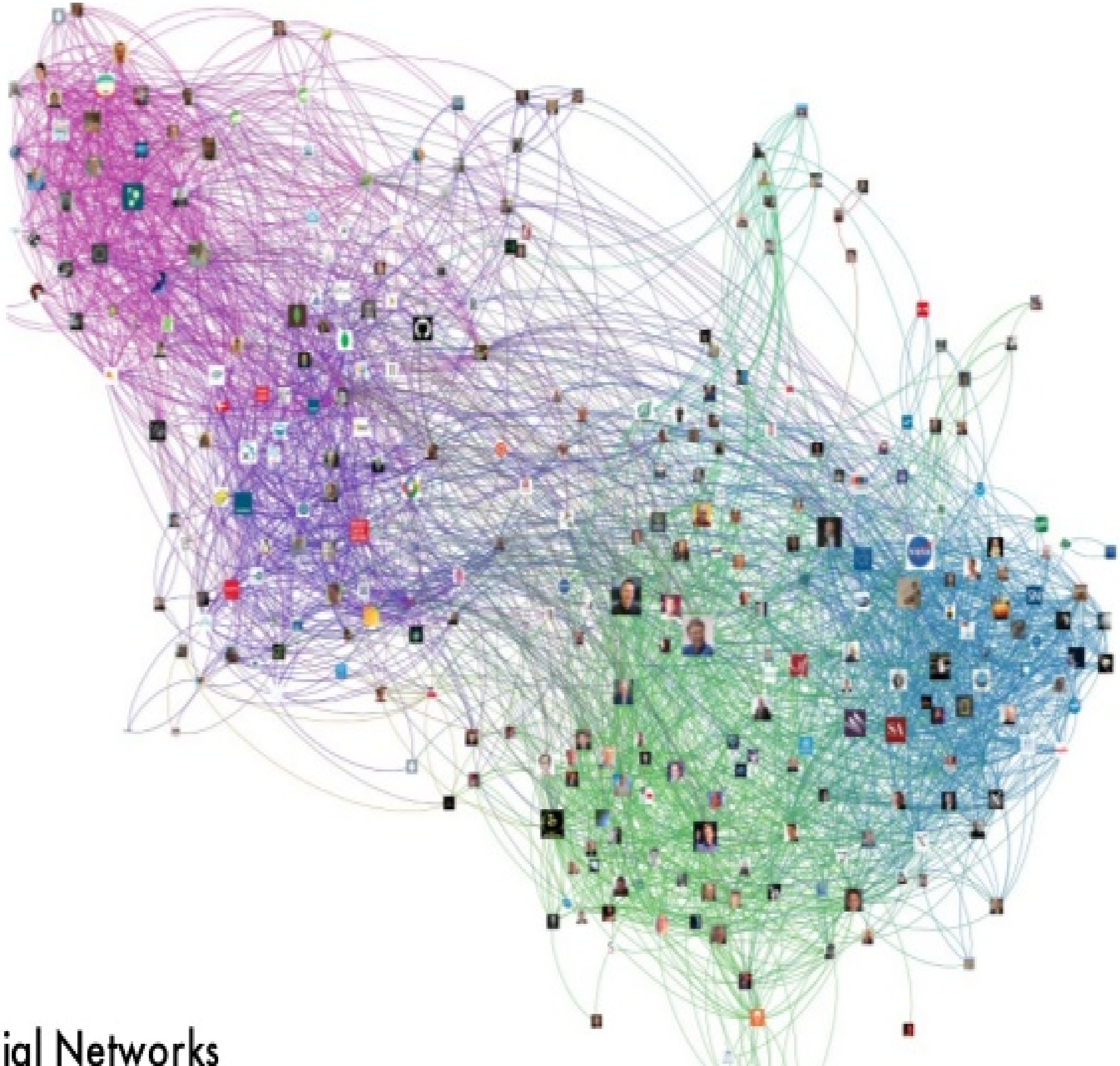
Tegra: Time-evolving Graph Processing on Commodity Clusters

Anand Iyer, Ion Stoica

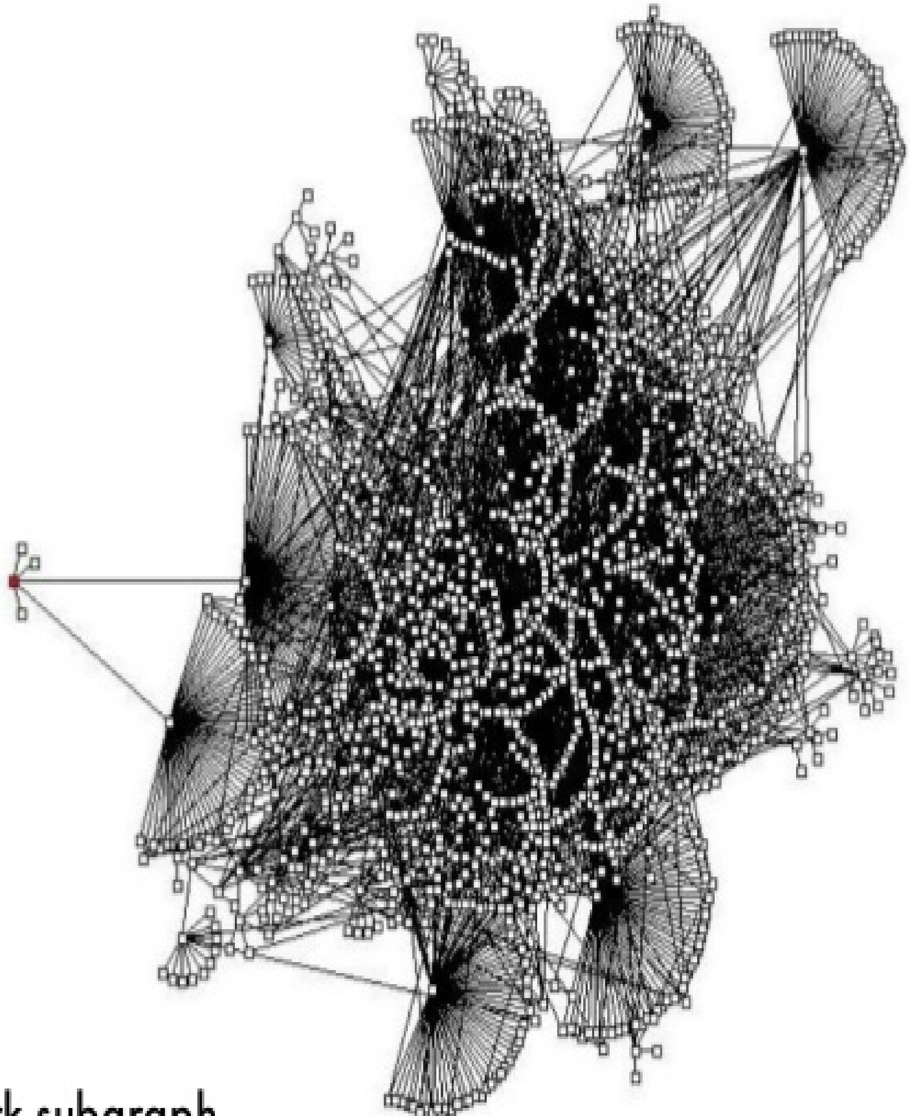
Presented by Ankur Dave

Graphs are everywhere...



