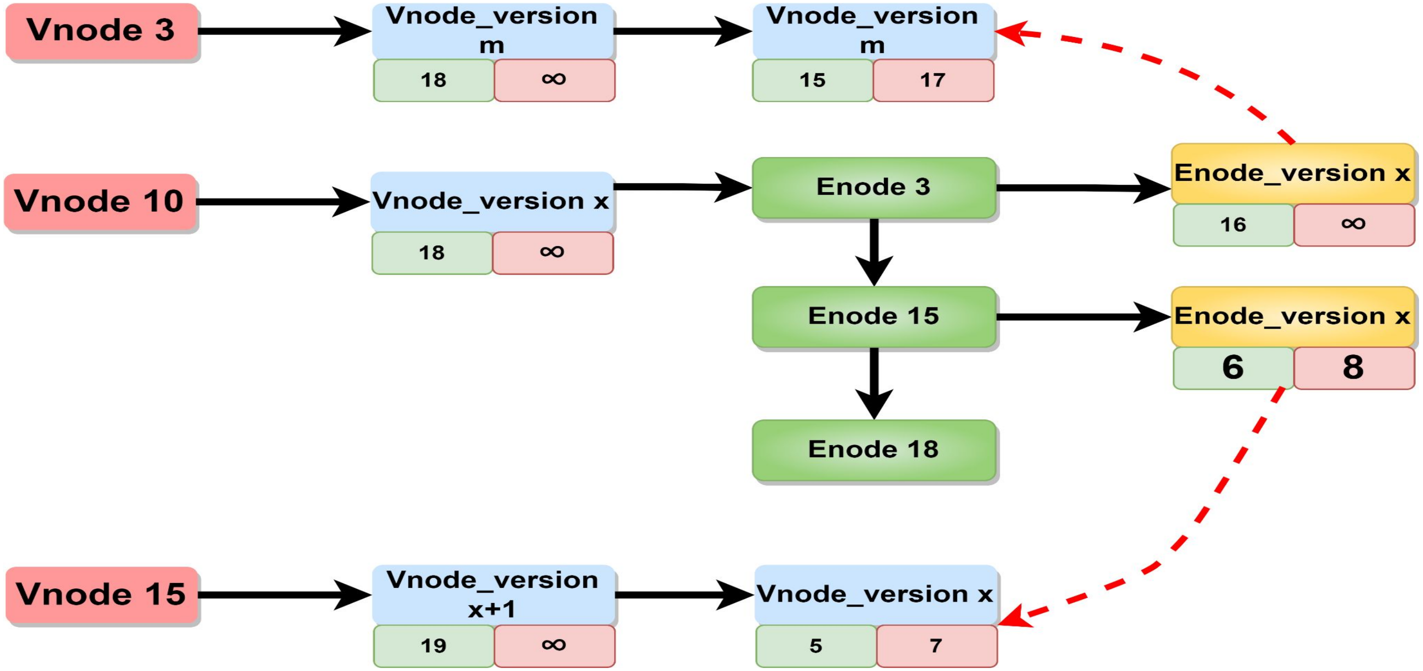
Add Edge(10,15)

Case2: Set insert timestamp to current timestamp.



