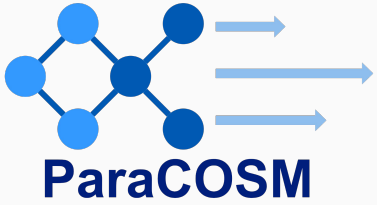


INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

ParaCOSM: A Parallel Framework for Continuous Subgraph Matching

Haibin Lai, Sicheng Zhou, Site Fan, Zhuozhao Li*

Sep 10, 2025

Southern University of Science and Technology

**Corresponding Author*

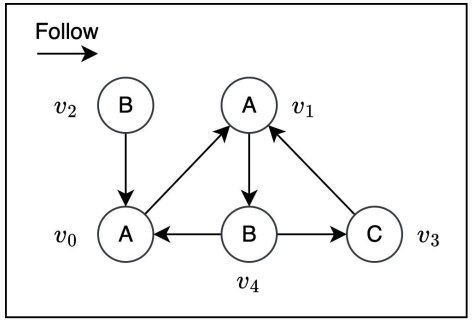
SUSTech Southern University of Science and Technology

HPC LAB

Streaming Graph Processing

Streaming Graph Model

Graph (label/unlabel) **G(V, E, L)** keeps on changing via a stream of edge/vertex updates



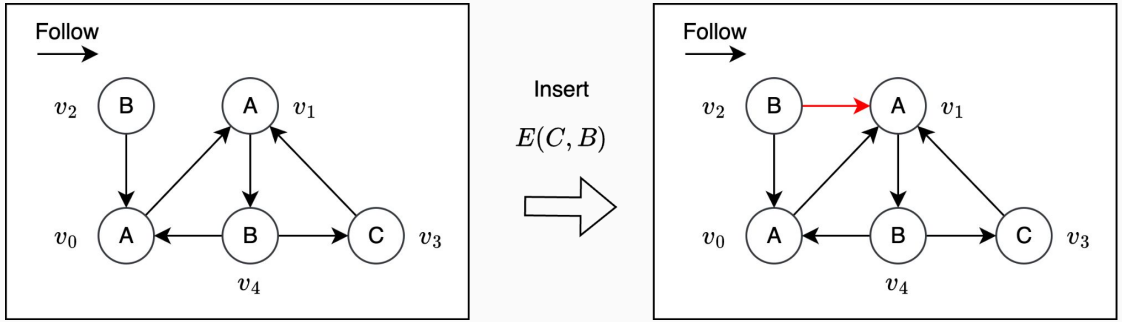
Main Usage

- Financial transaction
- Social network

Streaming Graph Processing

Streaming Graph Model

Graph (label/unlabel) **G(V, E, L)** keeps on changing via a stream of edge/vertex updates



Main Usage

- Financial transaction
- Social network

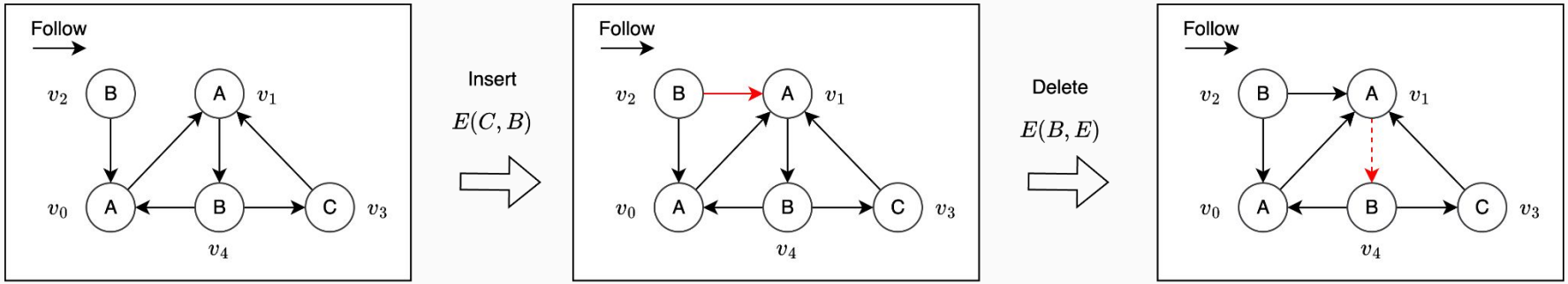
Streaming Graph Processing

Streaming Graph Model

Graph (label/unlabel) **G(V, E, L)** keeps on changing via a stream of edge/vertex updates

Main Usage

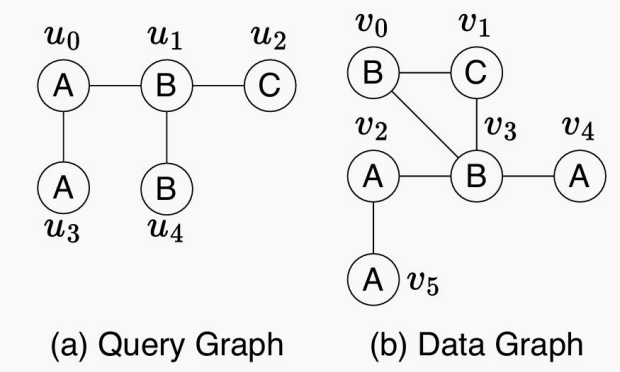
- Financial transaction
- Social network



Introduction to Continuous Subgraph Matching

Continuous Subgraph Matching Problem:

Incrementally identifies matches of Q with in G in response to each update $\Delta G \in \Delta G$

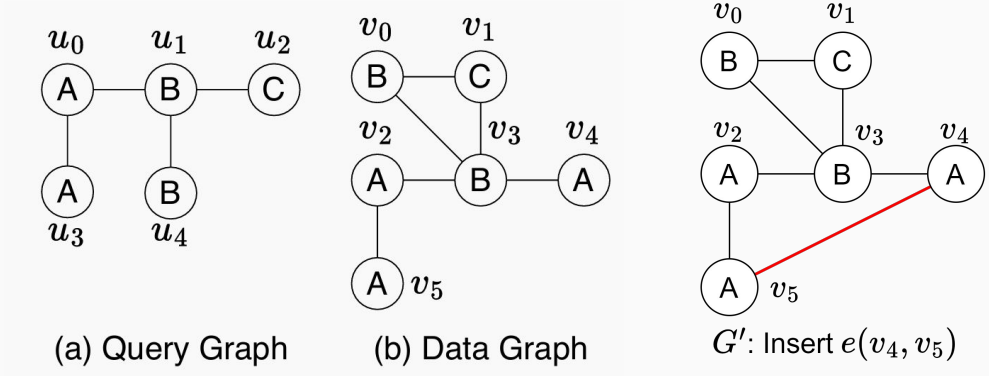


(a) Query Graph (b) Data Graph

Introduction to Continuous Subgraph Matching

Continuous Subgraph Matching Problem:

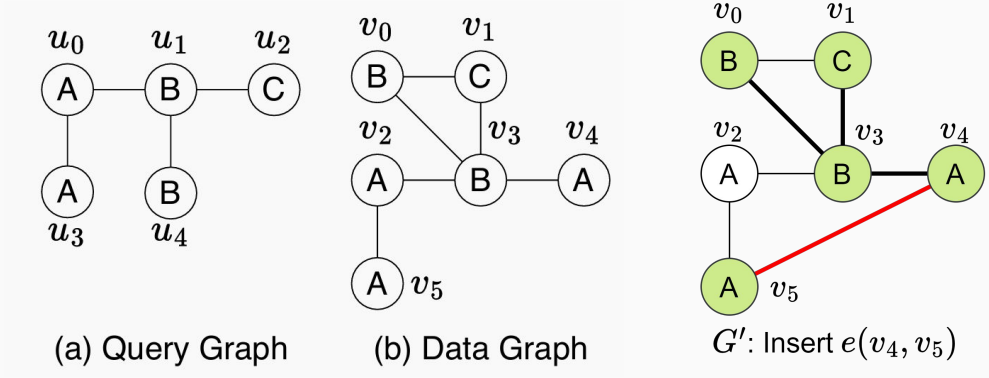
Incrementally identifies matches of Q with in G in response to each update $\Delta G \in \Delta G$



Introduction to Continuous Subgraph Matching

Continuous Subgraph Matching Problem:

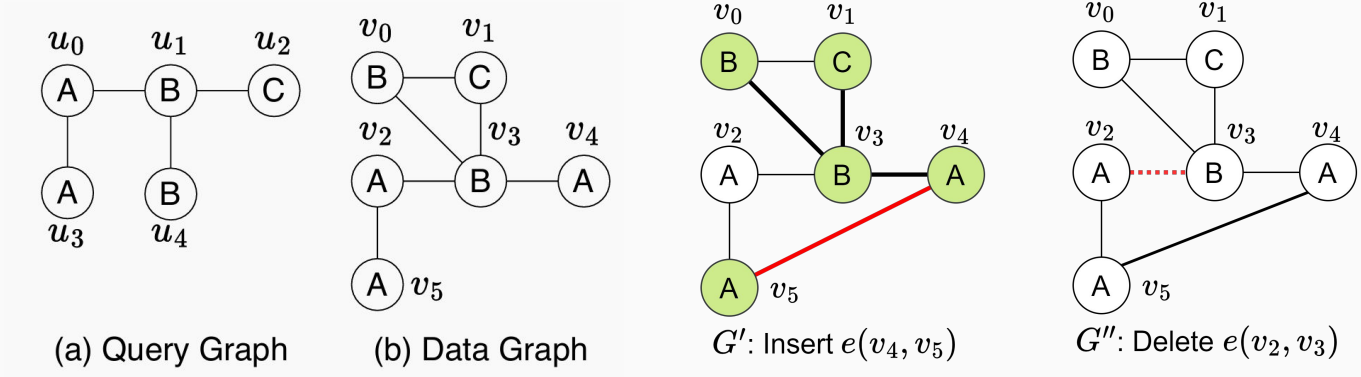
Incrementally identifies matches of Q with in G in response to each update $\Delta G \in \Delta G$



Introduction to Continuous Subgraph Matching

Continuous Subgraph Matching Problem:

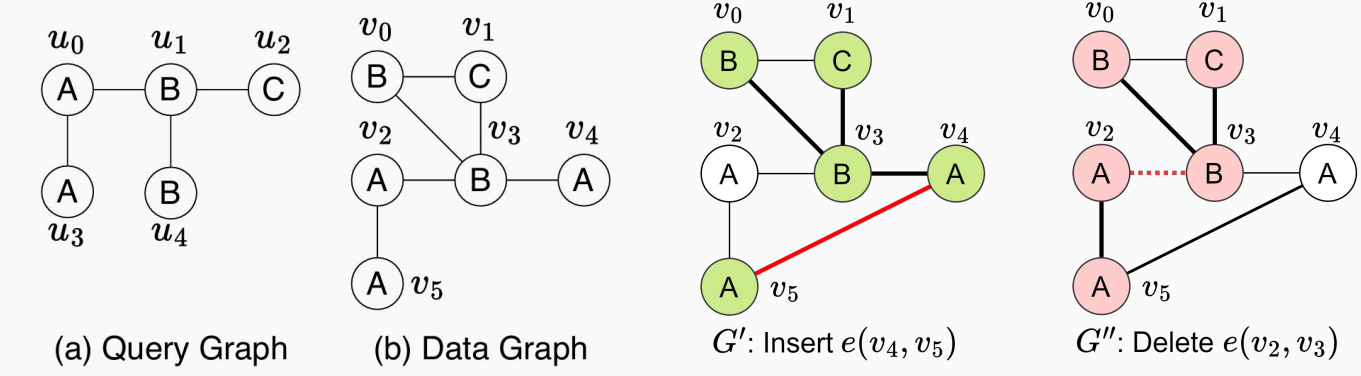
Incrementally identifies matches of Q with in G in response to each update $\Delta G \in \Delta G$



Introduction to Continuous Subgraph Matching

Continuous Subgraph Matching Problem:

Incrementally identifies matches of Q with in G in response to each update $\Delta G \in \Delta G$



(a) Query Graph (b) Data Graph

G': Insert $e(v_4, v_5)$

G'': Delete $e(v_2, v_3)$

Existing CSM Algorithms

- **Static Subgraph matching**
  - Examples: IncIsoMatch, VF2.
  - Limitation: Re-run static matching in a long period
- **Single-thread CSM**
  - Examples: RapidFlow, NewSP etc.
  - Limitation: Lack of multi-core utilization
- **Batched CSM**
  - Examples: Mnemonic etc.
  - Limitation: Workload imbalance
- **GPU-based CSM**
  - Examples: GAMMA etc.
  - Limitation: Extensive hardware cost + code efforts

Table 1: Existing CSM solutions in recent research. Para: Parallelism. Srch: search method (✓ = backtrack, ✗ = join-based).