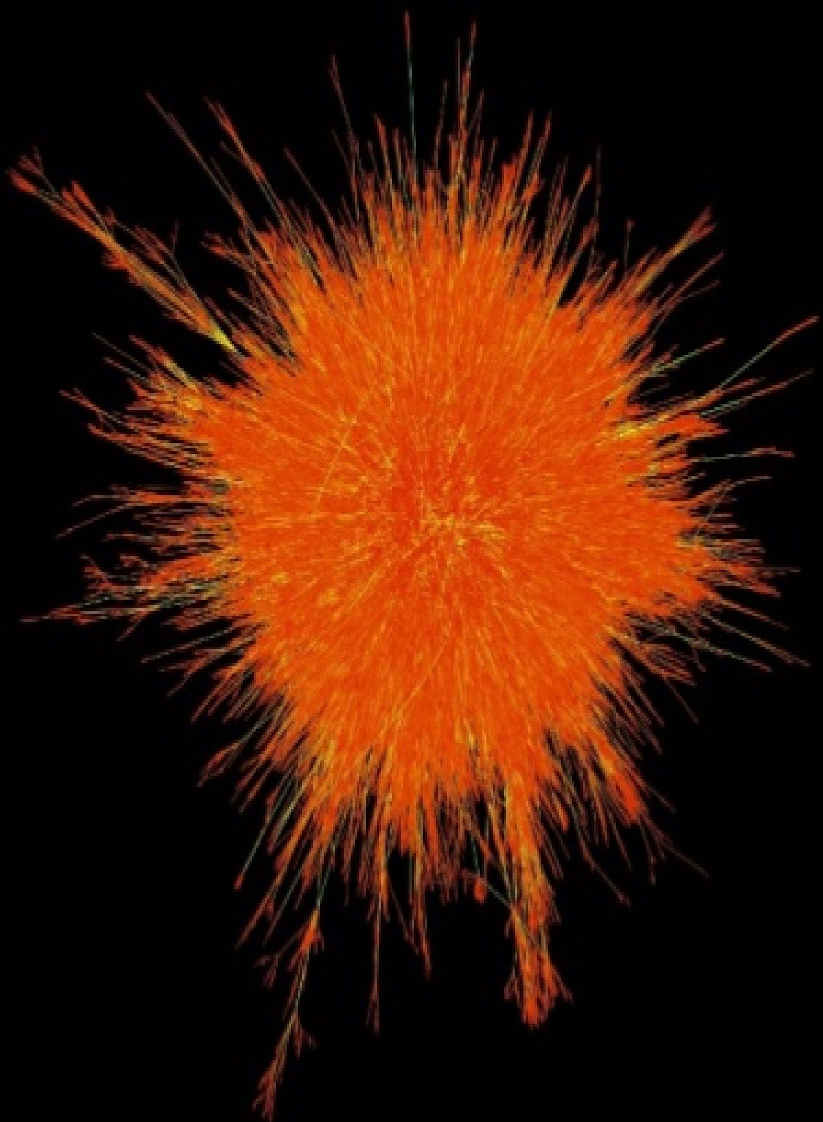
Social Networks

Graphs are everywhere...

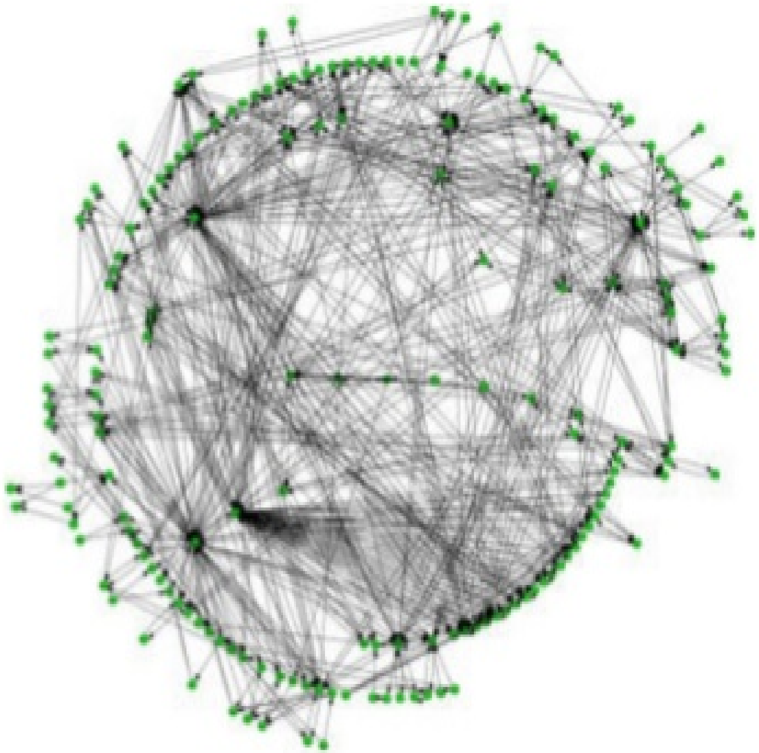


Gnutella network subgraph

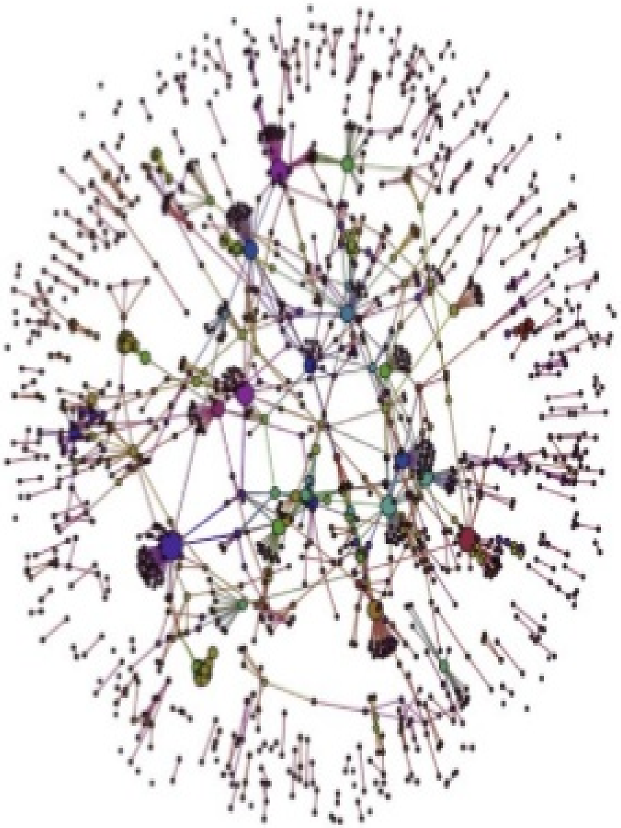
Graphs are everywhere...



Graphs are everywhere...



Metabolic network of a single cell organism



Tuberculosis

Plenty of interest in processing them

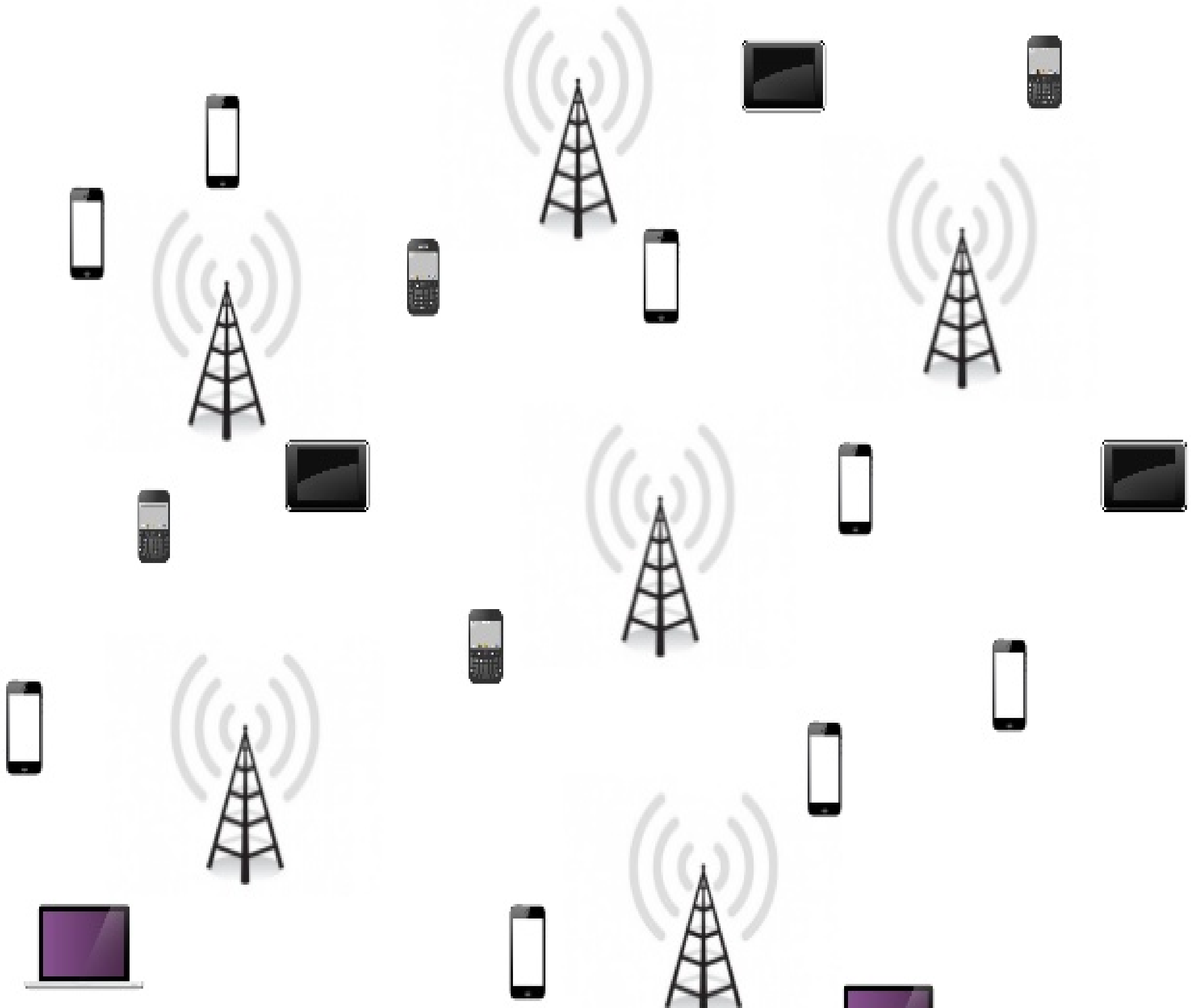
- Graph DBMS 25% of all enterprises by 2017[1]
- Many open-source and research prototypes on distributed graph processing frameworks: Giraph, Pregel, GraphLab, Chaos, GraphX, …

