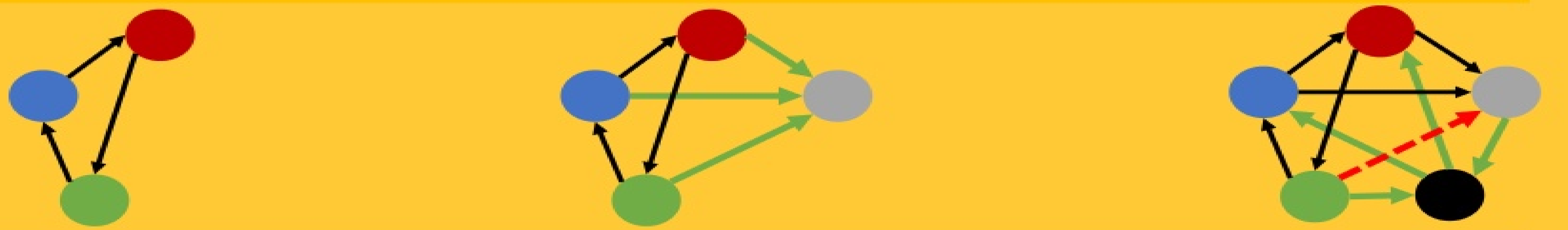
Tegra

Time-evolving Graph Processing on Commodity Clusters



Spark Summit East
8 February 2017

Anand Iyer

Qifan Pu

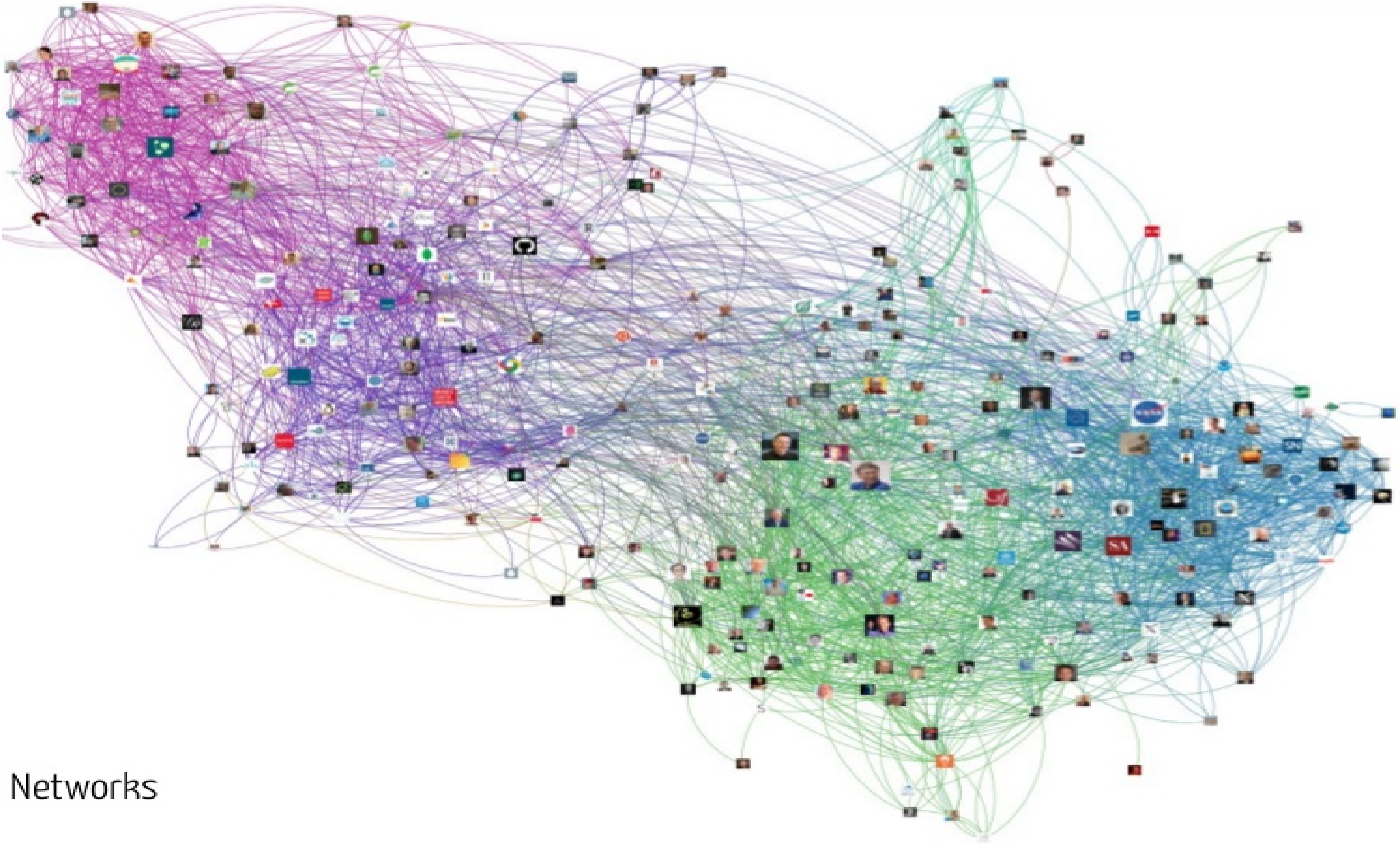
Joseph Gonzalez

Ion Stoica

About Me

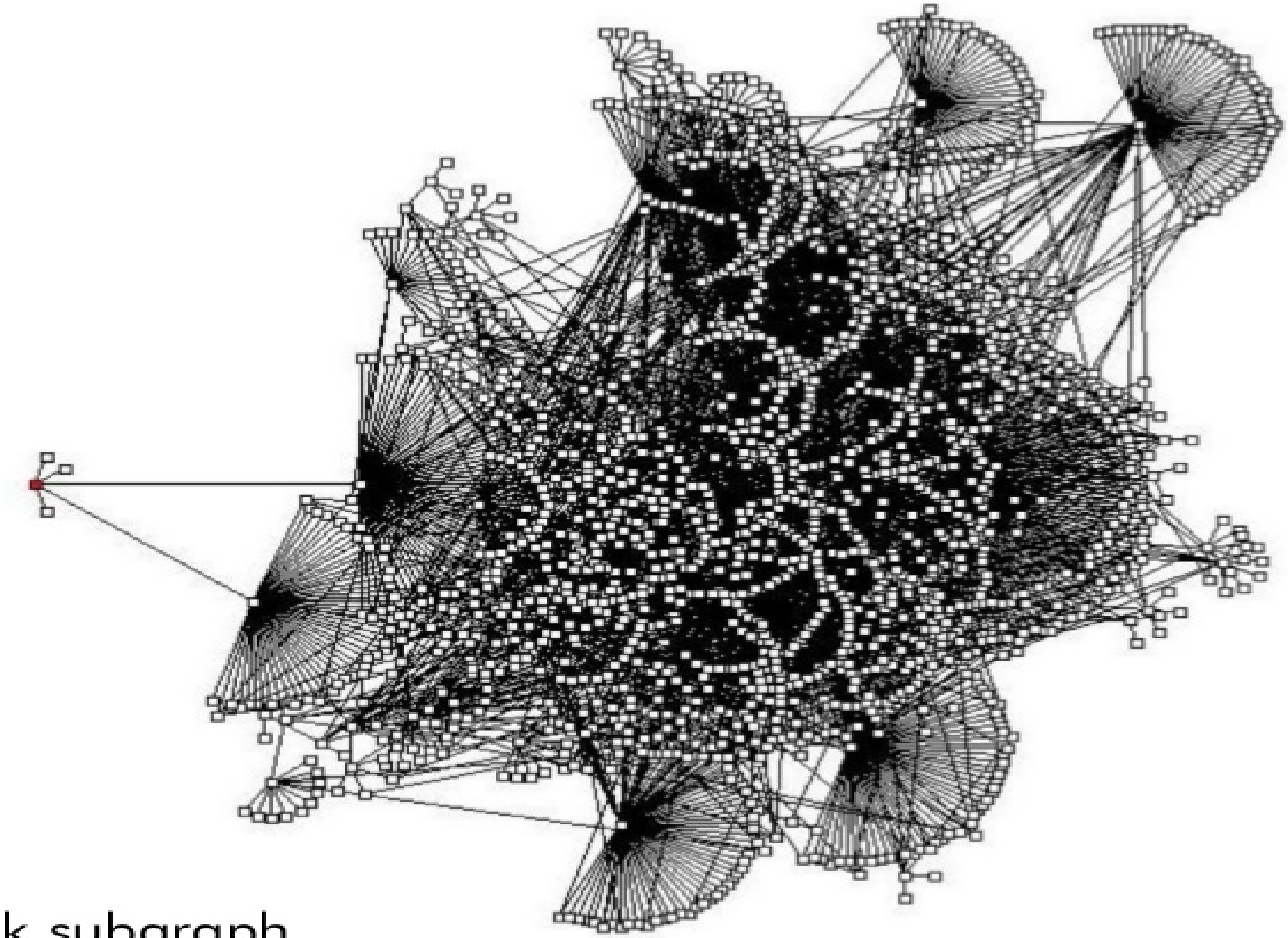
- PhD Candidate at AMP/RISE Lab at UC Berkeley
- Thesis on time-evolving graph processing
- Previous work:
  - Collaborative energy diagnosis for smartphones (carat.cs.berkeley.edu)
  - Approximate query processing (BlinkDB)
  - Cellular Network Analytics
  - Fundamental trade-offs in applying ML to real-time datasets