System	Para	index $\mathcal{A}$ update	Find Matches	Srch
		CPU Algorithms		
IncIsoMatch [9]	✗	Recomputation	N/A	✓
SJ-Tree [6]	✓	$O(\|E(G)\|^{\|E(Q)\|})$	$O(\|E(G)\|^{\|E(Q)\|})$	✗
Graphflow [15]	✓	O(1)	$O(d(G)^{\|V(Q)\|})$	✗
TurboFlux [16]	✗	$O(\|E(G)\|\|V(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
IEDyn [14]	✗	$O(\|E(G)\|\|V(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
Symbi [20]	✗	$O(\|E(G)\|\|E(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
RapidFlow [25]	✓[1]	$O(\|E(G)\|\|E(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
Mnemonic [1]	✓	O(1)	$O(d(G)^{\|V(Q)\|})$	✓
CaLiG [32]	✗	$O(\|E(G)\|\|E(Q)\|)$	$O(\|V(G)\|^{K})^{2}$	✓
NewSP [18]	✗	O(1)	$O(d(G)^{\|V(Q)\|})$	✓
		GPU Algorithms		
GAMMA [23]	✓	O(1)	$O(d(G)^{V(Q)-1}$ $d(Q)\log(V))$	✗
GCSM [30]	✓	N/A	N/A	✗

[1] RapidFlow has been parallelized in [30].

[2] K is the number of kernel vertices.

Existing CSM Algorithms

- **Static Subgraph matching**
  - Examples: IncIsoMatch, VF2.
  - Limitation: Re-run static matching in a long period
- **Single-thread CSM**
  - Examples: RapidFlow, NewSP etc.
  - Limitation: Lack of multi-core utilization
- **Batched CSM**
  - Examples: Mnemonic etc.
  - Limitation: Workload imbalance
- **GPU-based CSM**
  - Examples: GAMMA etc.
  - Limitation: Extensive hardware cost + code efforts

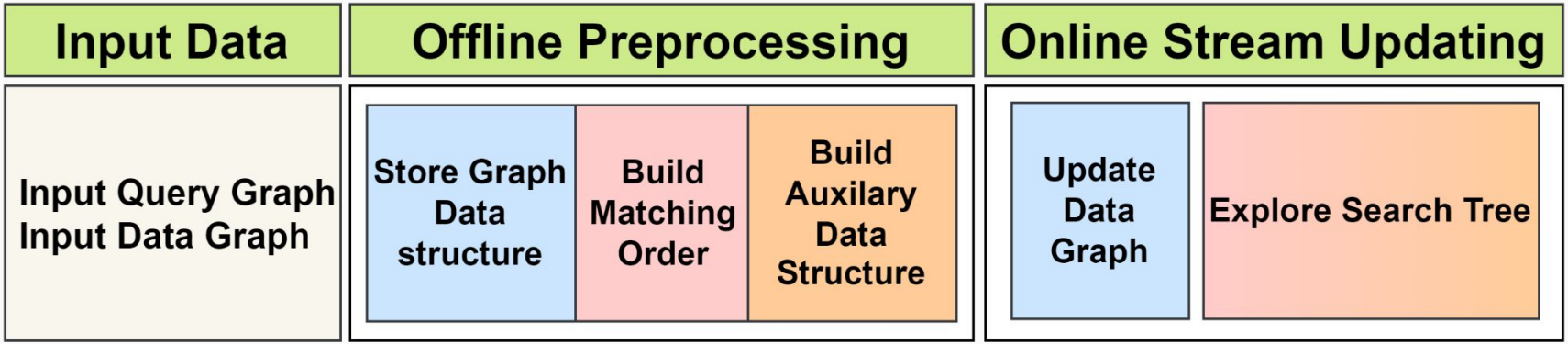
Table 1: Existing CSM solutions in recent research. Para: Parallelism. Srch: search method (✓ = backtrack, ✗ = join-based).

System	Para	index $\mathcal{A}$ update	Find Matches	Srch
		CPU Algorithms		
IncIsoMatch [9]	✗	Recomputation	N/A	✓
SJ-Tree [6]	✓	$O(\|E(G)\|^{\|E(Q)\|})$	$O(\|E(G)\|^{\|E(Q)\|})$	✗
Graphflow [15]	✓	O(1)	$O(d(G)^{\|V(Q)\|})$	✗
TurboFlux [16]	✗	$O(\|E(G)\|\|V(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
IEDyn [14]	✗	$O(\|E(G)\|\|V(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
Symbi [20]	✗	$O(\|E(G)\|\|E(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
RapidFlow [25]	✓[1]	$O(\|E(G)\|\|E(Q)\|)$	$O(d(G)^{\|V(Q)\|})$	✓
Mnemonic [1]	✓	O(1)	$O(d(G)^{\|V(Q)\|})$	✓
CaLiG [32]	✗	$O(\|E(G)\|\|E(Q)\|)$	$O(\|V(G)\|^{K})^{2}$	✓
NewSP [18]	✗	O(1)	$O(d(G)^{\|V(Q)\|})$	✓
		GPU Algorithms		
GAMMA [23]	✓	O(1)	$O(d(G)^{V(Q)-1}$ $d(Q)\log(V))$	✗
GCSM [30]	✓	N/A	N/A	✗

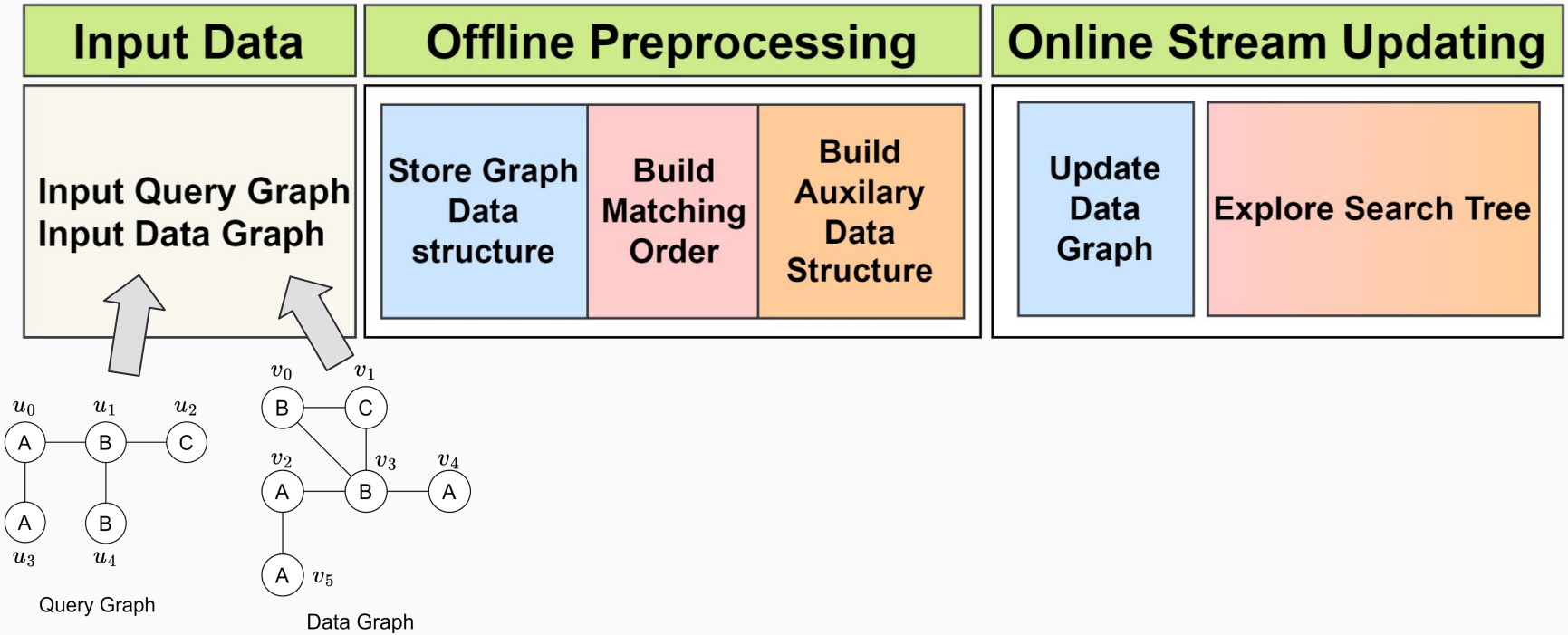
[1] RapidFlow has been parallelized in [30].

[2] K is the number of kernel vertices.