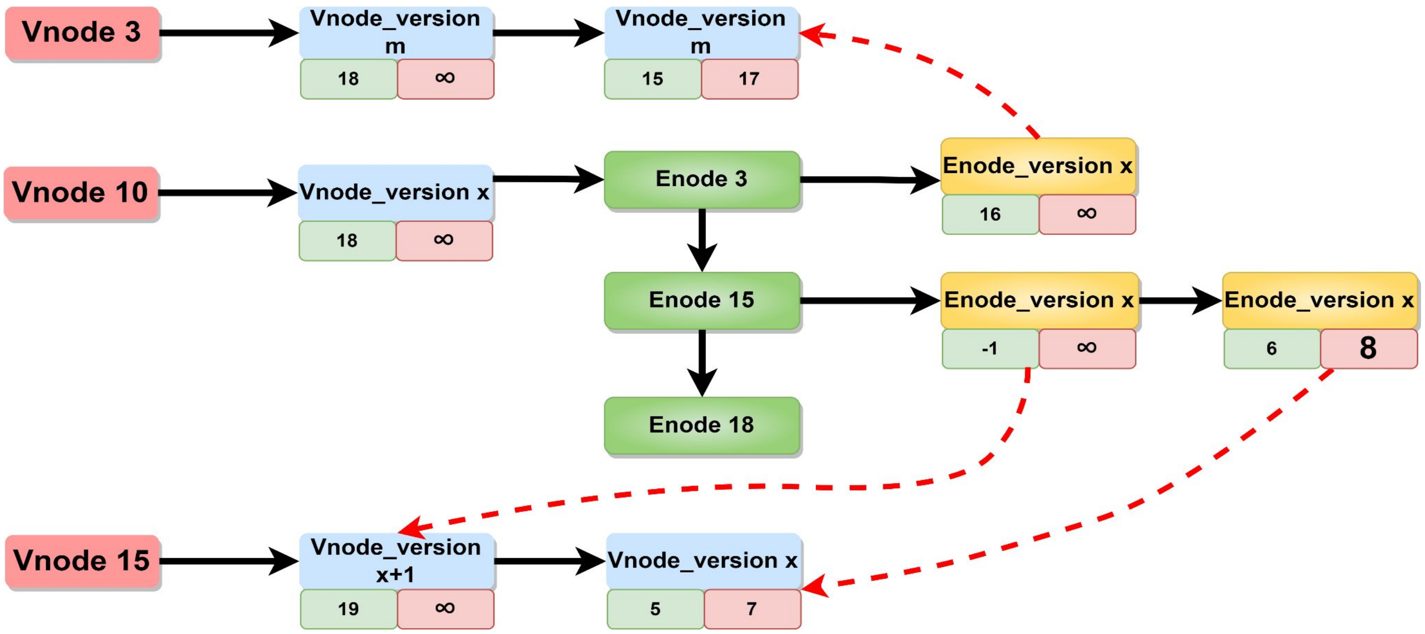
Add Edge(10,15) : Case-3

Case3: Adding a new Edge Version.