Graphs are everywhere...



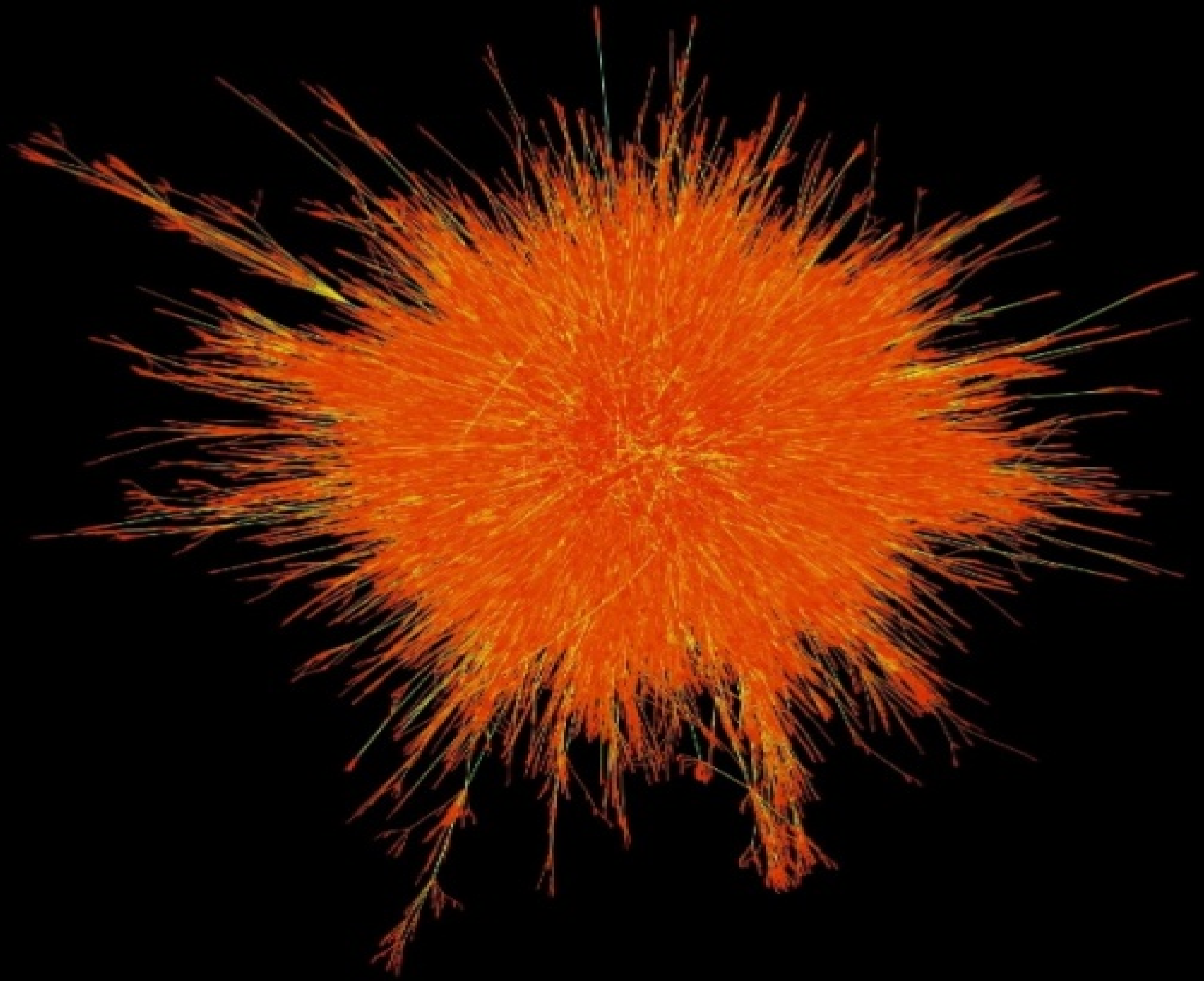
Social Networks

Graphs are everywhere...



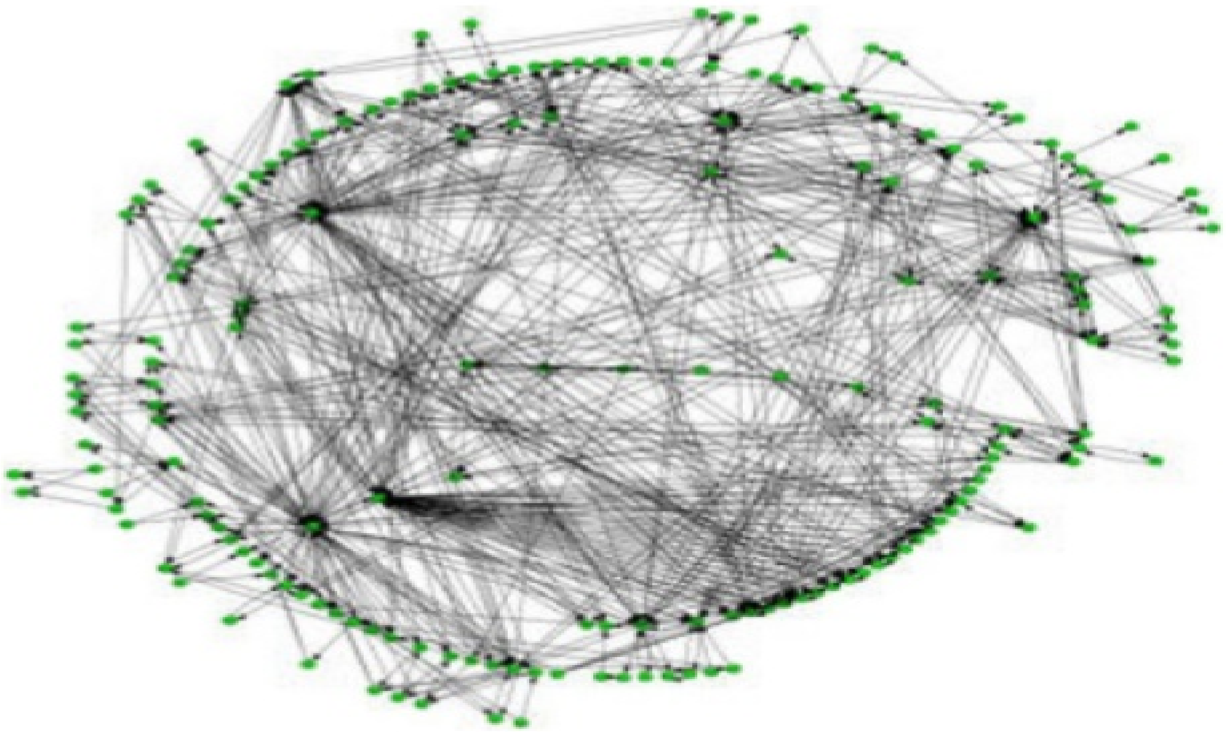
Gnutella network subgraph

Graphs are everywhere...



SNAP@web-Google. 1316100 nodes, 4925011 edges.

Graphs are everywhere...