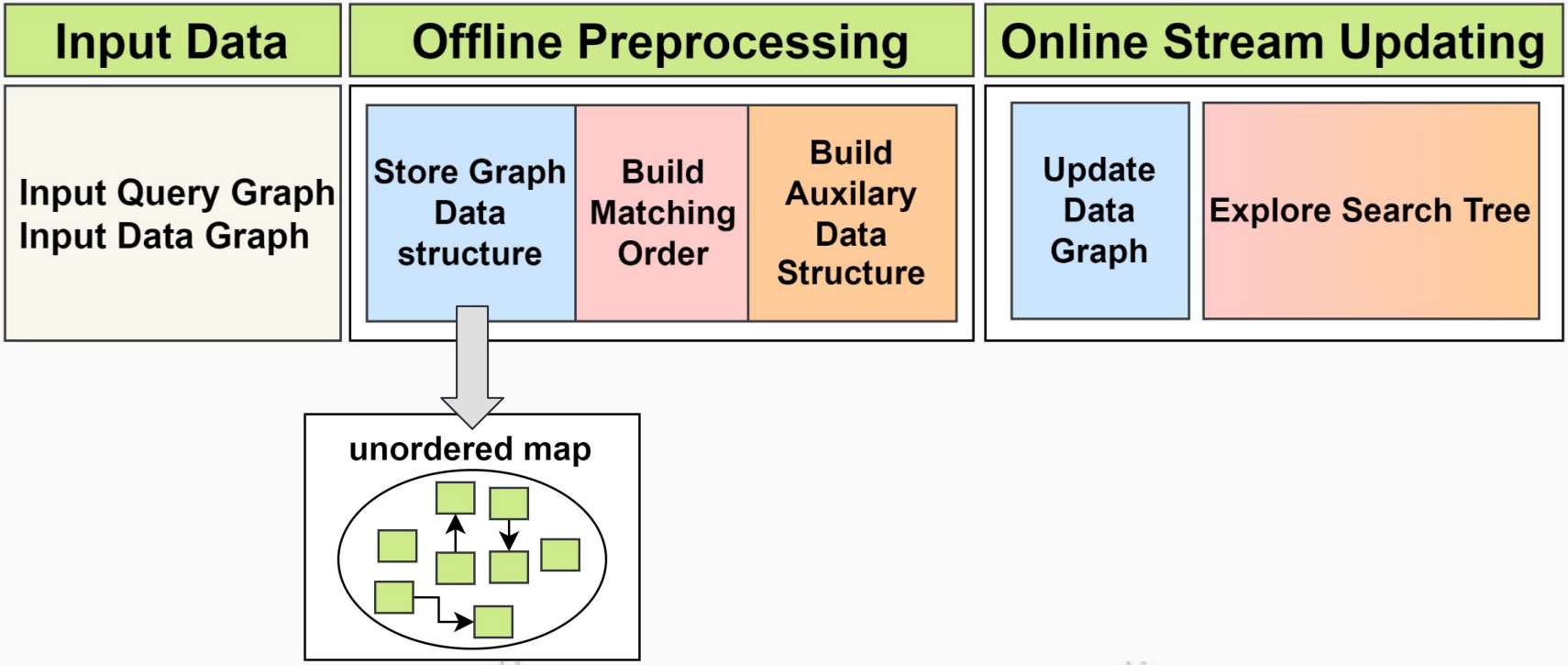
General Workflow of CSM Algorithms



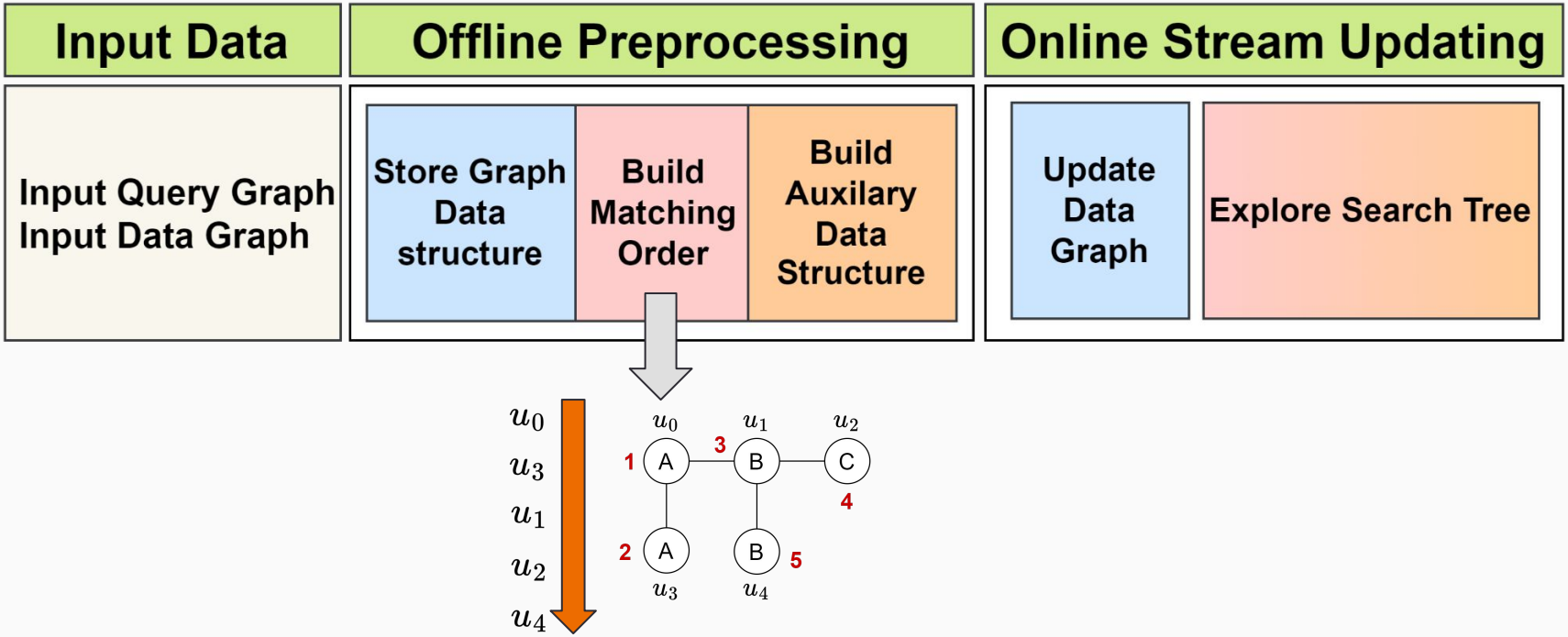
General Workflow of CSM Algorithms



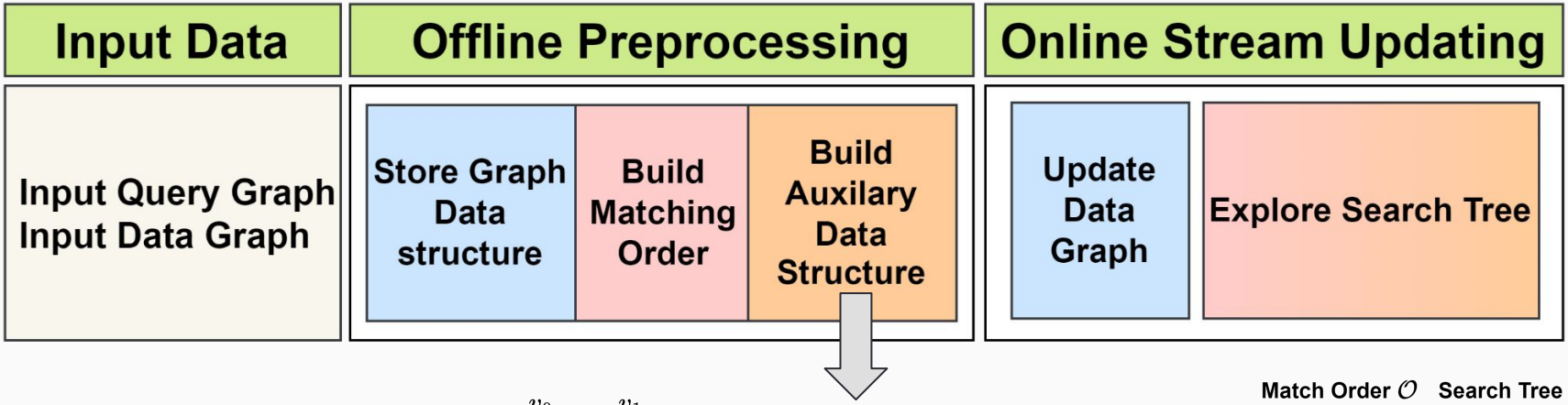
General Workflow of CSM Algorithms

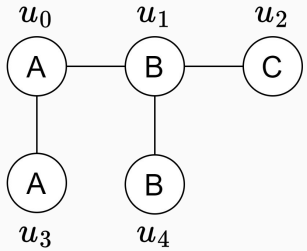


General Workflow of CSM Algorithms

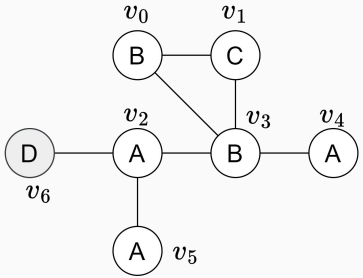


General Workflow of CSM Algorithms



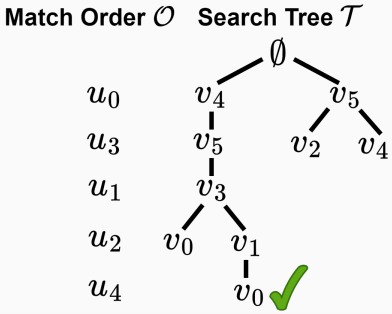


Query Graph

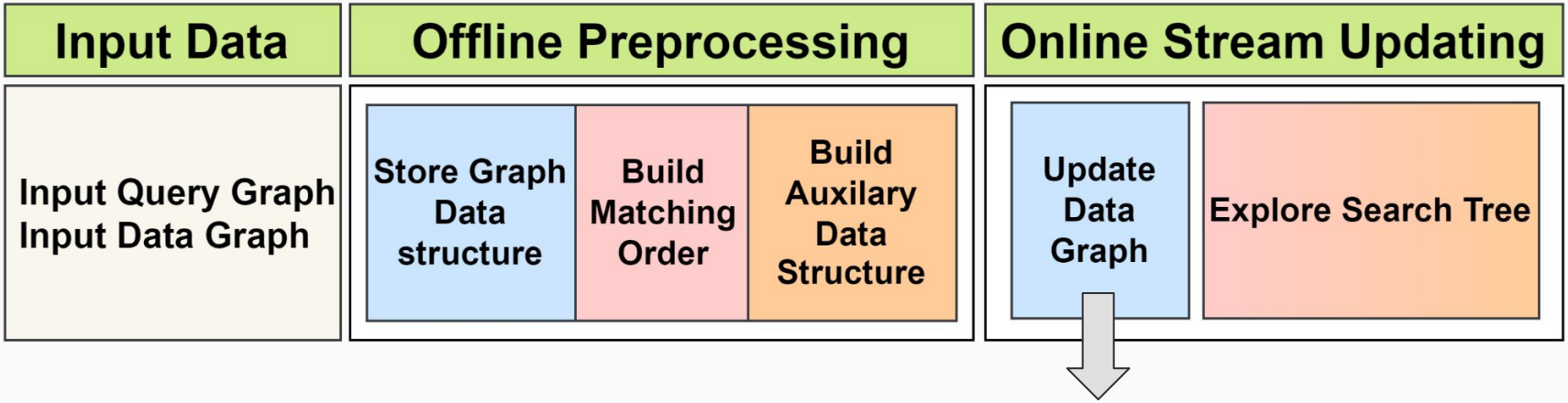


Data Graph

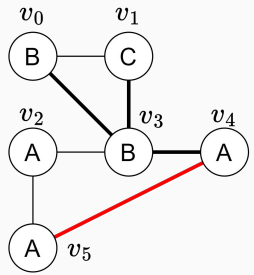
1. Prune the data Graph
2. Prune the search Tree