Real-world Graphs are Dynamic

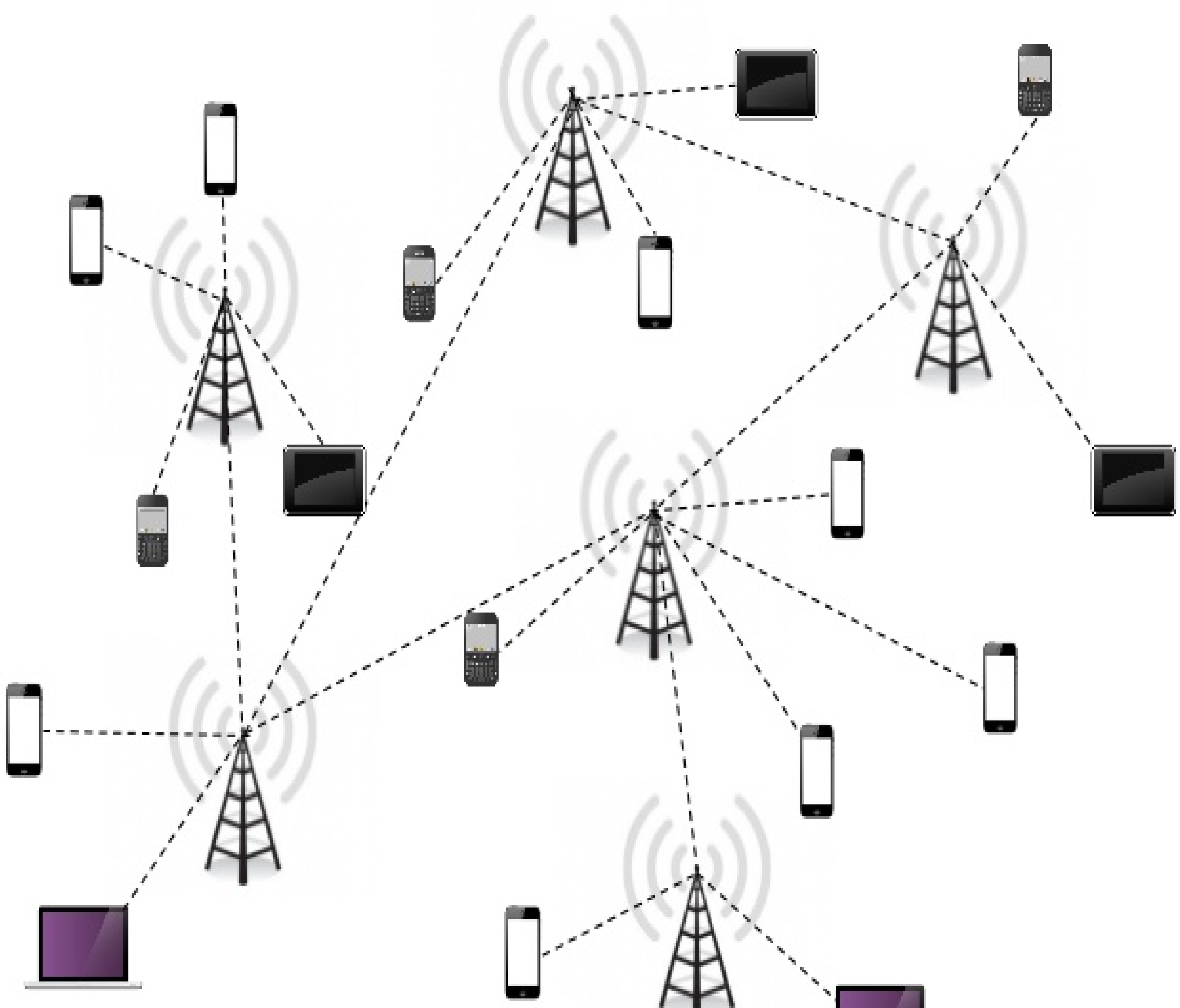
Many interesting business and research insights possible by processing them…

…but little work on incrementally updating computation results or on window-based operations

A Motivating Example...



A Motivating Example...



A Motivating Example...

Retrieve the network state when a disruption happened

Analyze the evolution of hotspot groups in the last day by hour

Track regions of heavy load in the last 10 minutes

What factors explain the performance in the network?

Tegra

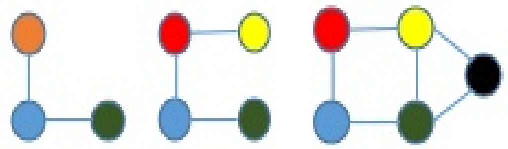
How do we perform efficient computations on time-evolving, dynamically changing graphs?

Goals

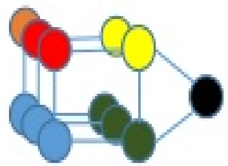
- Create and manage time-evolving graphs
  - *Retrieve the network state when a disruption happened*
- Temporal analytics on windows
  - *Analyze the evolution of hotspot groups in the last day by hour*
- Sliding window computations
  - *Track regions of heavy load in the last 10 minutes interval*
- Mix graph and data parallel computing
  - *What factors explain the performance in the network*