Metabolic network of a single cell organism

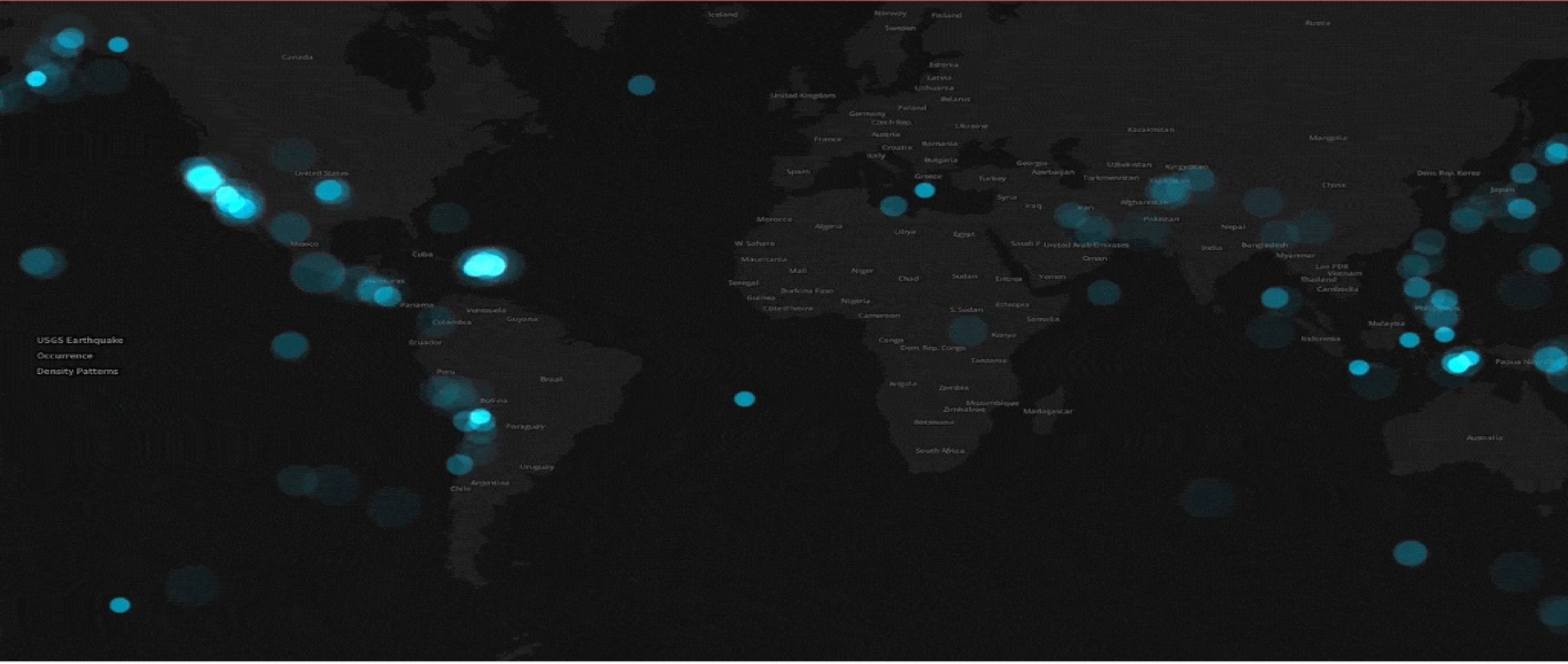
Tuberculosis

Plenty of interest in processing them

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

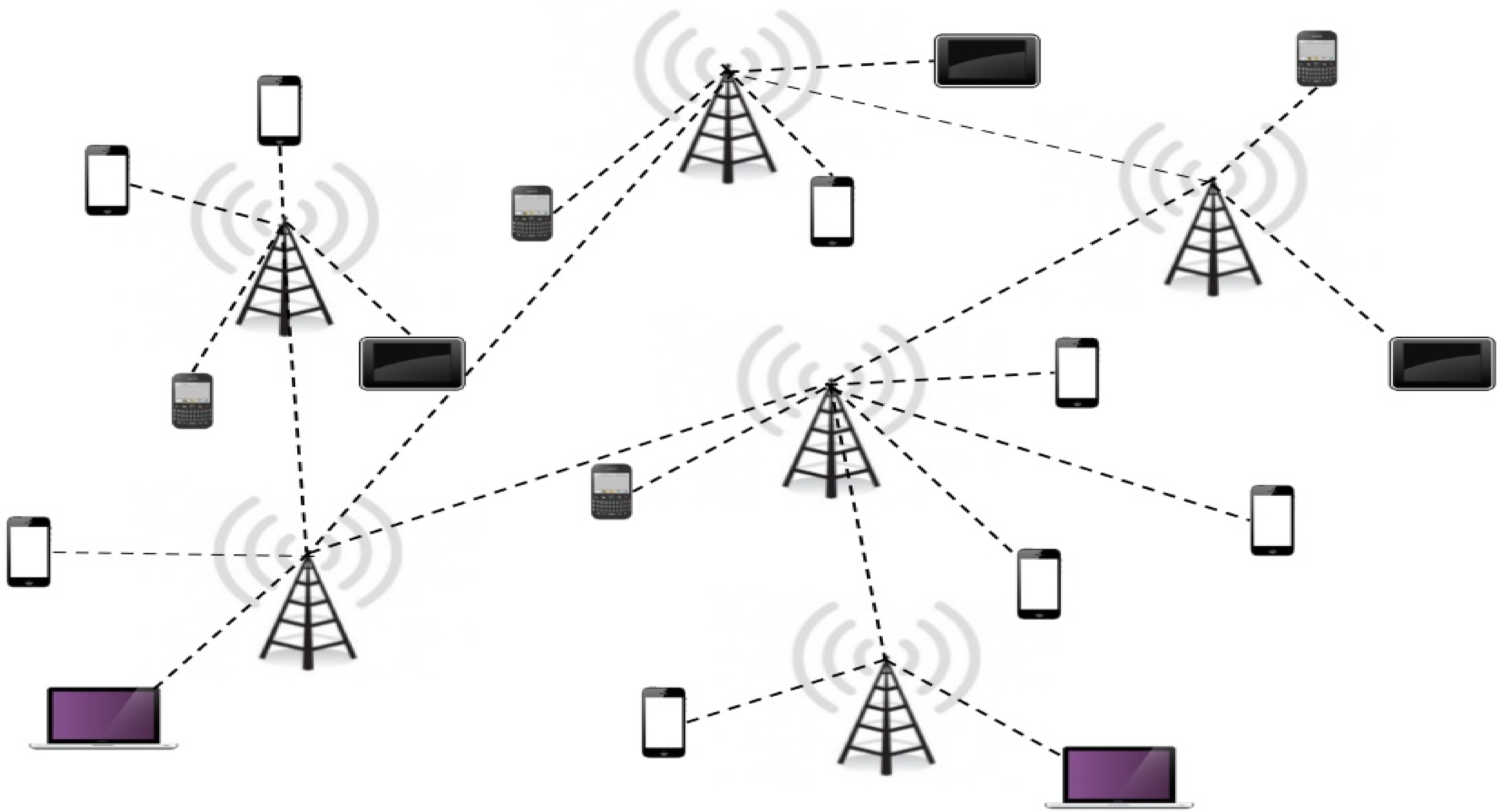