General Workflow of CSM Algorithms

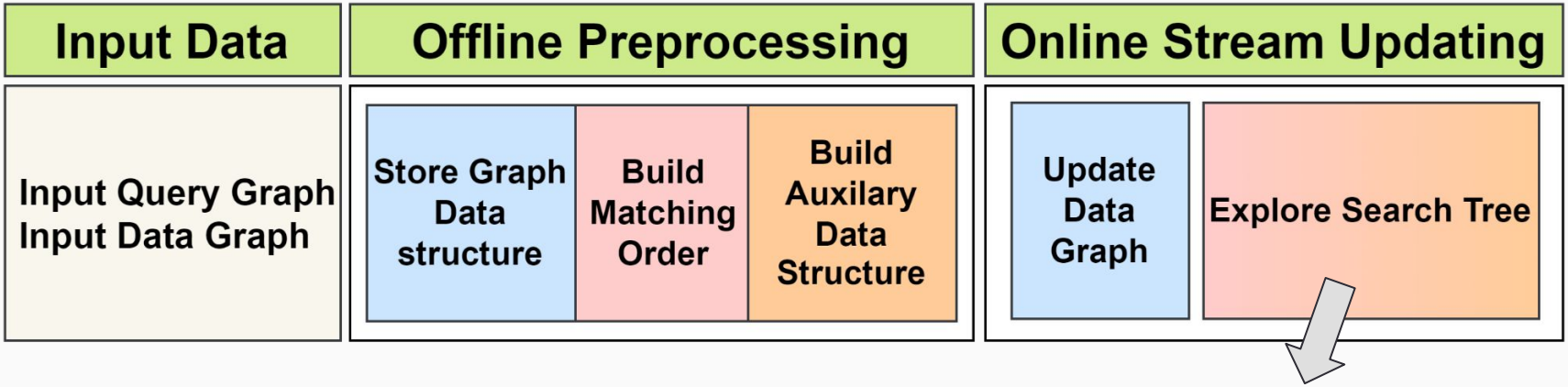


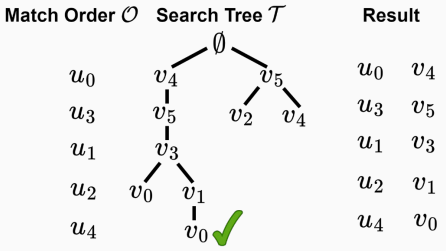
G': Insert $e(v_4, v_5)$



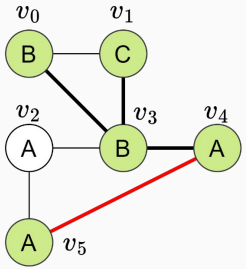
Update Datagraph & ADS

General Workflow of CSM Algorithms



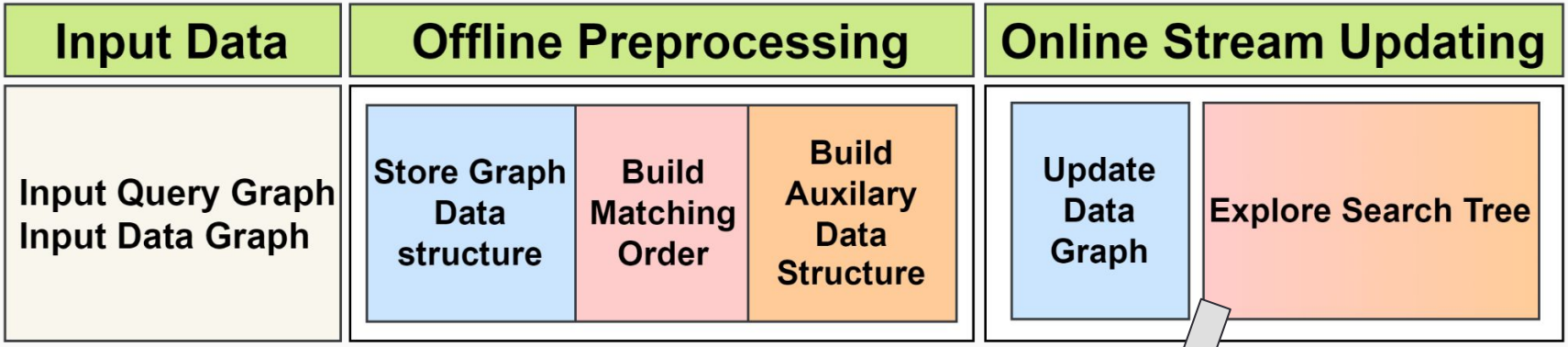


Explore Search Tree with matching order

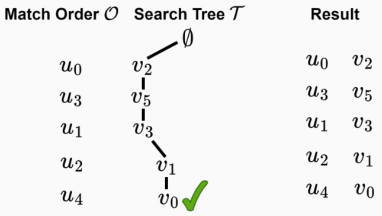


Find Incremental Increase Matches

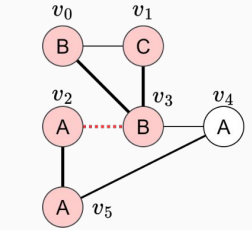
General Workflow of CSM Algorithms



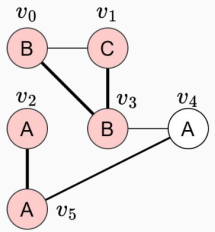
G'': Delete $e(v_2, v_3)$



Explore Search Tree with matching order

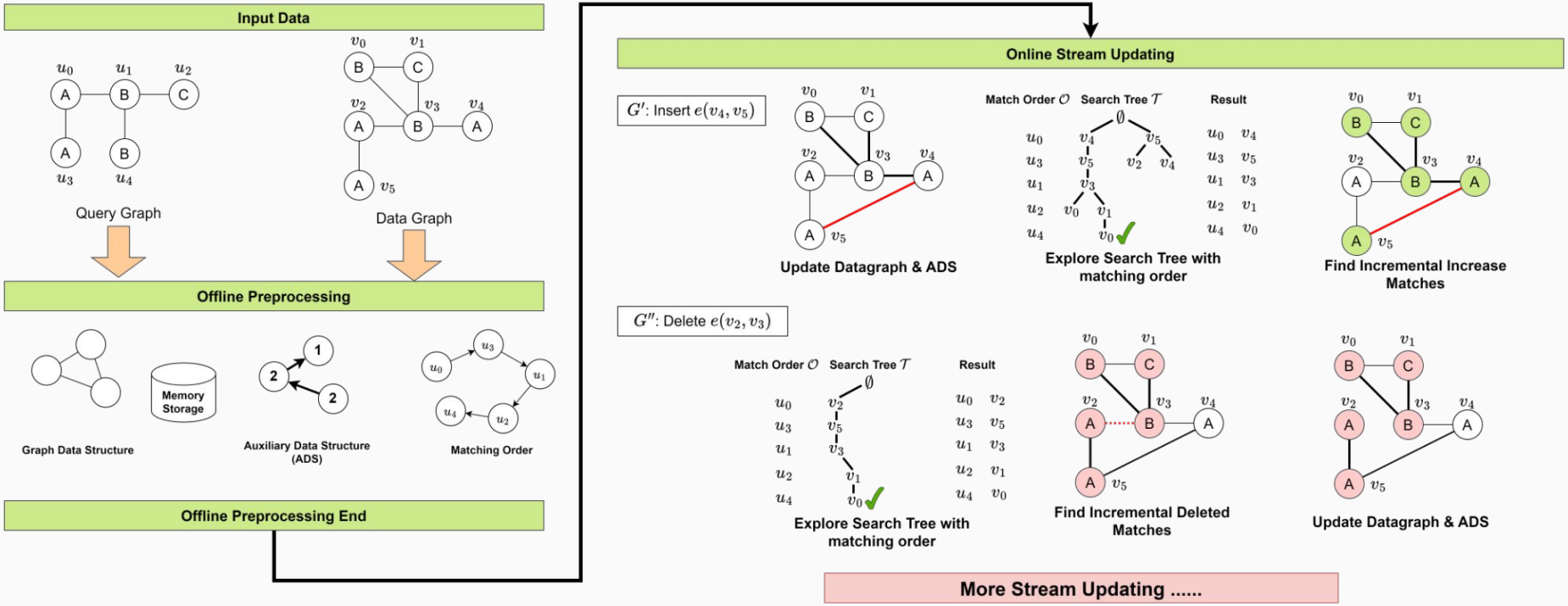


Find Incremental Deleted Matches



Update Datagraph & ADS

General Workflow

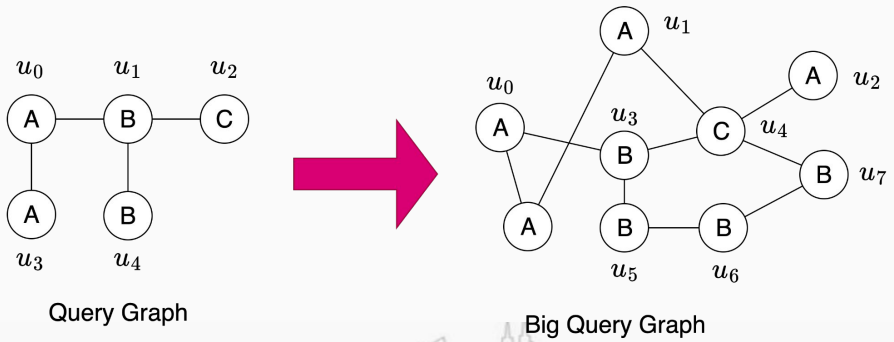


Big Query Graphs take much more time!

- **Real-world CSM Scenarios**

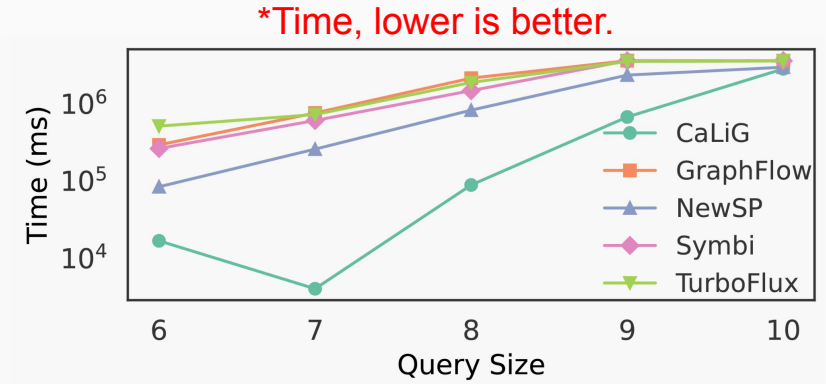
Complex data/query graphs
- millions of vertices, billions of edges

Update Propagation
- updates trigger 5-10 hops traverse



- **Scalability Limitations**

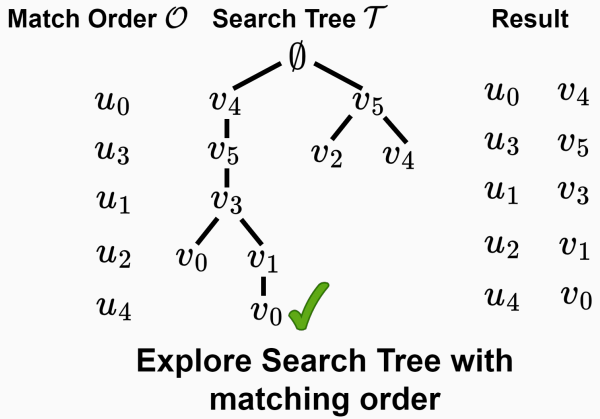
Computation time sharply increases with query size.



Challenges in Parallelizing CSM Algorithms

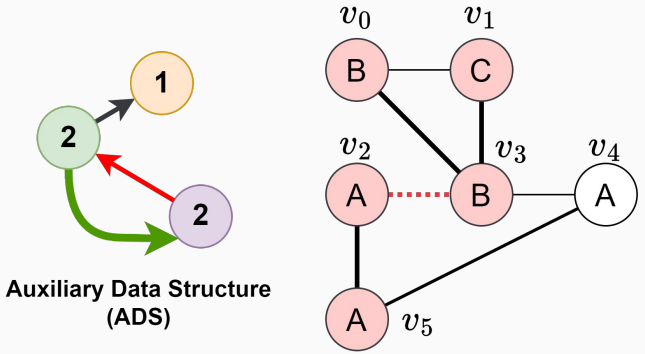
Real-world CSM Query Graph

Imbalance introduced by irregular workloads



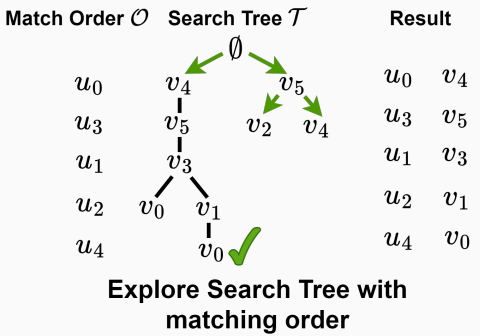
General Workflow of CSM

Bottleneck caused by sequential auxiliary data structure updates



Observation 1 : Find Matches Process Dominates

- **Insight**
  - Explore Search Tree dominates computation time (**>90%**)
  - Search tree well-defined for task decomposition
- **Innovation**
  - Parallelize Find Matches Process
  - Decompose search tree into subtasks
  - Dynamic thread-level task assignment (load balancing)

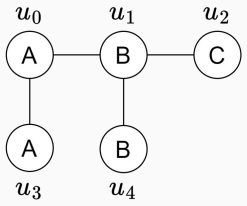


Algorithm	Query Size 6			Query Size 7			Query Size 8			Query Size 9			Query Size 10		
	ADS Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate	Aux Upd	Find Matches	Succ Rate
CaLiG	0.49	99.33	100	3.69	95.28	100	0.64	99.25	99	0.07	99.91	94	0.68	99.30	24
GraphFlow	N/A	N/A	92	N/A	N/A	83	N/A	N/A	68	N/A	N/A	0	N/A	N/A	0
NewSP	0.75	53.64	98	0.17	85.59	94	0.02	98.13	90	0.01	98.62	50	0.01	98.85	30
Symbi	2.57	80.78	93	1.16	91.71	85	0.15	98.94	84	0.13	99.36	1	0.09	99.22	1
TurboFlux	1.80	86.01	87	0.72	94.45	83	0.13	98.99	70	0.09	99.30	6	0.28	97.83	1

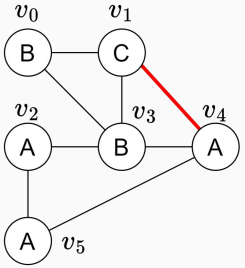
Observation 2 : High Safe Update Ratio

Safe Update Definition

A safe update is a graph modification that leaves the query match set unchanged.



Query Graph



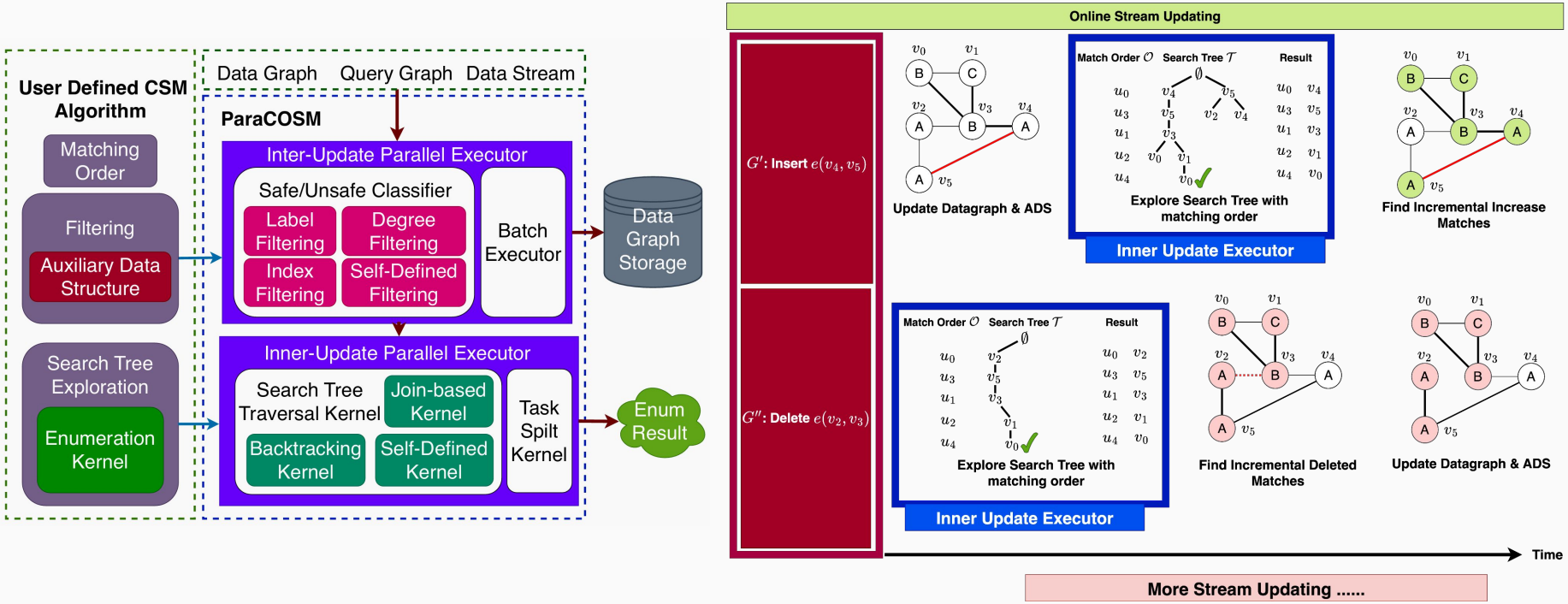
G': Insert $e(v_1, v_4)$

No New Matches!