Tegra

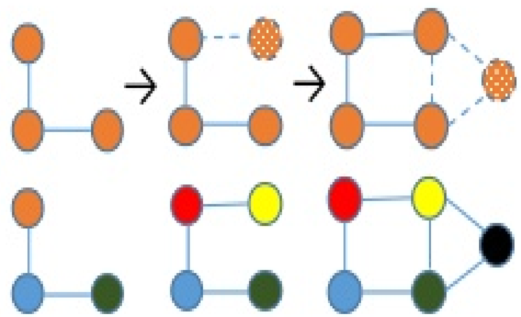
Graph Snapshot Index



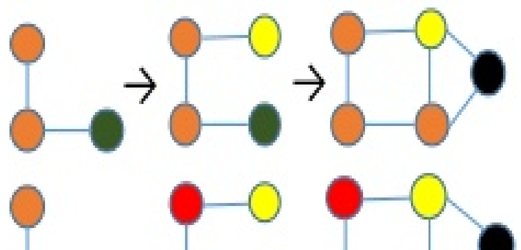
Timelapse Abstraction



Lightweight Incremental Computations



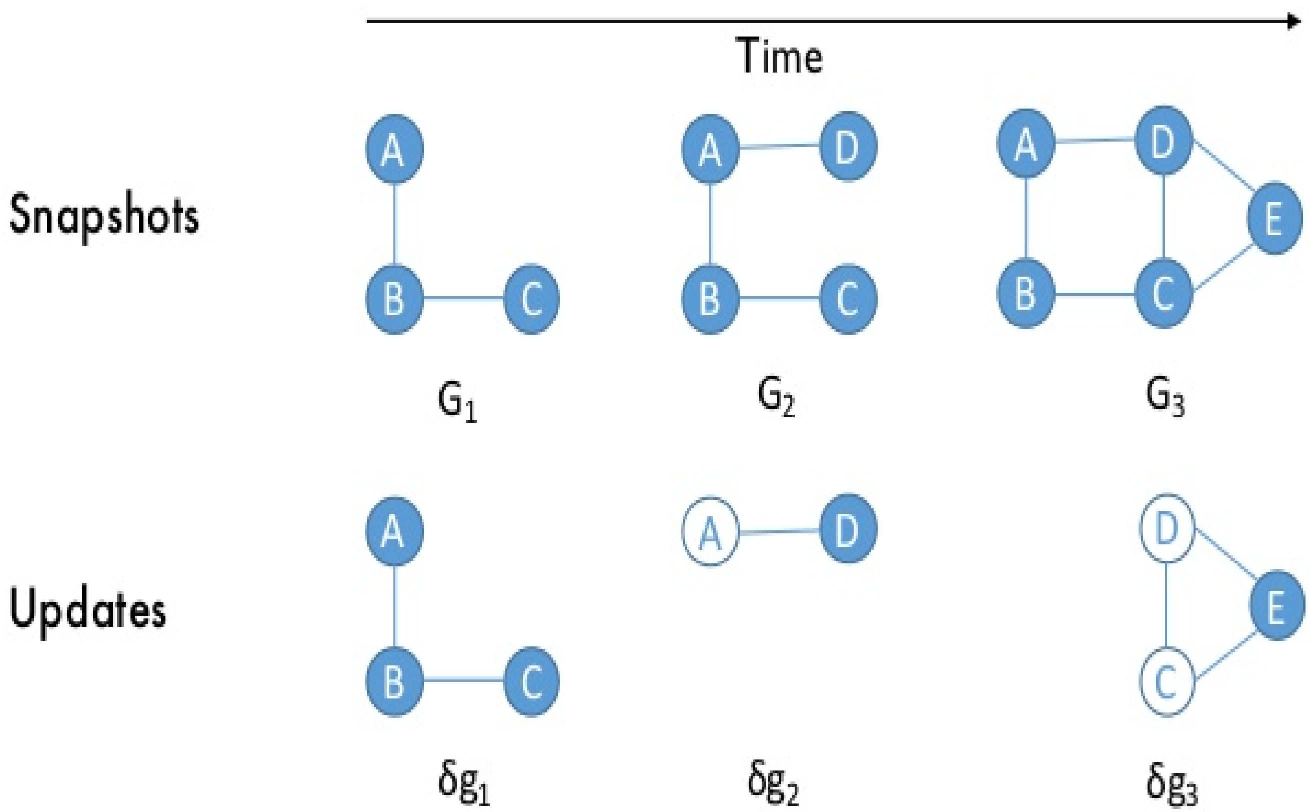
Pause-Shift-Resume Computations



Goals

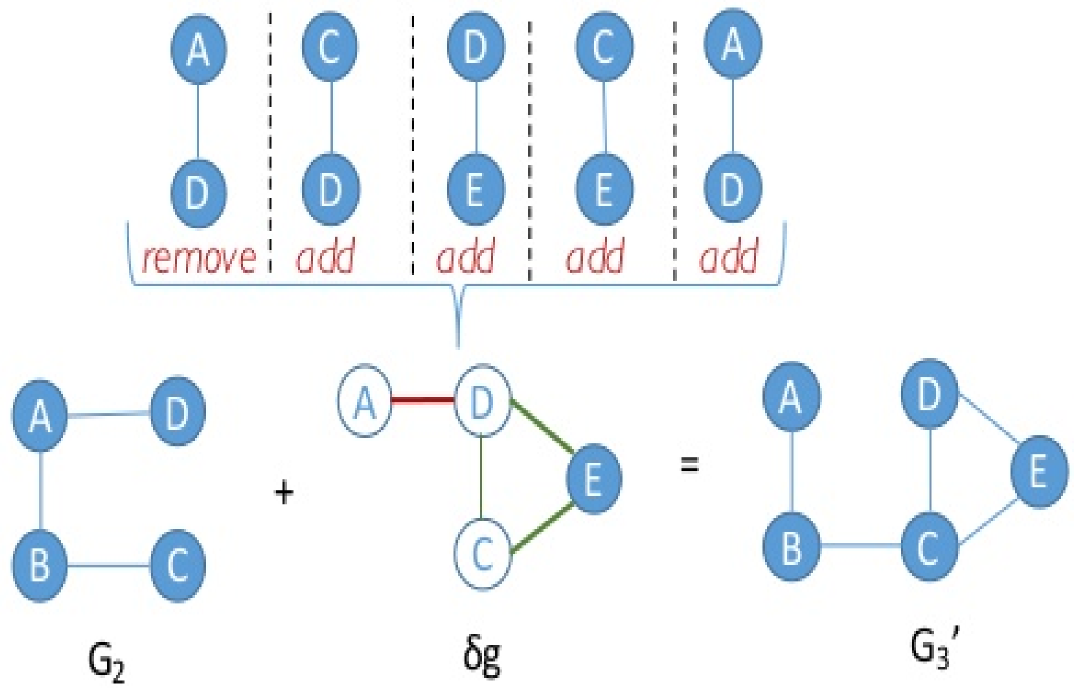
- Create and manage time-evolving graphs using Graph Snapshot Index
  - *Retrieve the network state when a disruption happened*
- Temporal analytics on windows
  - *Analyze the evolution of hotspot groups in the last day by hour*
- Sliding window computations
  - *Track regions of heavy load in the last 10 minutes interval*
- Mix graph and data parallel computing
  - *What factors explain the performance in the network*