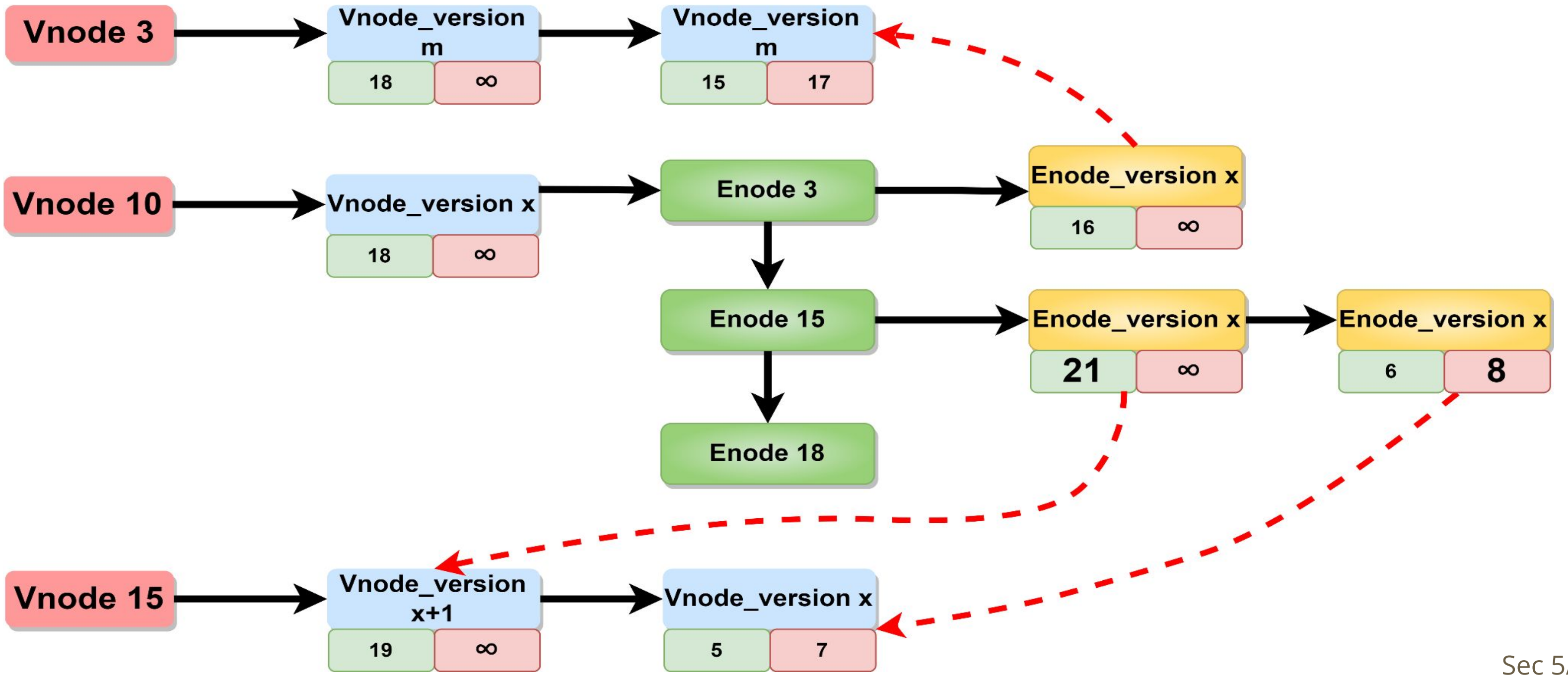
Add Edge(10,15) : Case-3

Case3: Adding a new Edge Version.



Add Edge(10,15) : Case-3

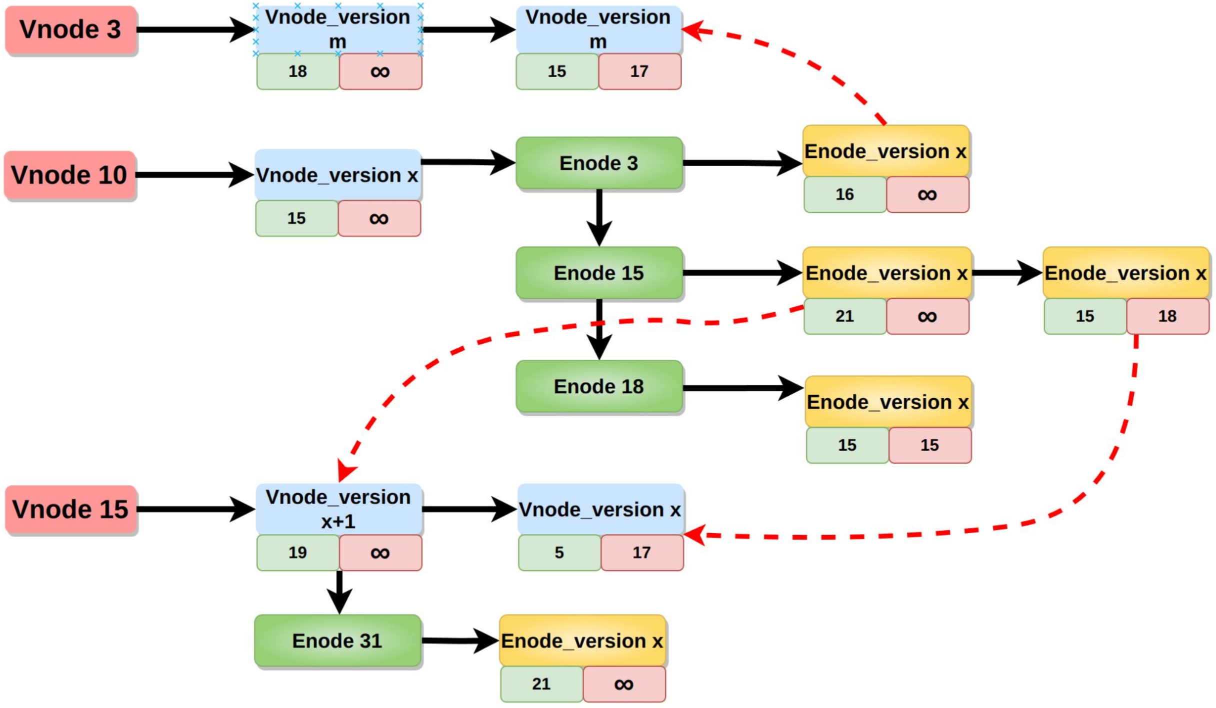
Case3: Adding a new Edge Version.