Dataset	Query Size				
	6	7	8	9	10
LSBench	1.0794	1.5688	1.1622	0.4257	0.3192
LiveJournal	0.3028	0.3618	0.3848	0.3266	0.3057
Orkut	0.0010	0.0010	0.0010	0.0012	0.0011
Amazon	0.5356	0.6759	0.5249	0.5254	0.5724

- **Insight**
  - **> 98%** updates do not affect match results
  - Safe/unsafe updates distinguishable during runtime
- **Innovation**
  - Dynamically classify safe/unsafe updates in runtime
  - Apply safe updates in batch, unsafe in sequence

Design: ParaCOSM Architecture



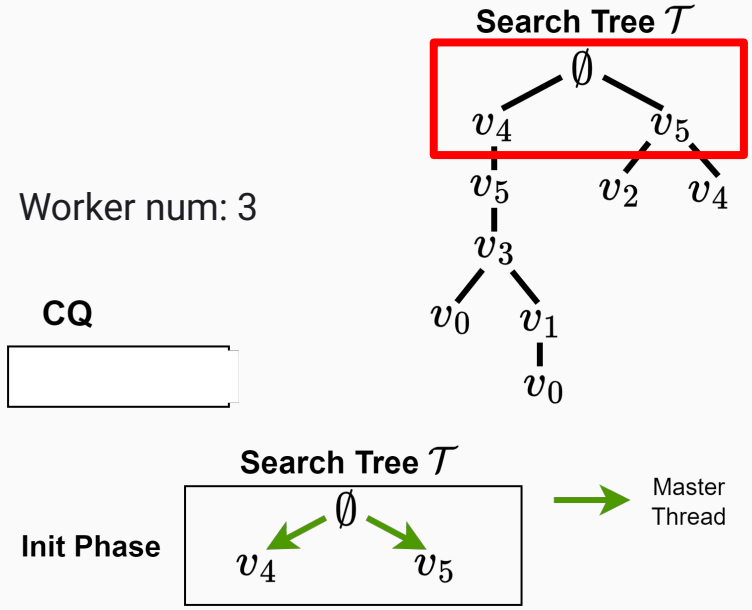
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



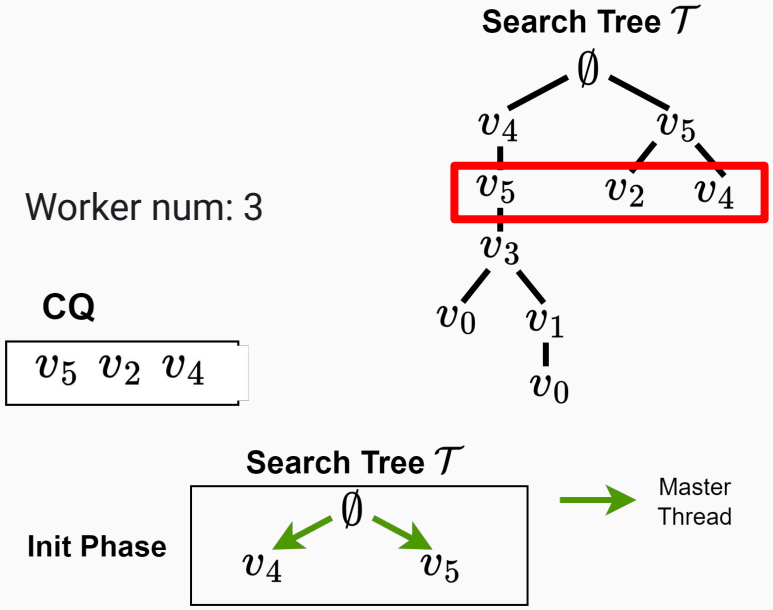
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



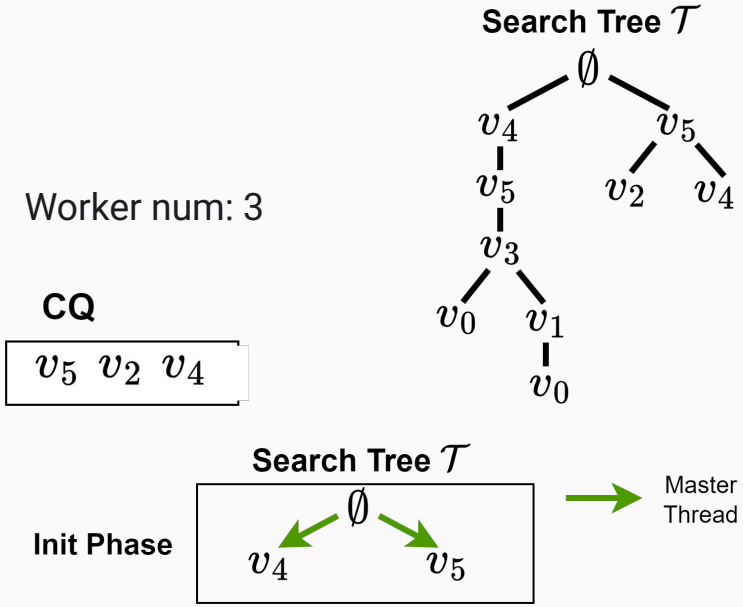
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



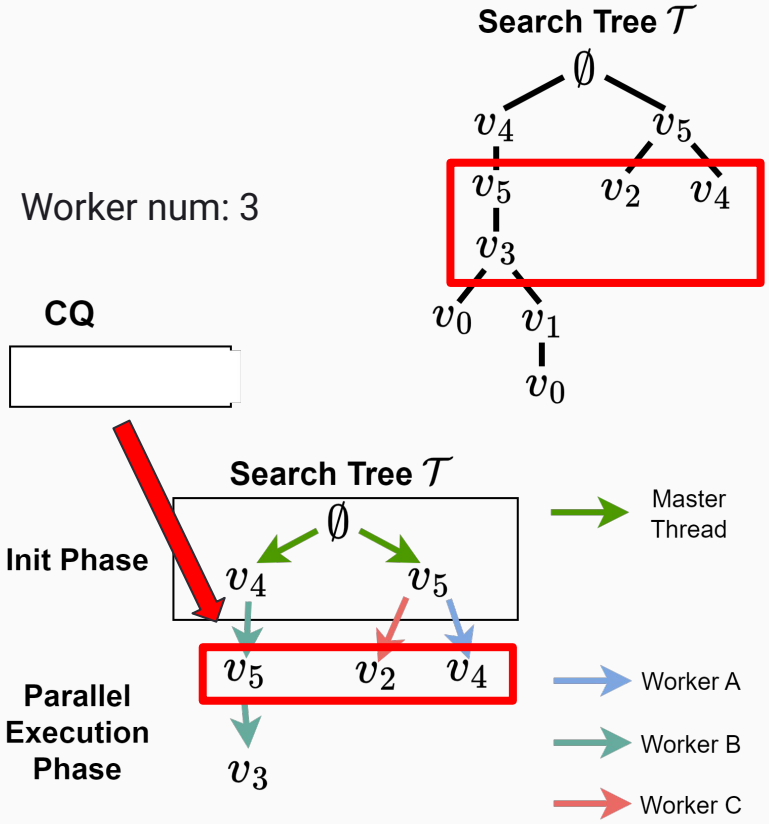
Design: Inner-update

- **Initialization Phase**

  1. BFS on search tree
  2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
  3. Stop when subtasks are enough

- **Parallel Execution Phase**

  1. Workers fetch subtasks from CQ
  2. Search with the original CSM logic
  3. Offload subtasks to CQ adaptively



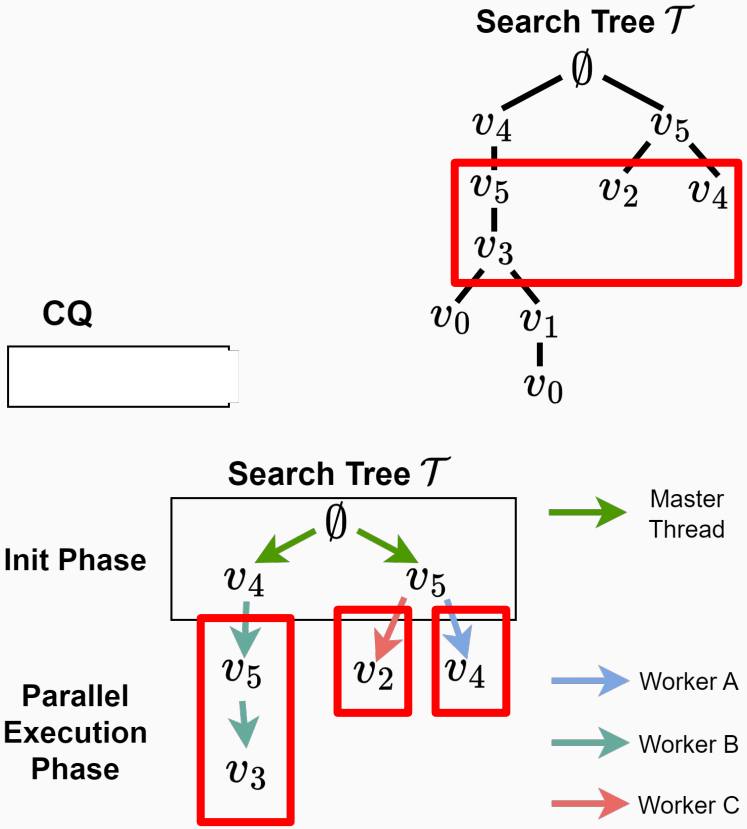
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



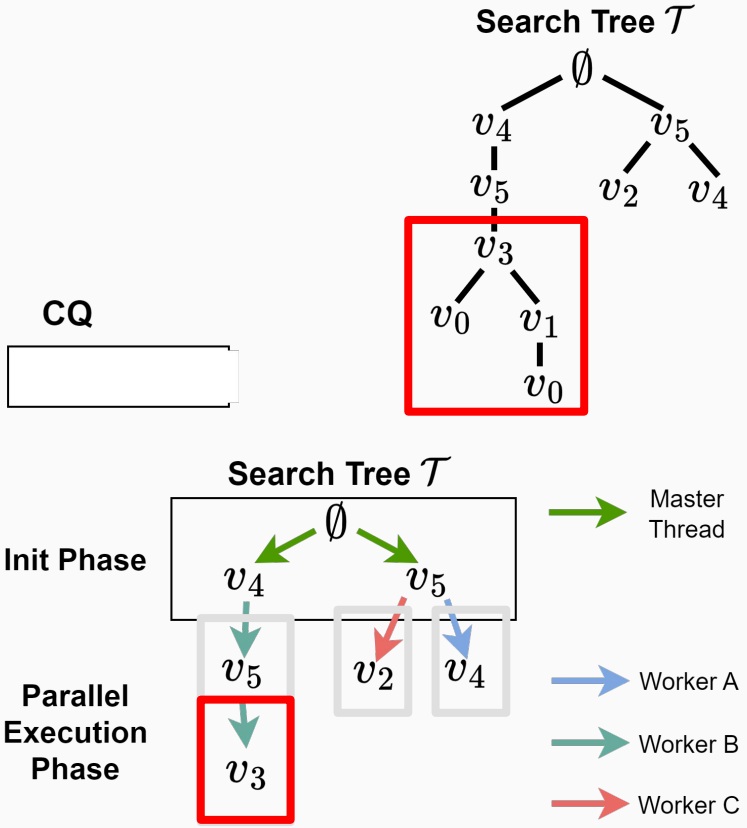
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



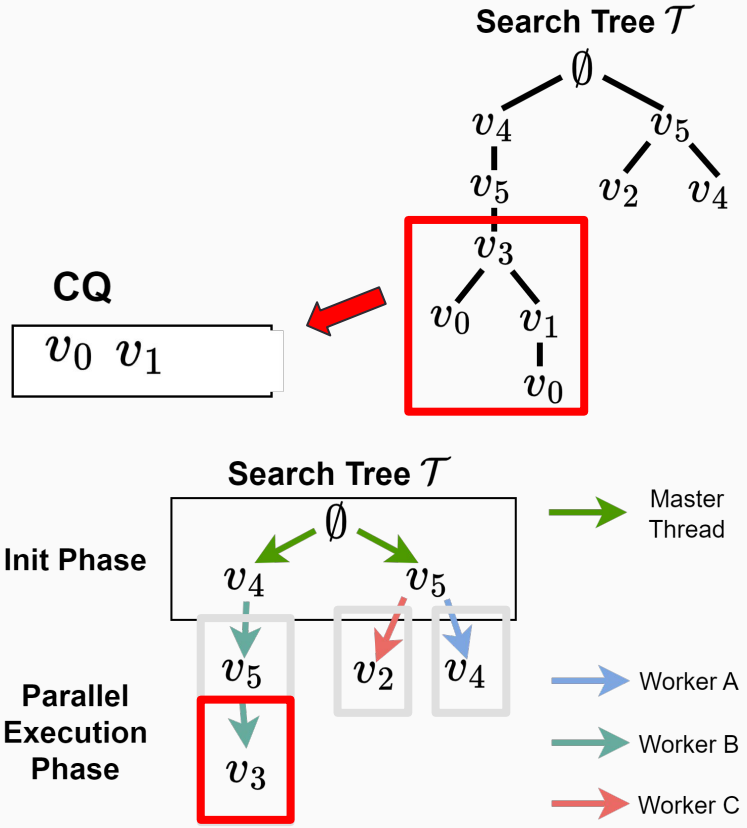
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