Representing Evolving Graphs



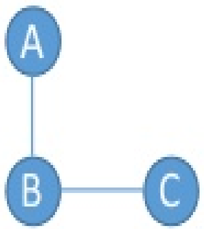
Graph Composition

Updating graphs depend on application semantics

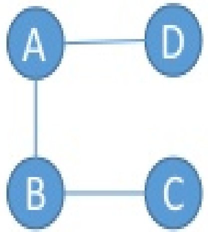


Maintaining Multiple Snapshots

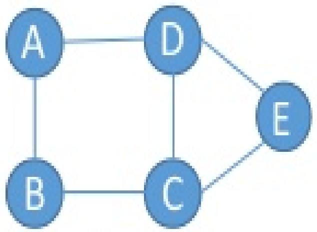
Store entire snapshots



G_1



G_2

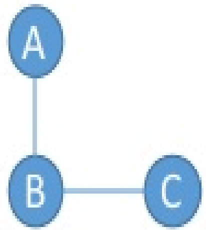


G_3

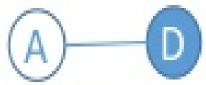
+ Efficient retrieval

- Storage overhead

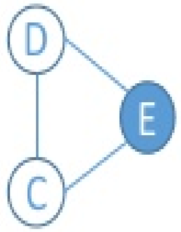
Store only deltas



δg_1



δg_2



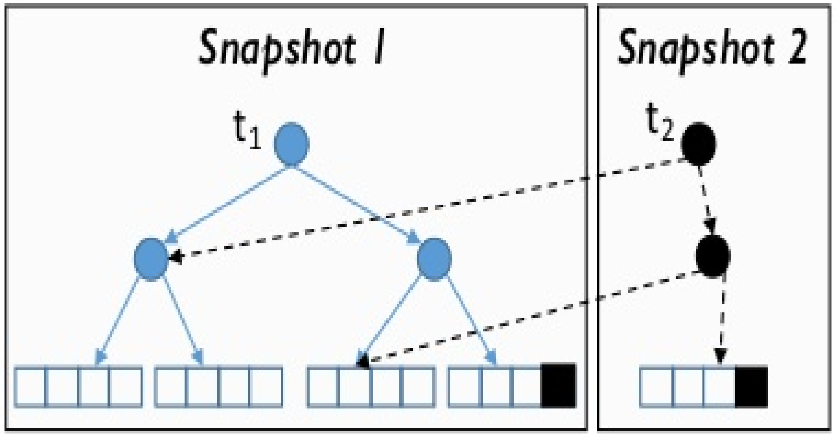
δg_3

+ Efficient storage

- Retrieval overhead

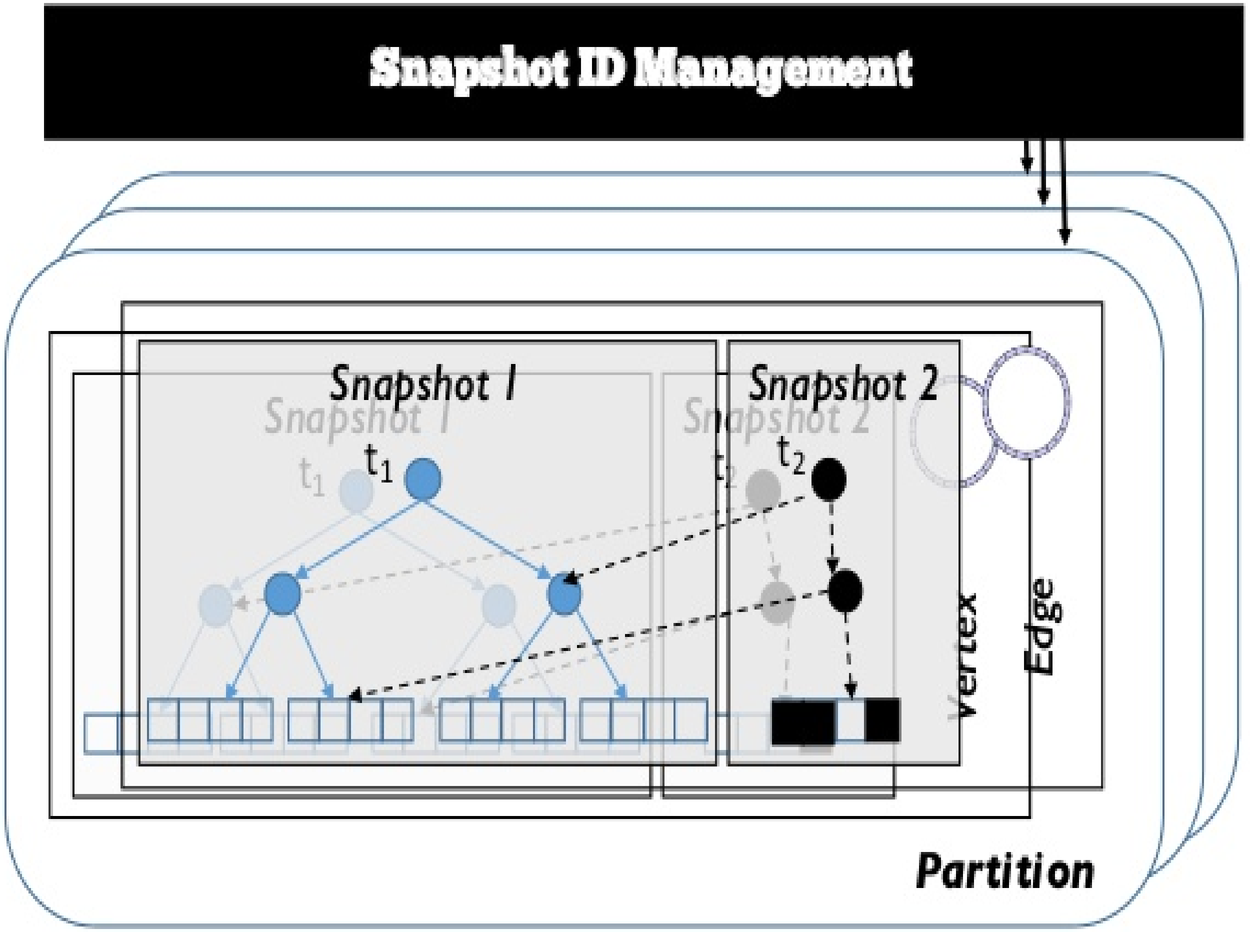
Maintaining Multiple Snapshots

Use a structure-sharing persistent data-structure.

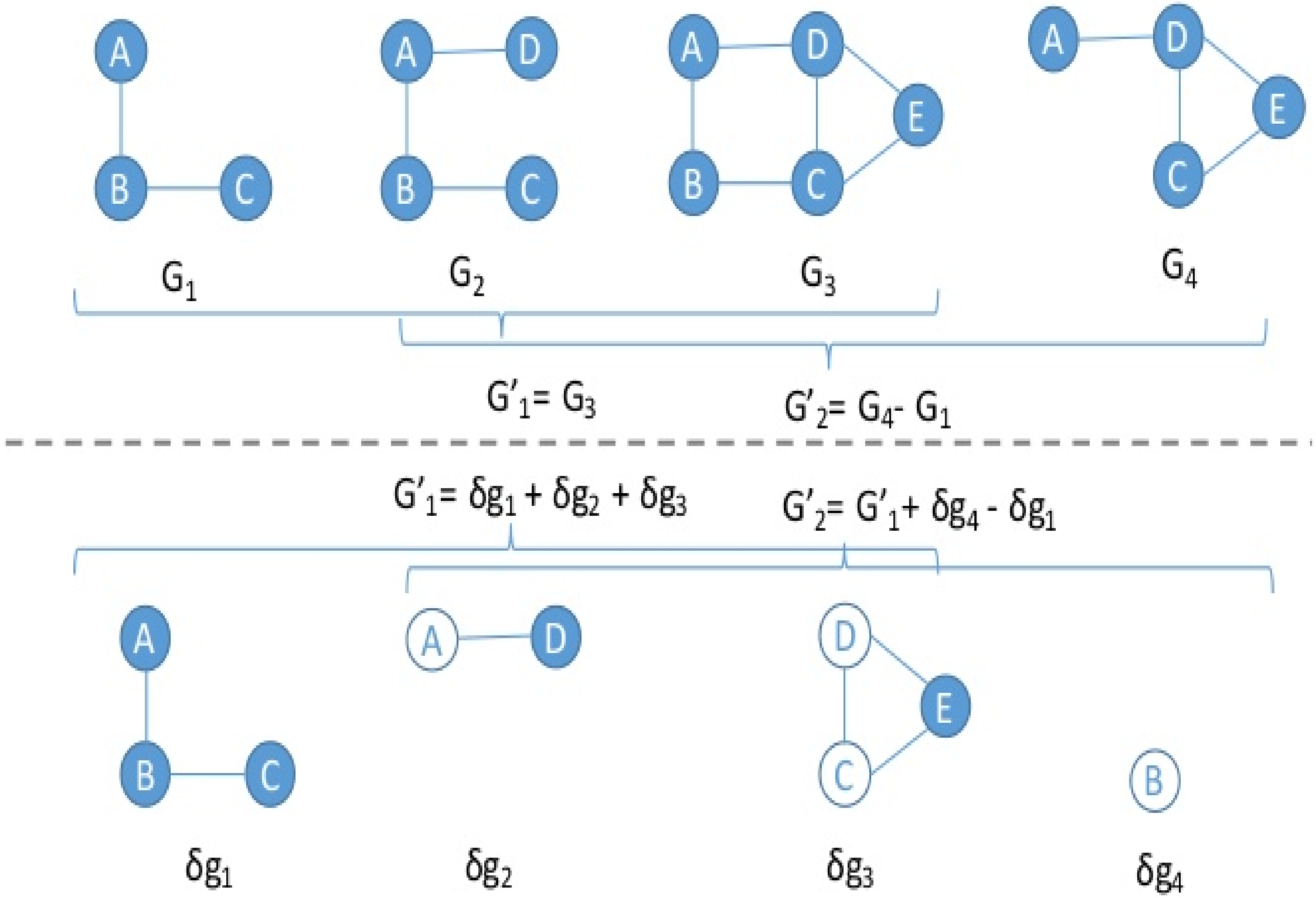


Persistent Adaptive Radix Tree (PART) is one solution available for Spark.

Graph Snapshot Index



Graph Windowing



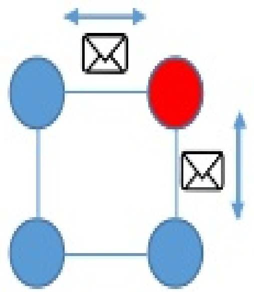
Goals

- Create and manage time-evolving graphs using Distributed Graph Snapshot Index
  - *Retrieve the network state when a disruption happened*
- Temporal analytics on windows using Timelapse Abstraction
  - *Analyze the evolution of hotspot groups in the last day by hour*
- Sliding window computations
  - *Track regions of heavy load in the last 10 minutes interval*
- Mix graph and data parallel computing
  - *What factors explain the performance in the network*

Graph Parallel Computation

Many approaches

- Vertex centric, edge centric, subgraph centric, ...