BFS(vertex 10) with TS 16

Current Partial

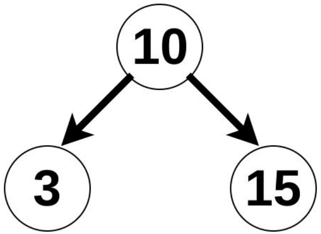
Snapshot

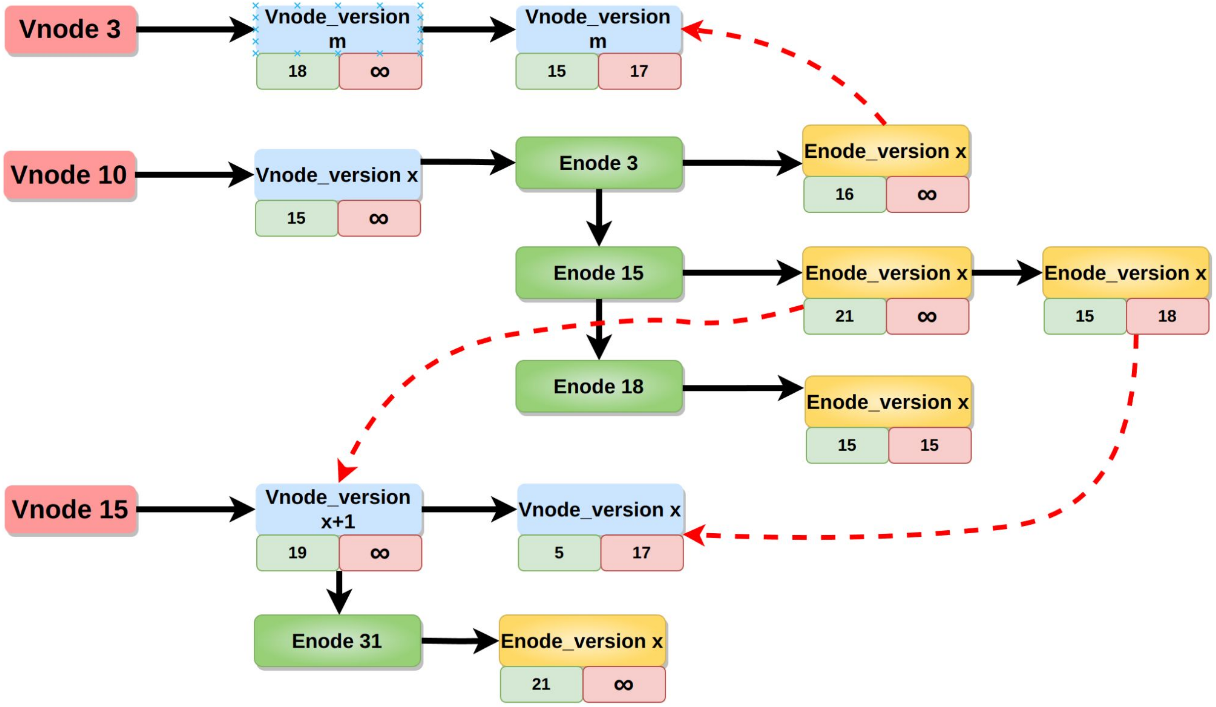


BFS(vertex 10) with TS 16

Current Partial

Snapshot





Memory Reclamation

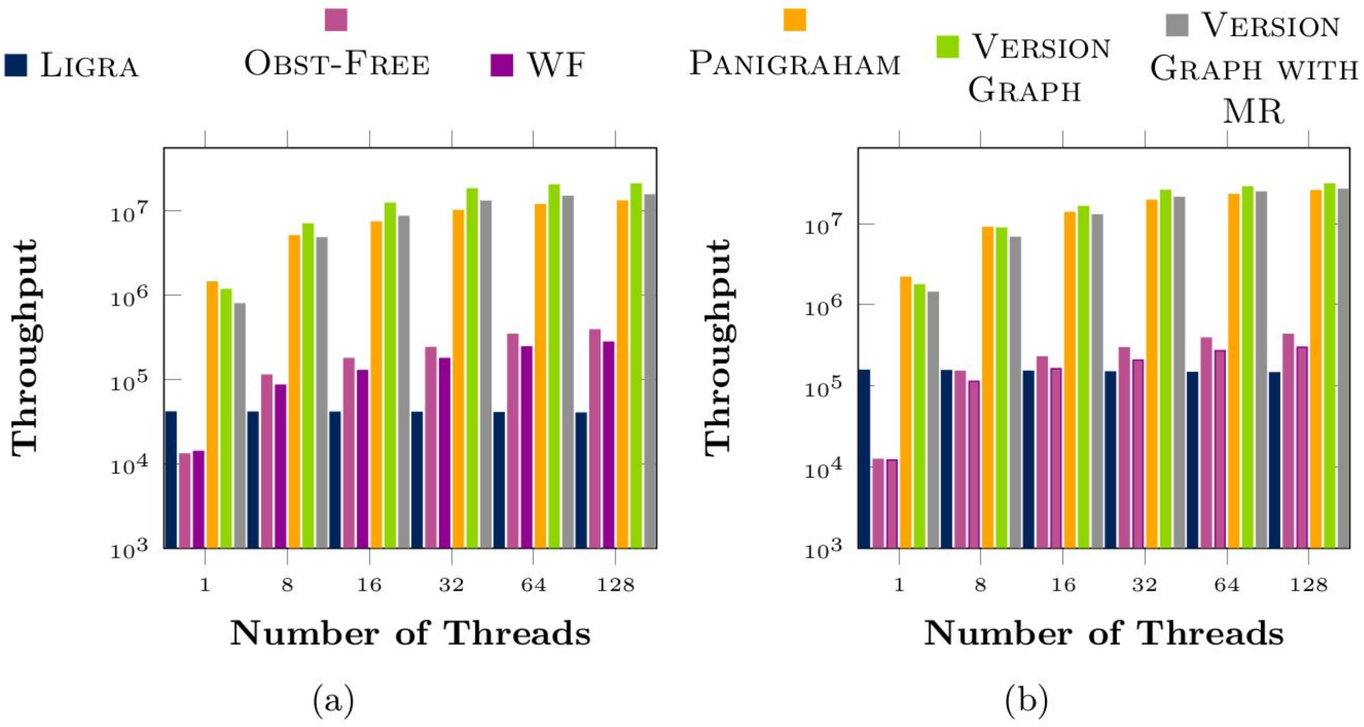
- **Step-1:** Capture the smallest active snapshot timestamp among all the threads, say T.
- **Step-2:** Iterate the complete data structure and pass all the edge nodes having timestamp smaller than or equal to T-1 to DEBRA.
- **Step-3:** After removing all the reference to vertex nodes from edge nodes in step-2, re-iterate the complete data structure and remove all the vertex nodes having timestamp smaller than or equal to T-1 to DEBRA.