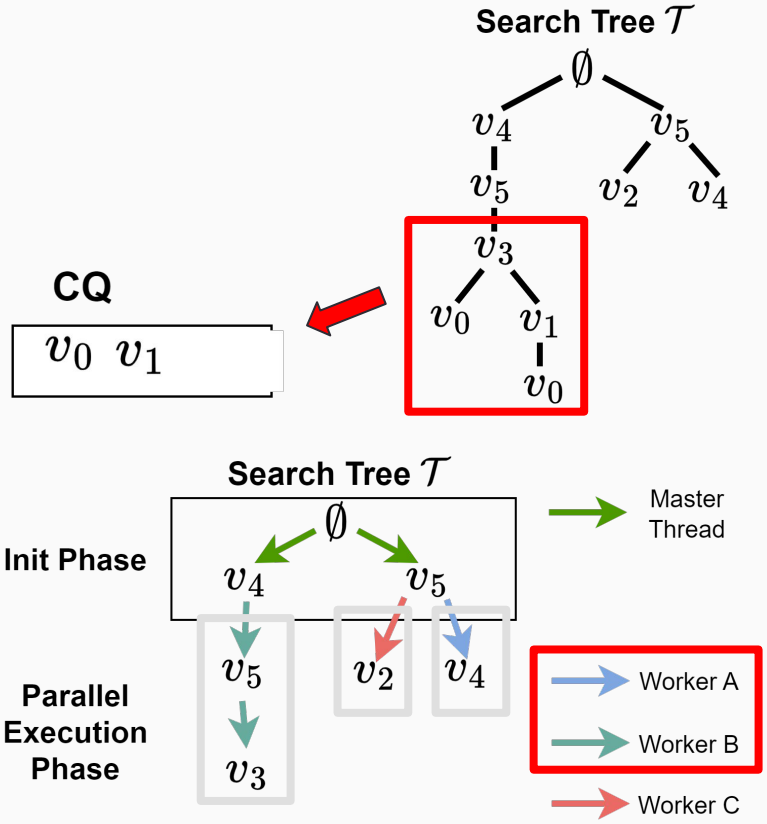
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