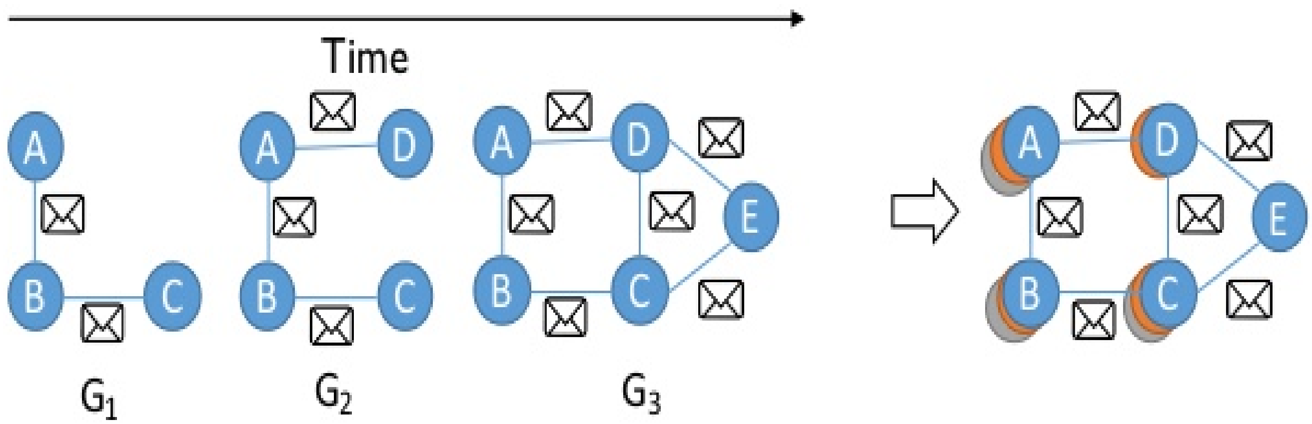


Timelapse

Operations on windows of snapshots result in redundant computation

Timelapse

Instead, expose the temporal evolution of a node

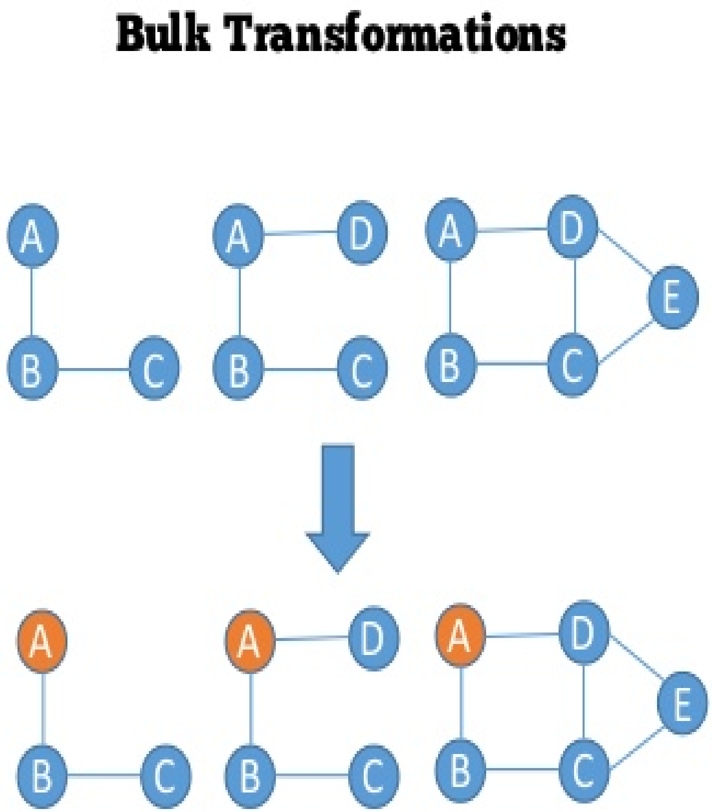


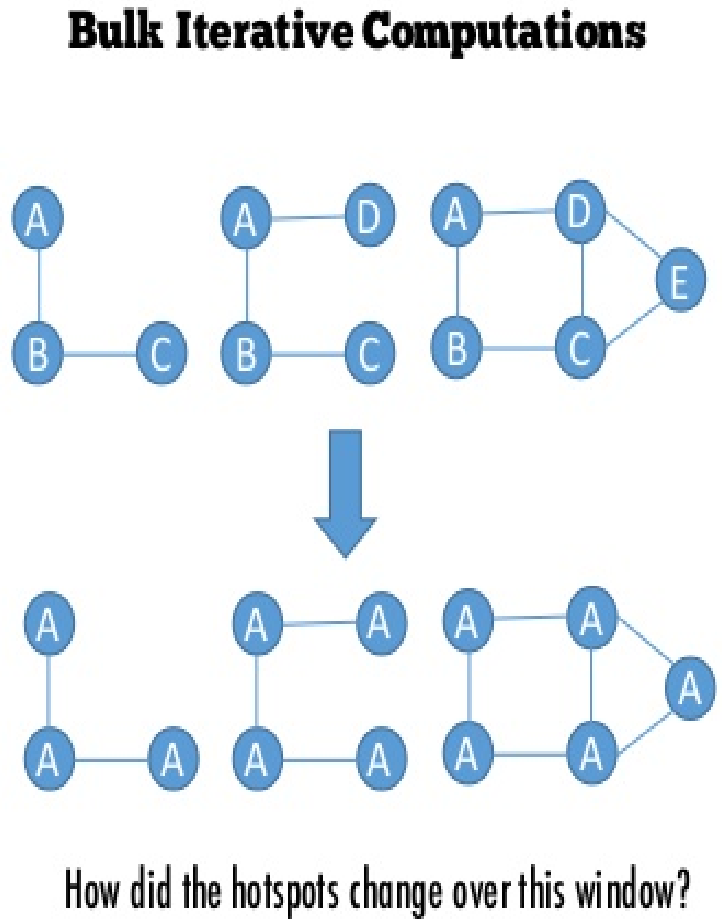
- Significant reduction in messages exchanged between graph nodes
- Avoids redundant computations

Timelapse API

```
class Graph[V, E] {
  // Collection views
  def vertices(sid: Int): Collection[(Id, V)]
  def edges(sid: Int): Collection[(Id, Id, E)]
  def triplets(sid: Int): Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M,
      sids: Array[Int]): Collection[(Int, Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V,
      sids: Array[Int]): Graph[V, E]
  def mapE(f: (Id, Id, E) => E
      sids: Array[Int]): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V,
      sids: Array[Int]): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E,
      sids: Array[Int]): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean,
      sids: Array[Int]): Graph[V, E]
  def reverse(sids: Array[Int]): Graph[V, E]
}
```

Temporal Operations





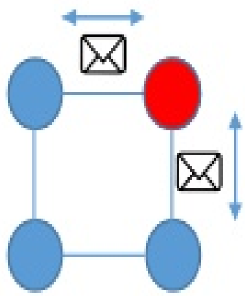
Evolution analysis with no state keeping requirements

Goals

- Create and manage time-evolving graphs using Distributed Graph Snapshot Index
  - *Retrieve the network state when a disruption happened*
- Temporal analytics on windows using Timelapse Abstraction
  - *Analyze the evolution of hotspot groups in the last day by hour*
- Sliding window computations
  - *Track regions of heavy load in the last 10 minutes interval*
- Mix graph and data parallel computing
  - *What factors explain the performance in the network*