Results and Analysis :

- **Workload Distribution :** The distribution is over the following ordered set of Operations (AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge, and Critical Operation(Snapshot/BFS/Betweenness Centrality).
  - Read Heavy Workload : 3%, 2%, 44%, 3%, 2%, 44% , 2%
  - Update Heavy Workload: 13%, 12%, 24%, 13%, 12%, 24% , 2%

- **Algorithms :** We compare our implementation to Obstruction-free and wait-free implementations of same operations namely
  - **Obst-Free: "A Simple and Practical Concurrent Non-blocking Unbounded Graph with Reachability Queries",** Bapi Chatterjee, Sathya Peri, Muktikanta Sa, Nandini Singhal in the 20th International Conference on Distributed Computing and Networking (ICDCN), Bangalore, India, January 2019.
  - **PANIGRAHAM: "Non-blocking dynamic unbounded graphs with worst-case amortized bounds",** Bapi Chatterjee, Sathya Peri, Muktikanta Sa, Manogana In: International Conference on Principles of Distributed Systems (2021)
  - **Graph_Iterator: "Non-blocking Dynamic Unbounded Graphs with Wait-Free Snapshot",** Gaurav Bhardwaj, Sathya Peri, and Pratik Shetty: 25th International Symposium on Stabilization, Safety, and Security (2023)
  - **LIGRA**