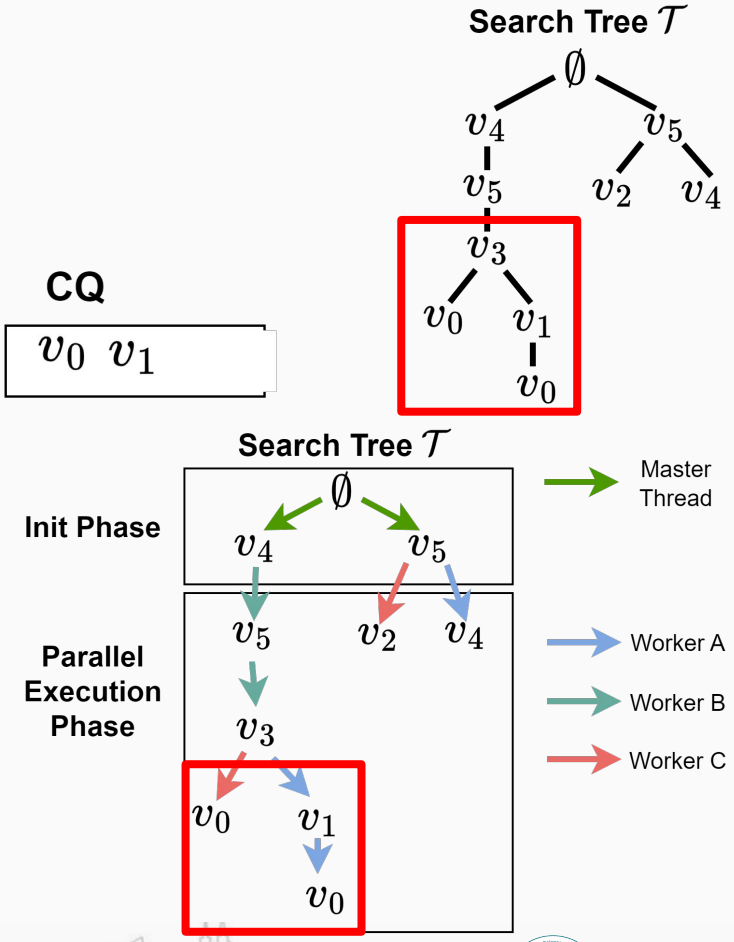
Design: Inner-update

- **Initialization Phase**

  1. BFS on search tree
  2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
  3. Stop when subtasks are enough

- **Parallel Execution Phase**

  1. Workers fetch subtasks from CQ
  2. Search with the original CSM logic
  3. Offload subtasks to CQ adaptively



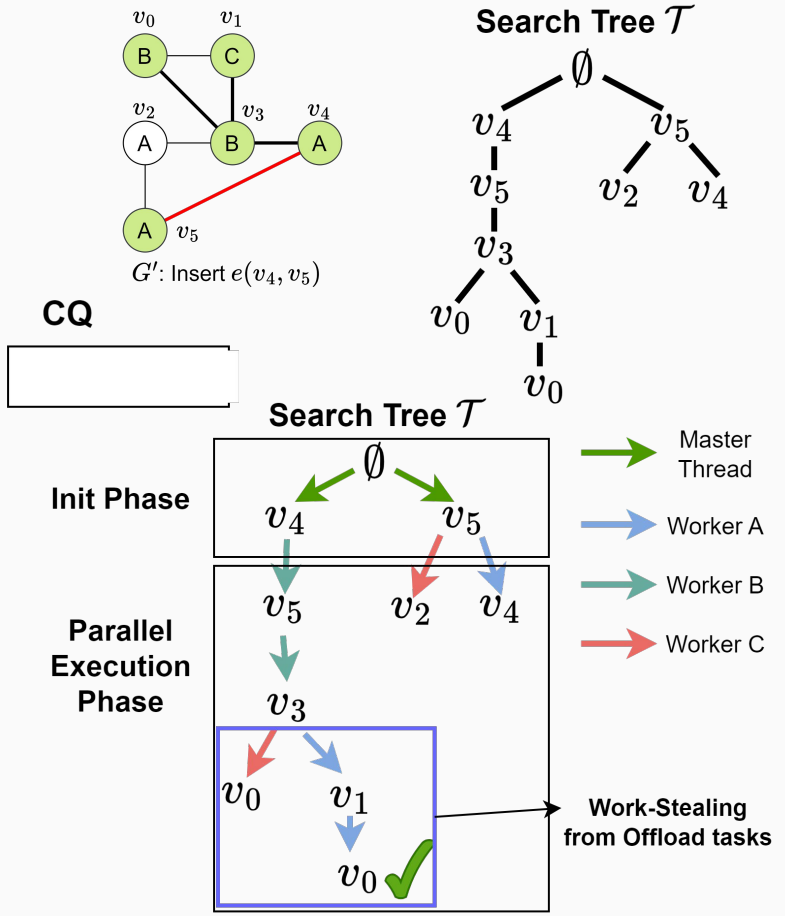
Design: Inner-update

- **Initialization Phase**

1. BFS on search tree
2. Enqueue subtrees as subtasks into a concurrent task queue (CQ)
3. Stop when subtasks are enough

- **Parallel Execution Phase**

1. Workers fetch subtasks from CQ
2. Search with the original CSM logic
3. Offload subtasks to CQ adaptively



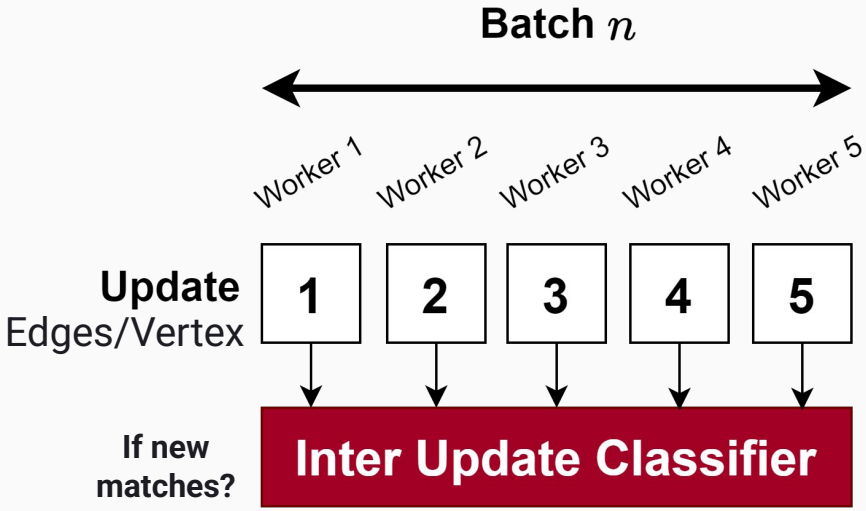
Design: Inter-update

- **Update Type Classifier**

1. Label filtering
2. Degree filtering
3. Candidate filtering

- **Batch Executor**

1. Parallel classification
2. Detect unsafe updates
3. Batch coordination



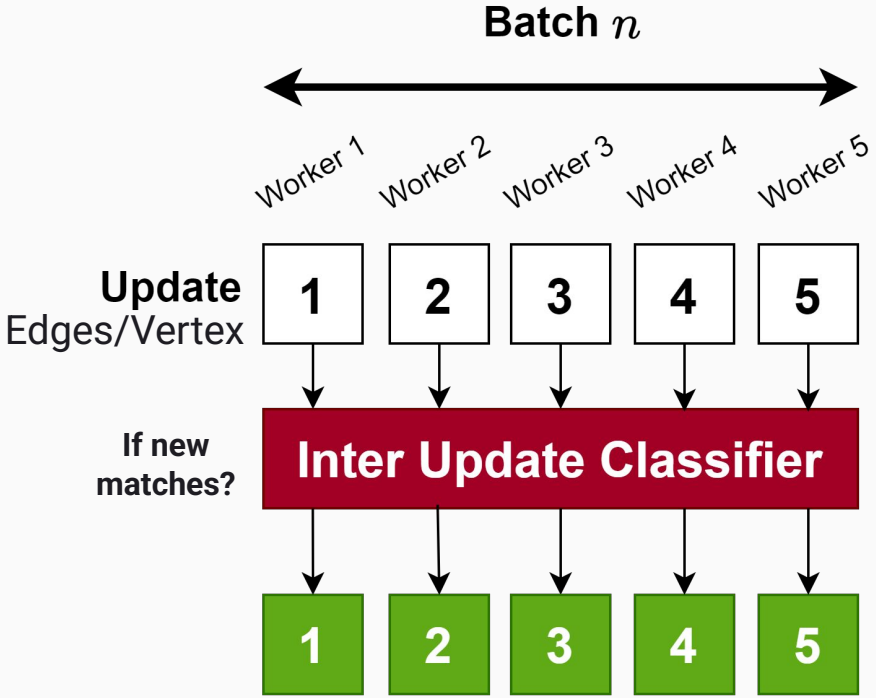
Design: Inter-update

- **Update Type Classifier**

1. Label filtering
2. Degree filtering
3. Candidate filtering

- **Batch Executor**

1. Parallel classification
2. Detect unsafe updates
3. Batch coordination



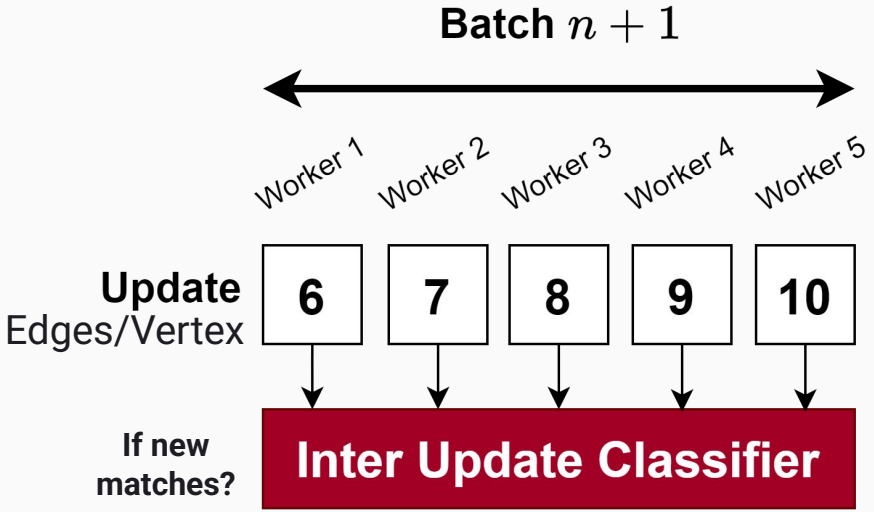
Design: Inter-update

- **Update Type Classifier**

  1. Label filtering
  2. Degree filtering
  3. Candidate filtering

- **Batch Executor**

  1. Parallel classification
  2. Detect unsafe updates
  3. Batch coordination



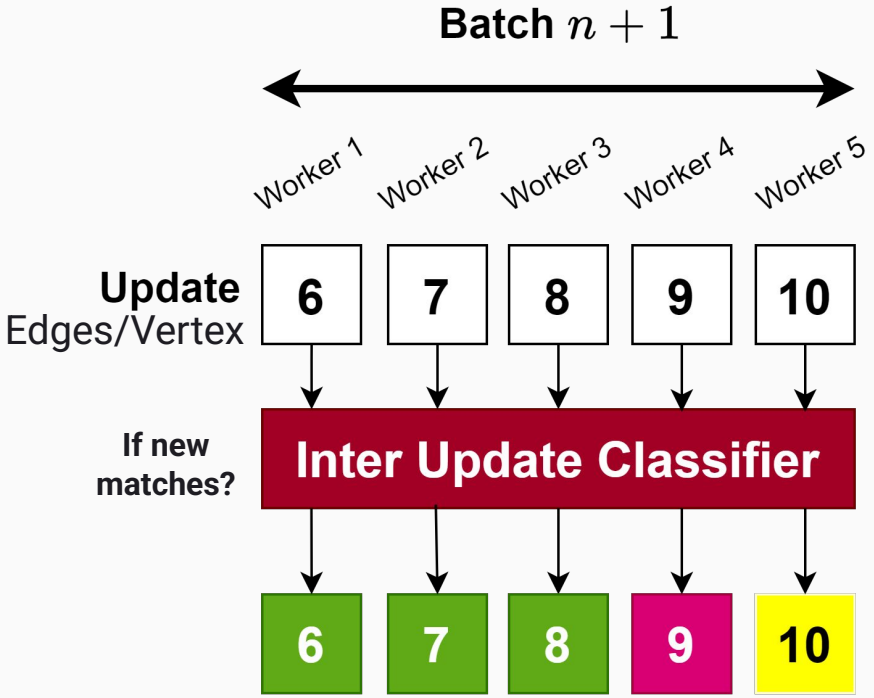
Design: Inter-update

- **Update Type Classifier**

1. Label filtering
2. Degree filtering
3. Candidate filtering

- **Batch Executor**

1. Parallel classification
2. Detect unsafe updates
3. Batch coordination



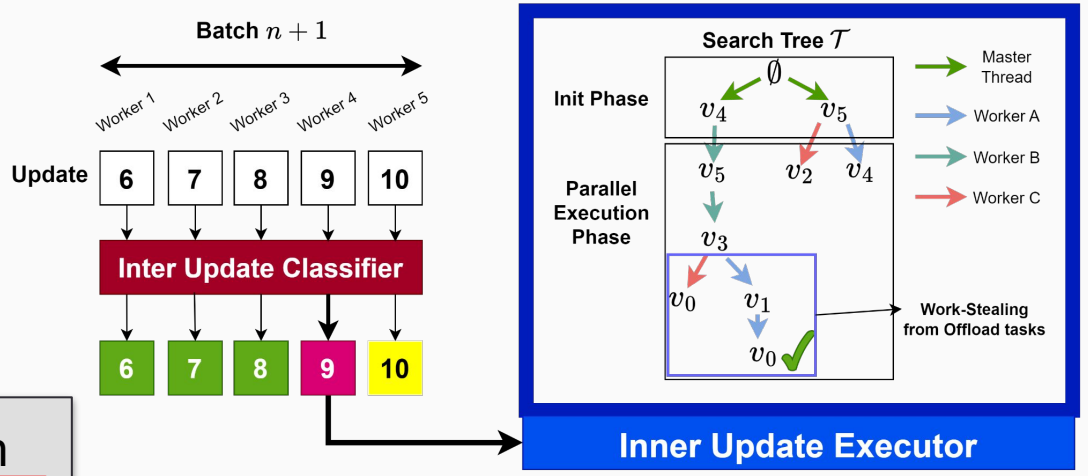
Design: Inter-update

- **Update Type Classifier**

1. Label filtering
2. Degree filtering
3. Candidate filtering

- **Batch Executor**

1. Parallel classification
2. Detect unsafe updates
3. Batch coordination



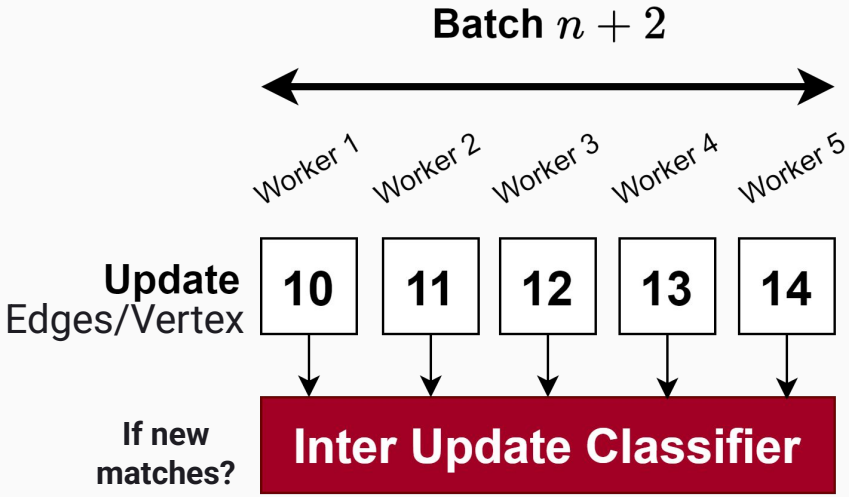
Design: Inter-update

- **Update Type Classifier**

  1. Label filtering
  2. Degree filtering
  3. Candidate filtering

- **Batch Executor**

  1. Parallel classification
  2. Detect unsafe updates
  3. Batch coordination



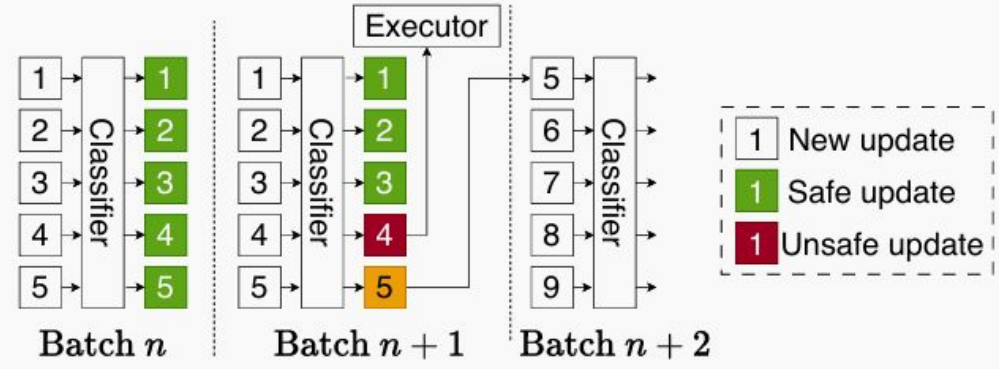
Design: Inter-update

- **Update Type Classifier**

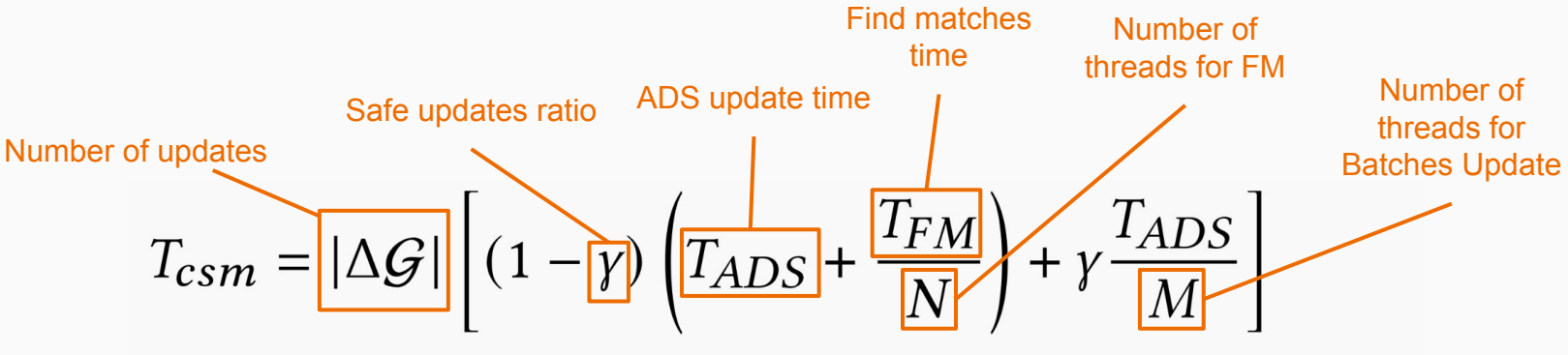
1. Label filtering
2. Degree filtering
3. Candidate filtering

- **Batch Executor**

1. Parallel classification
2. Detect unsafe updates
3. Batch coordination



Speedup Analysis



The framework gains more from inter-update parallelism as γ rises.

Safe Update Ratio Analysis

The probability of an inserted edge matches a specific edge in E(Q)

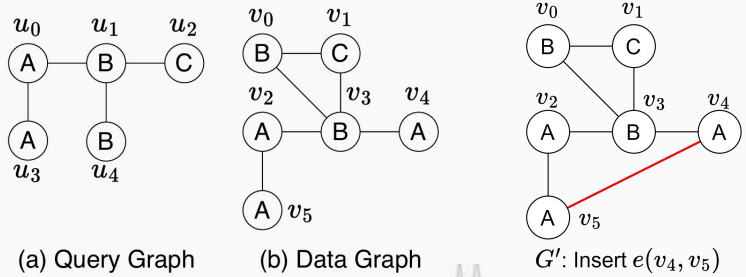
$$P(\text{match}) = \frac{1}{|L(E)|} \times \frac{1}{|L(V)|} \times \frac{1}{|L(V)|} = \frac{1}{|L(E)||L(V)|^2}$$

The probability of an unsafe update

$$P(\text{unsafe}) = |E(Q)| \times \frac{1}{|L(E)||L(V)|^2} = \frac{|E(Q)|}{|L(E)||L(V)|^2}$$

Calculate using the metadata of **LiveJournal**

$$P(\text{unsafe}) = \frac{6}{30^2 \cdot 1} = 0.677\%, \quad P(\text{safe}) = 99.33\%.$$



(a) Query Graph (b) Data Graph G': Insert $e(v_4, v_5)$

Most of updates are safe!

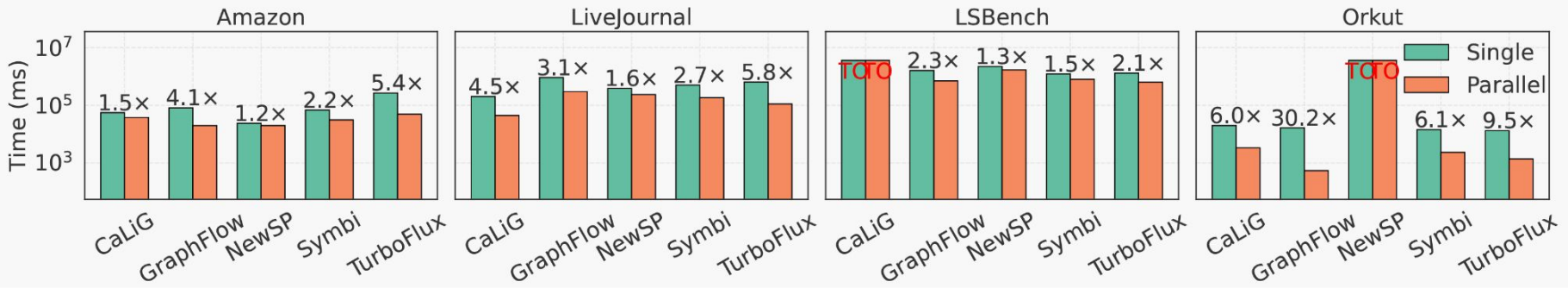
Experiments: Setup

- **Testbed**
  - Intel Xeon Platinum 8380 CPU (80 physical cores / 160 threads), 250 GB of memory, Ubuntu 22.04.
- **Query Generation**
  - Extract subgraphs from the data graph randomly.
- **Metrics**
  - **Time**: incremental matching time.
  - **Success rate**: the percentage of queries completed within one hour.