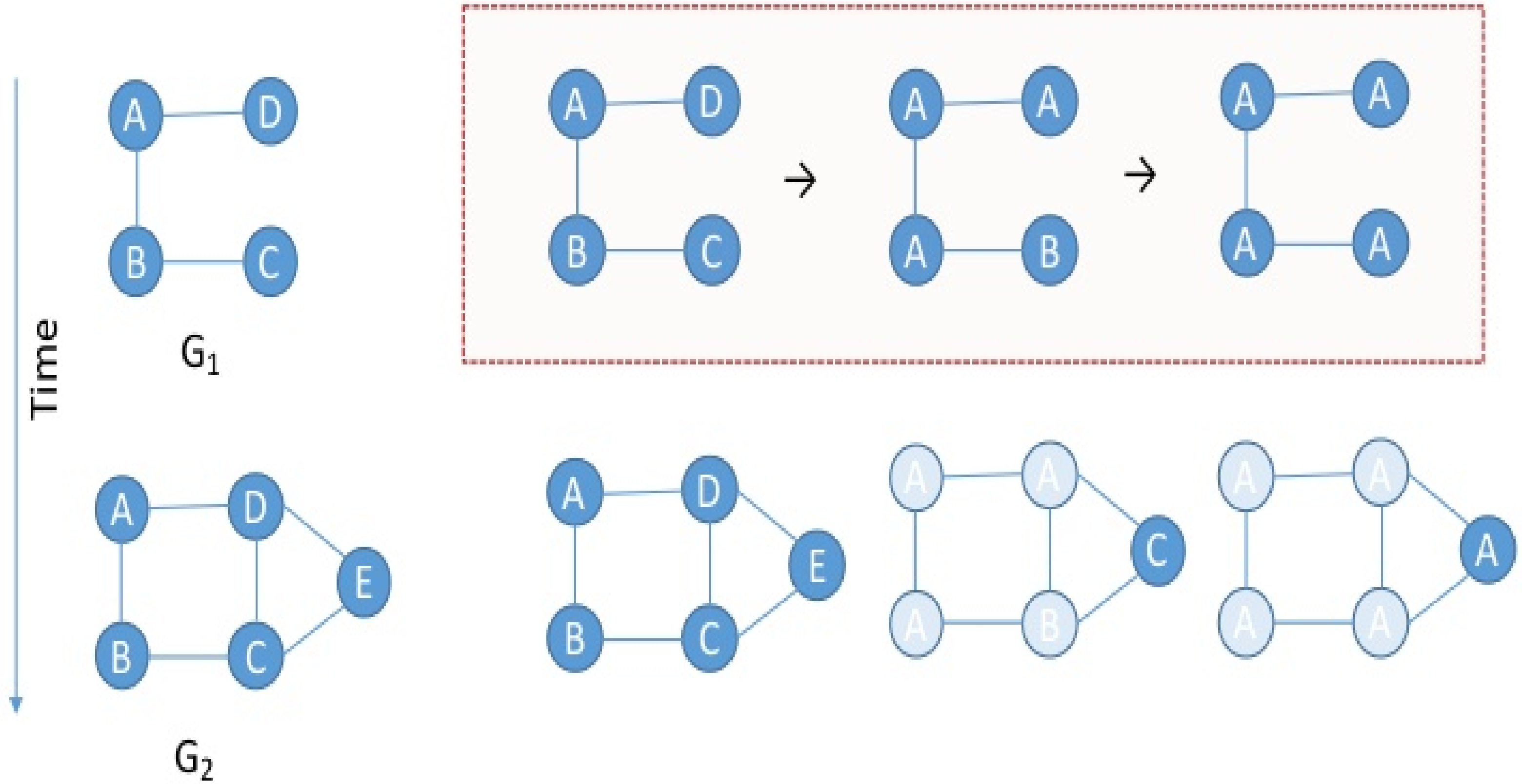
Incremental Computation

- If results from a previous snapshot is available, how can we reuse them?
- Three approaches in the past:
  - Restart the algorithm
    - Redundant computations
  - Memoization (GraphInc[1])
    - Too much state
  - Operator-wise state (Naiad[2,3])
    - Too much overhead



[1]Facilitating real- time graph mining, CloudDB '12

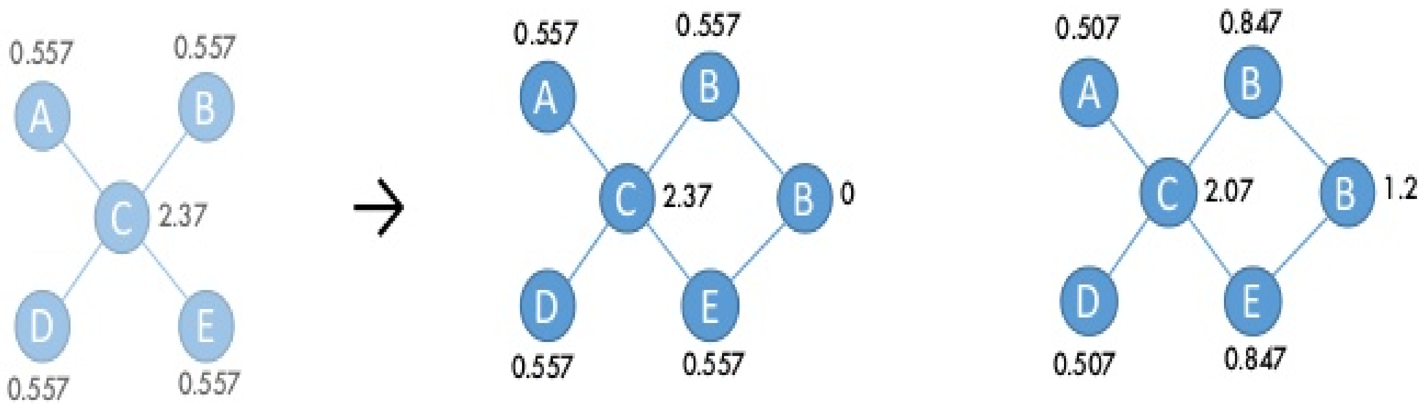
Incremental Computation



- Can keep state as an efficient time-evolving graph
- Not limited to vertex-centric computations

Incremental Computation

- Some iterative graph algorithms are robust to graph changes
  - Allow them to proceed without keeping any state



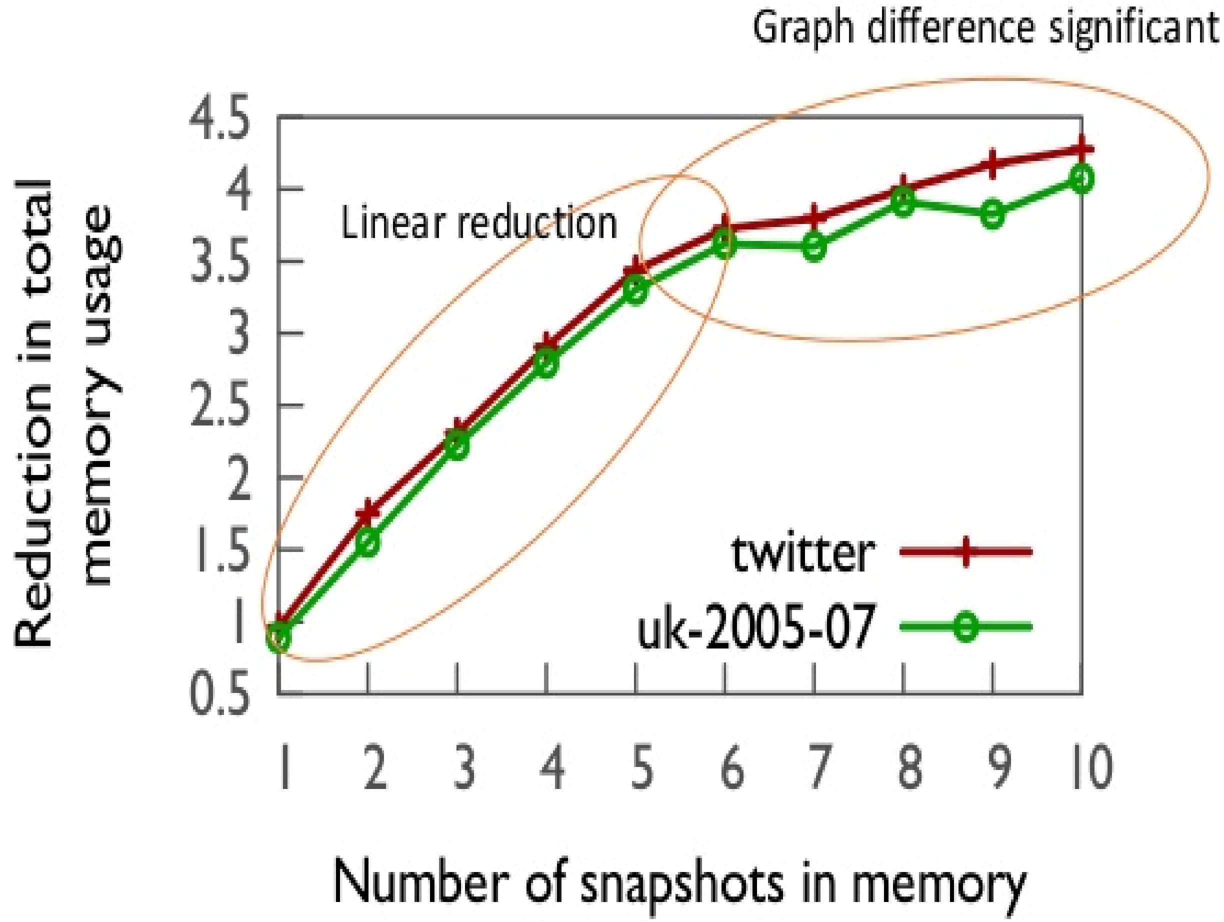
Goals

- Create and manage time-evolving graphs using Distributed Graph Snapshot Index
  - *Retrieve the network state when a disruption happened*
- Temporal analytics on windows using Timelapse Abstraction
  - *Analyze the evolution of hotspot groups in the last day by hour*
- Sliding window computations
  - *Track regions of heavy load in the last 10 minutes interval*
- Mix graph and data parallel computing
  - *What factors explain the performance in the network*