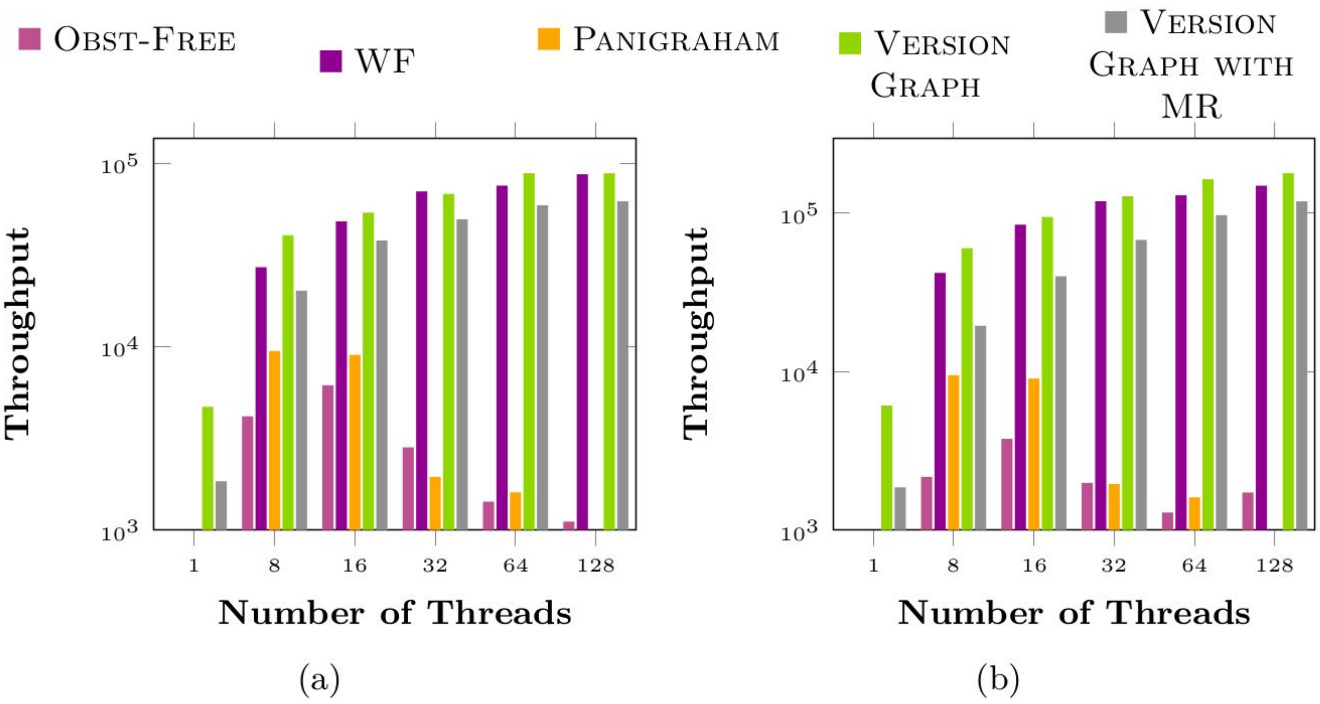
Results: Point Operation



Update Heavy Workload

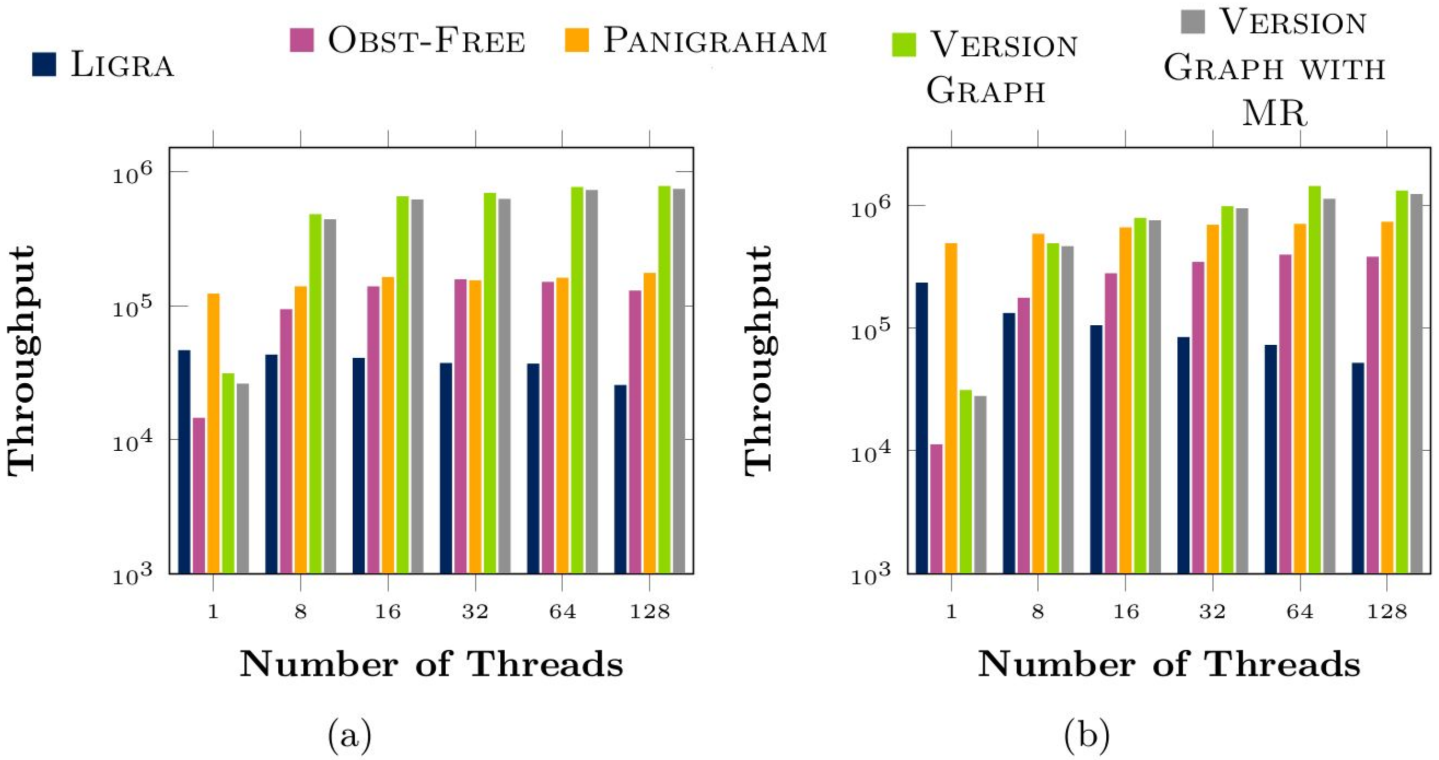
Read Heavy Workload

Results: Snapshot



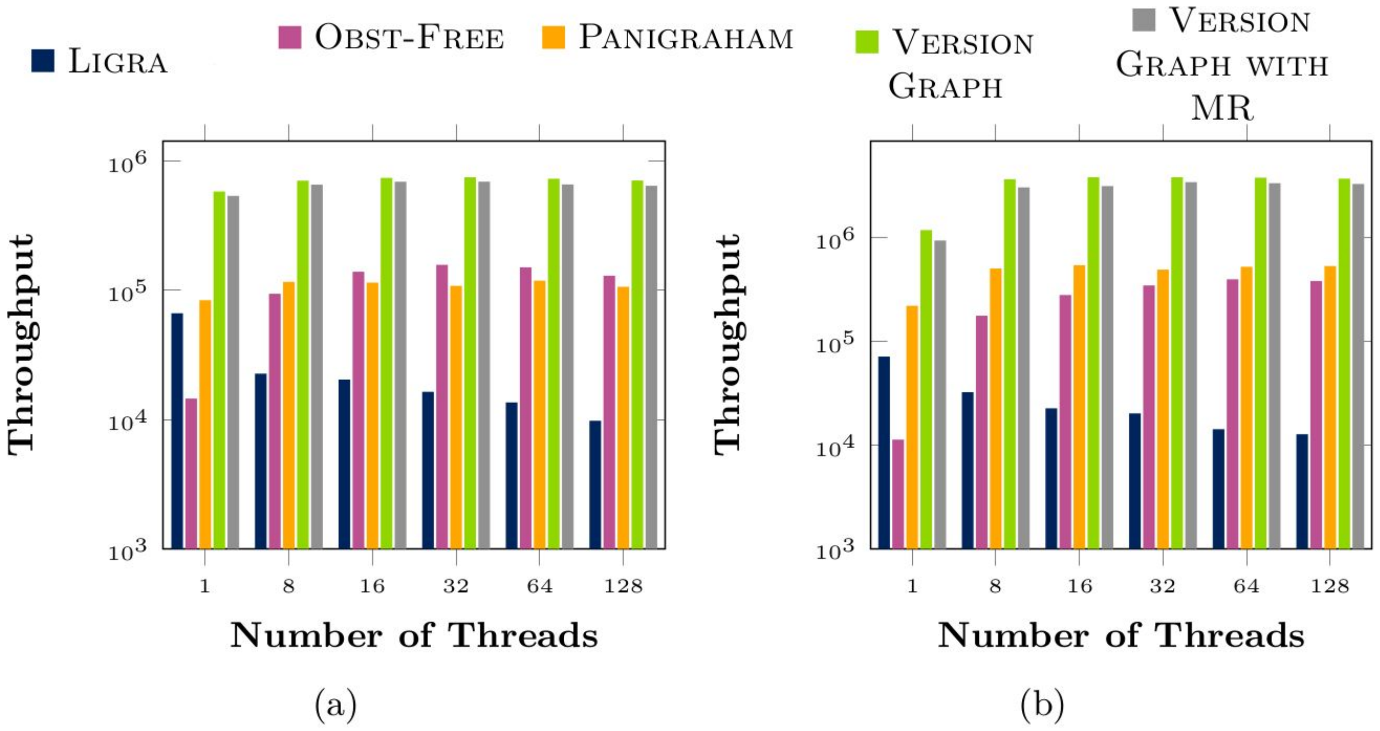
Update Heavy Workload **Read Heavy Workload**

Results: BFS



Update Heavy Workload **Read Heavy Workload**

Result: SSSP



Update Heavy Workload

Read Heavy Workload

Conclusion:

- First algorithm which supports partial wait-free snapshot for the graph.
- Significant improvement in the performance when compared to its counterparts.


Thank
You