Dataset	$\|V\|$	$\|E\|$	$\|L(V)\|$	$\|L(E)\|$	$d(G) = \frac{2\|E\|}{\|V\|}$
Amazon	403,394	2,433,408	6	1	12.06
LiveJournal	4,847,571	42,841,237	30	1	17.68
LSBench	5,210,099	20,270,676	1	44	7.78
Orkut	3,072,441	117,185,083	20	20	20

Experiments: Overall Comparison

*Time, lower is better.



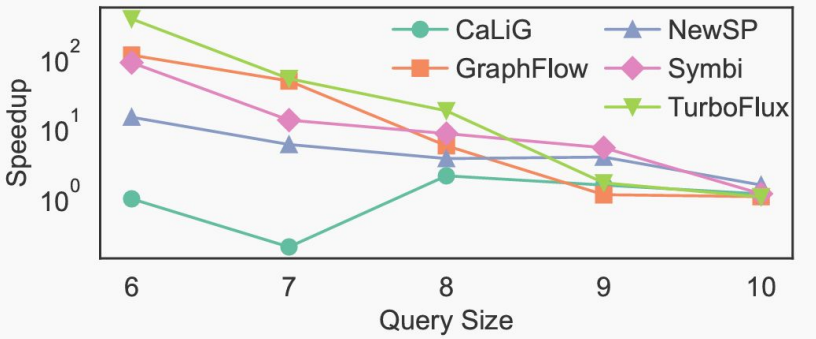
The performance of CSM algorithms varies significantly across datasets.

Significant speedups across all algorithms and datasets, with GraphFlow (up to 30.2×) and TurboFlux (up to 9.5×)

1.2× to 30.2× speedup across datasets

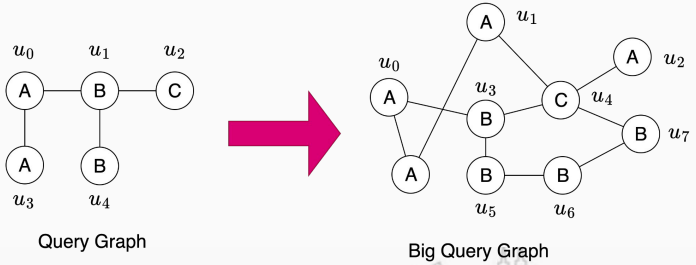
Count searching time > 3600s as timeouts (TO)

Experiments: Large Query Graph Acceleration



Alg. (Parallel)	Query Size				
	6	7	8	9	10
CaLiG	100 (+0)	99 (-1)	96 (-3)	84 (-10)	40 (+16)
GraphFlow	100 (+8)	97 (+14)	92 (+24)	11 (+11)	0 (+0)
NewSP	99 (+1)	95 (+1)	98 (+8)	55 (+5)	45 (+15)
Symbi	100 (+7)	99 (+14)	92 (+6)	72 (+71)	13 (+12)
TurboFlux	100 (+13)	98 (+15)	94 (+24)	32 (+26)	0 (-1)

On large query graphs ParaCOSM maintain effective gains for large queries.

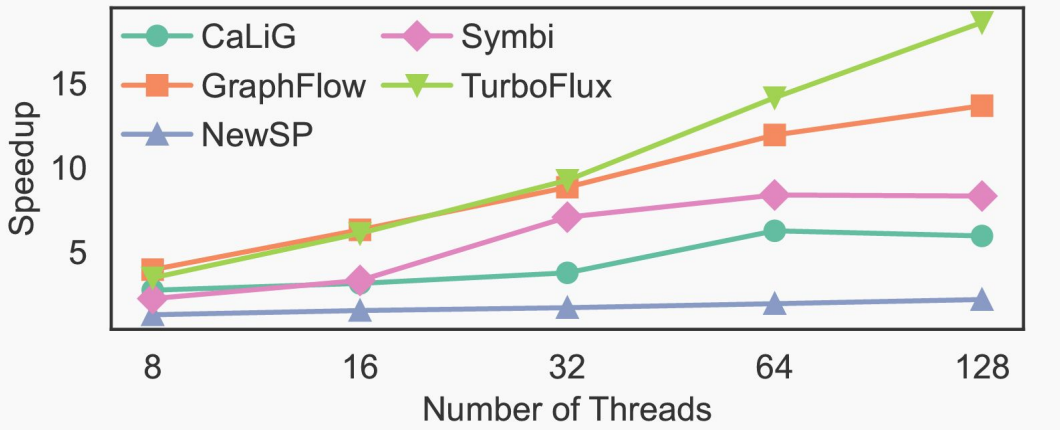


Up to two orders of magnitude faster execution

Up to 71% higher success rates on large query graphs

Experiments: Scalability

*Speedup, higher is better.

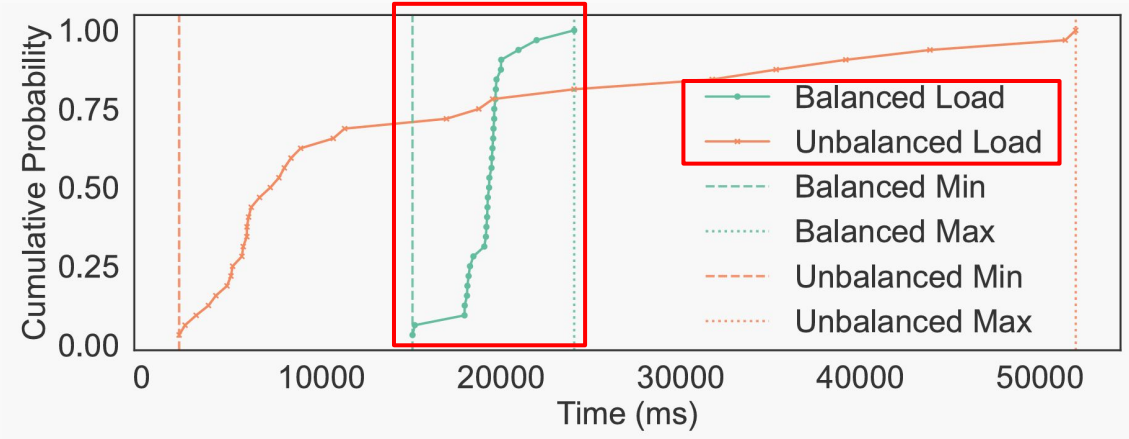


1. TurboFlux scales best (3.4×→18.6×).
2. GraphFlow scales steadily (3.9×→13.7×).
3. Symbi (7.0×) and CaLiG (3.2×) peak at 32 threads.
4. NewSP shows limited scalability (up to 2.1× at 128 threads).

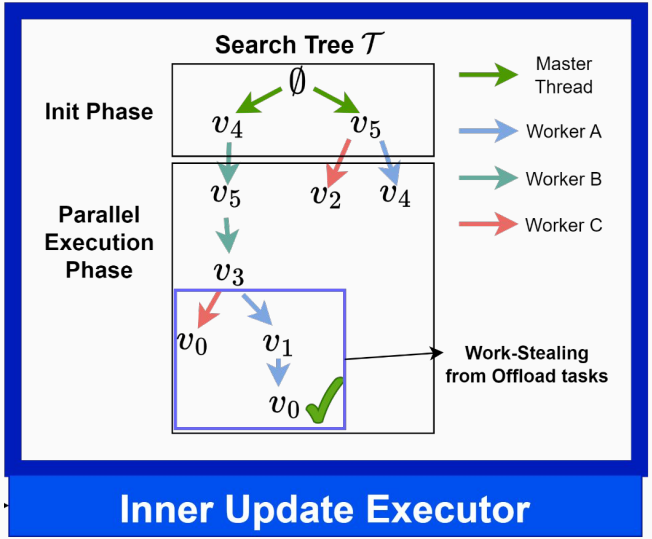
Count searching time > 3600s as timeouts (TO)

Experiments: Load balance

*Load balancing. More close to each other is better.

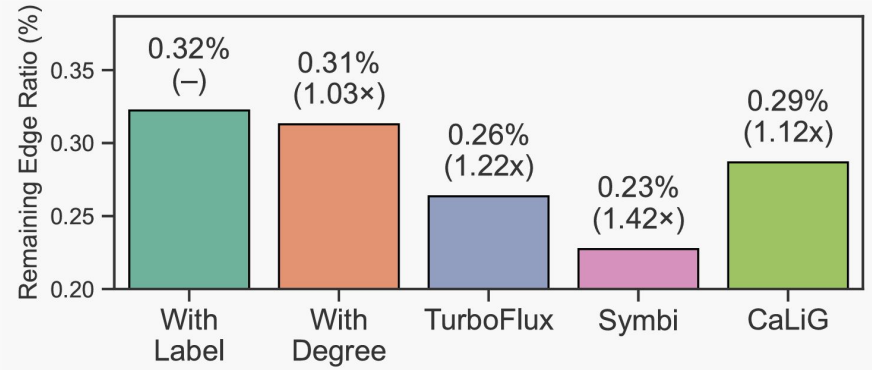


Each point denotes a thread's execution time

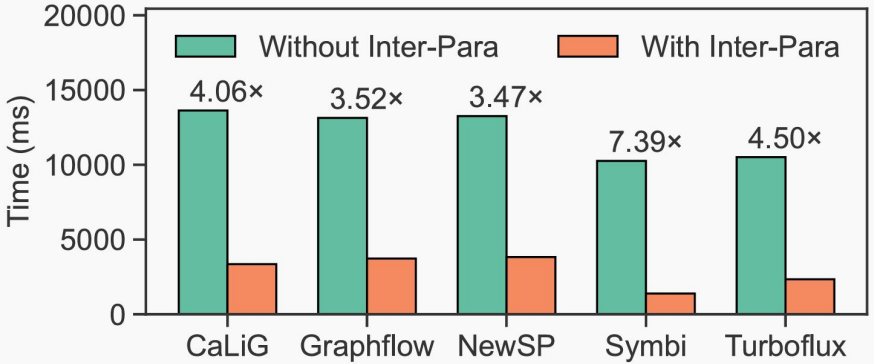


Experiments: Classifier

*Remaining edges, lower is better.

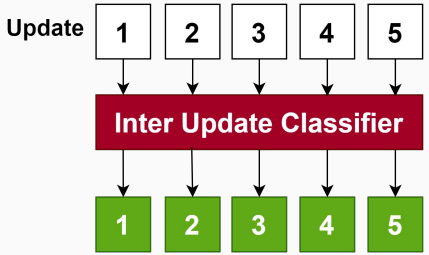


*Time, lower is better.



- **Update Type Classifier**

1. Label filtering
2. Degree filtering
3. ADS filtering



Conclusion

ParaCOSM

ParaCOSM Framework

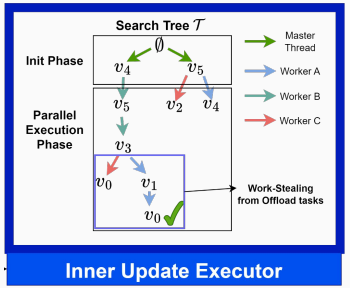
Advanced Parallel framework for CSM

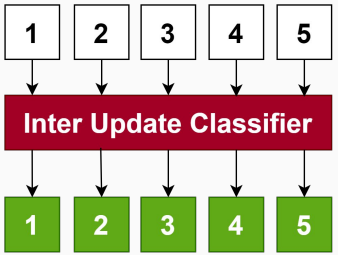
Inner-update Parallelism

Breaks dynamic search tree → fine-grained parallelism

Inter-update Parallelism

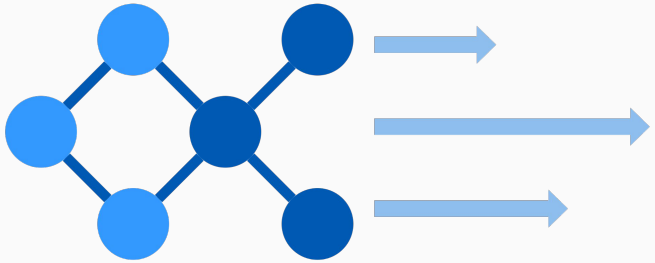
Exploits safe updates → concurrent updates





- **1.2× to 30.2× speedup across datasets**
- **Up to two orders of magnitude faster execution**
- **Up to 71% higher success rates on large query graphs**

Thanks for watching !



https://github.com/SUSTech-HPCLab/ParaCOSM.git

Contact:

12211612@mail.sustech.edu.cn