Implementation & Evaluation

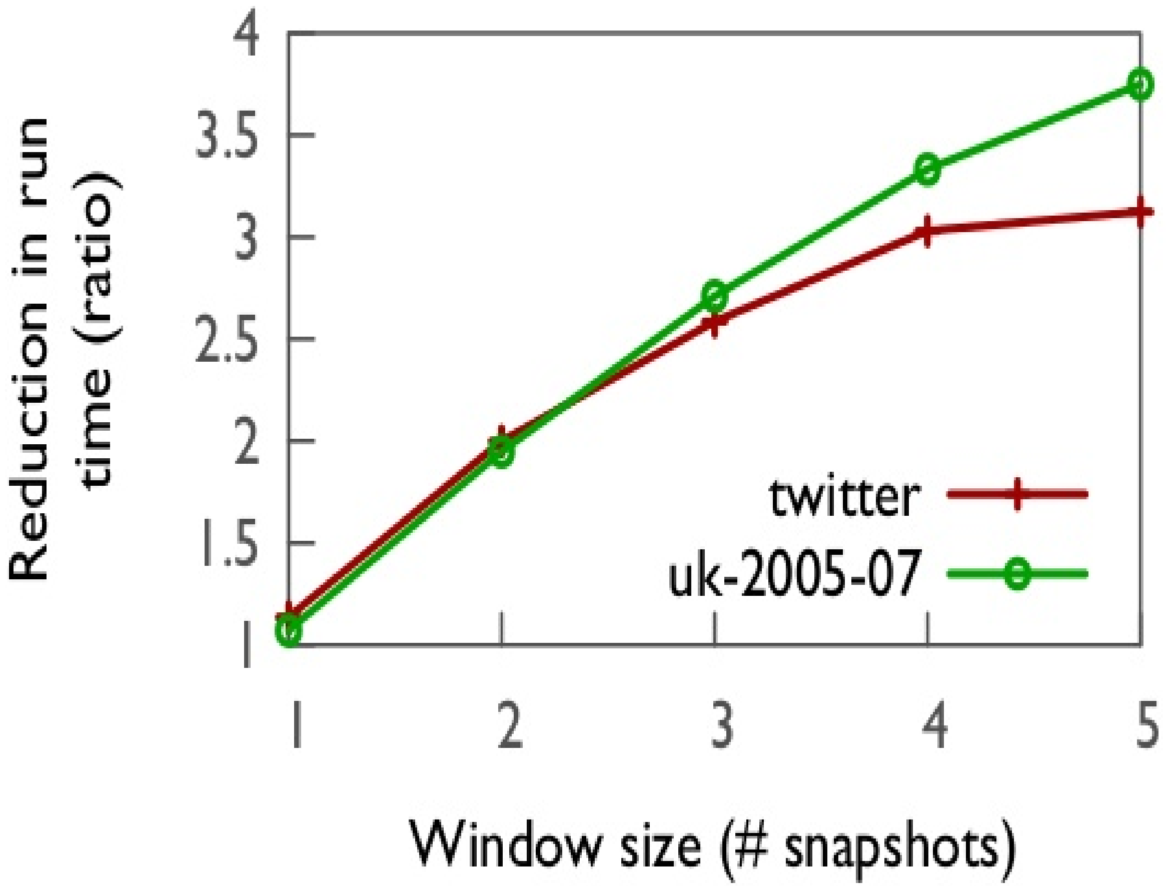
- Implemented as a major enhancement to GraphX
- Evaluated on two open source real-world graphs
  - **Twitter:** 41,652,230 vertices, 1,468,365,182 edges
  - **uk-2007:** 105,896,555 vertices, 3,738,733,648 edges

Preliminary Evaluation



Tegra can pack more snapshots in memory

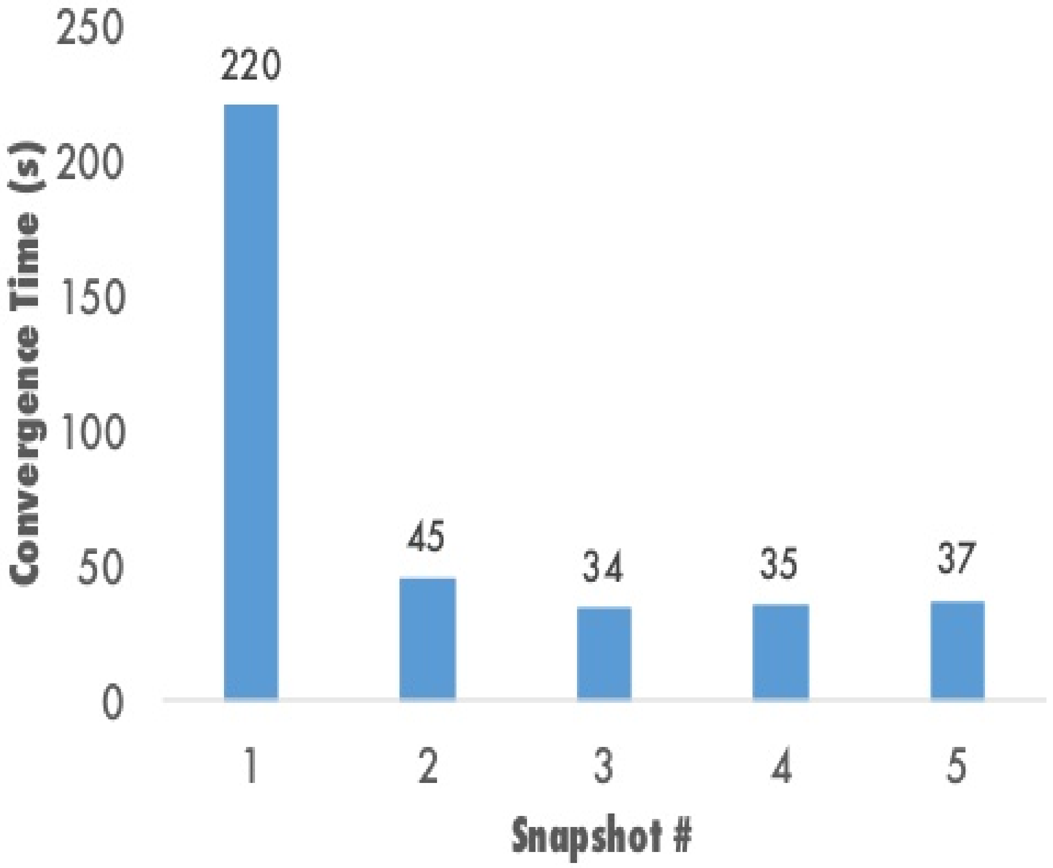
Preliminary Evaluation



Timelapse results in run time reduction

Preliminary Evaluation

Effectiveness of incremental computation



Summary

- Processing time-evolving graph efficiently can be useful.
- Efficient storage of multiple snapshots and reducing communication between graph nodes key to evolving graph analysis.

Ongoing Work

- Expand timelapse and its incremental computation model to other graph-parallel paradigms
  - Other interesting graph algorithms:
    - Fraud detection/prediction/incremental pattern matching
- Add graph querying support
  - Graph queries and analytics in a single system
- **Stay tuned for code release!**