[1]Forrester Research

Real-world Graphs are Dynamic



Earthquake Occurrence Density

Real-world Graphs are Dynamic



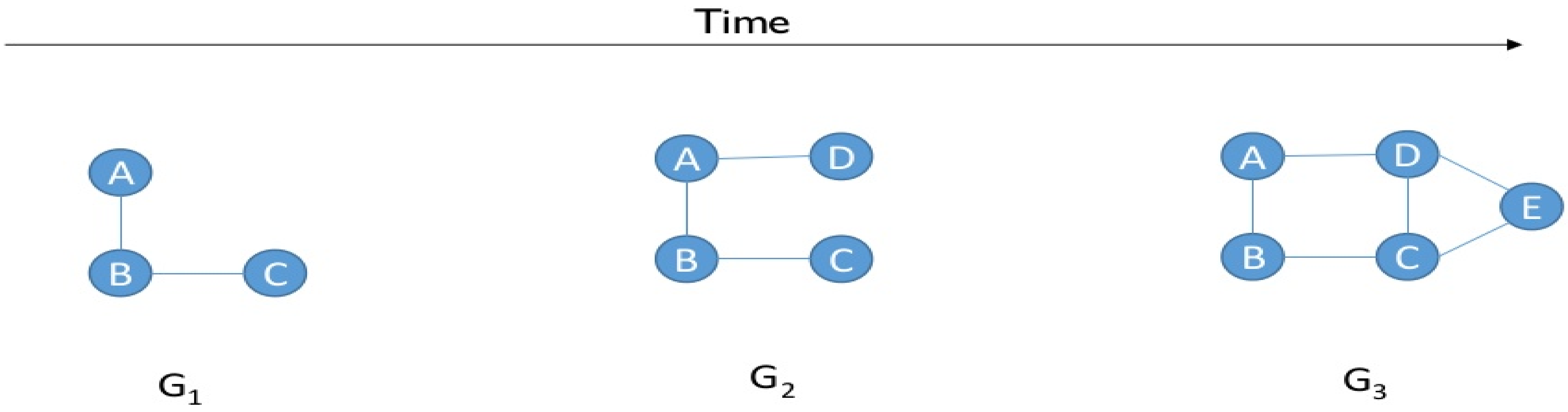
Real-world Graphs are Dynamic

Processing Time-evolving Graphs

Many interesting business and research insights possible by processing such dynamic graphs…

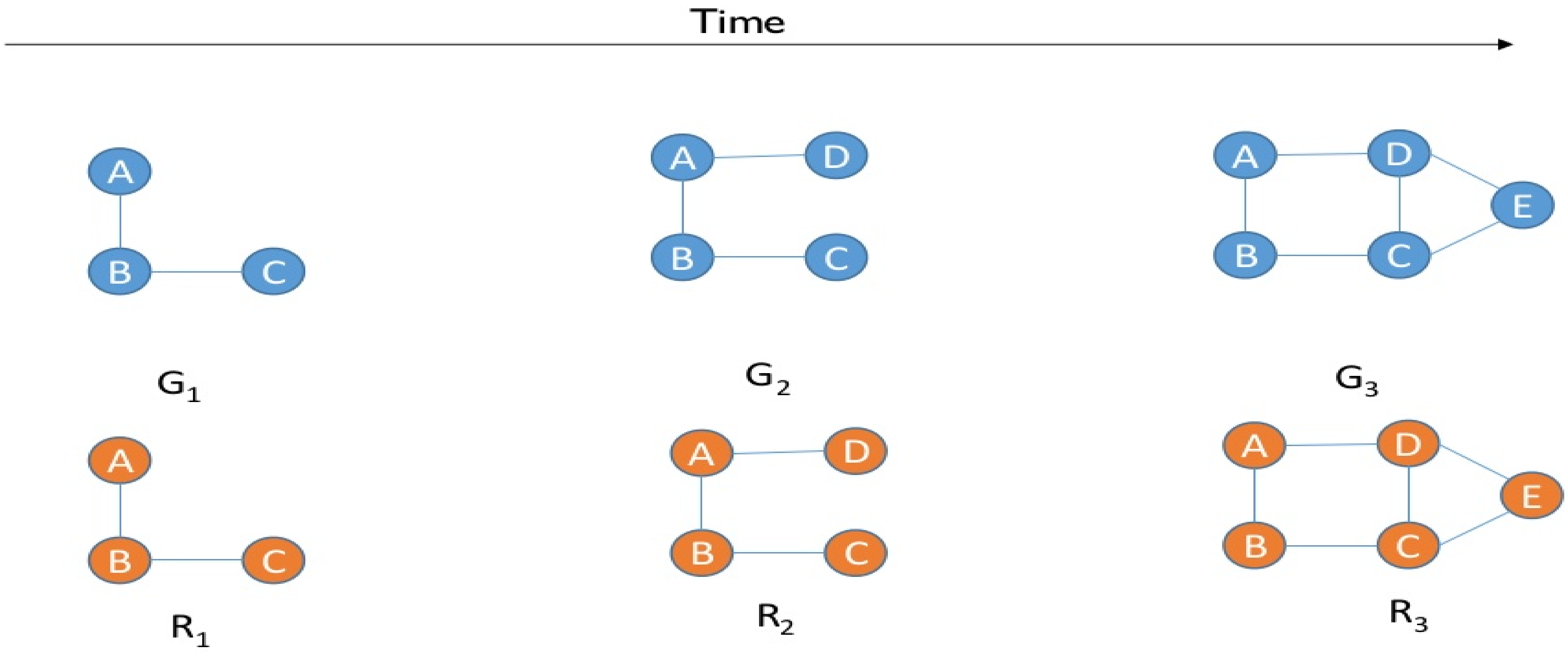
… little or no work in supporting such workloads in existing graph-processing frameworks

Challenge #1: Storage



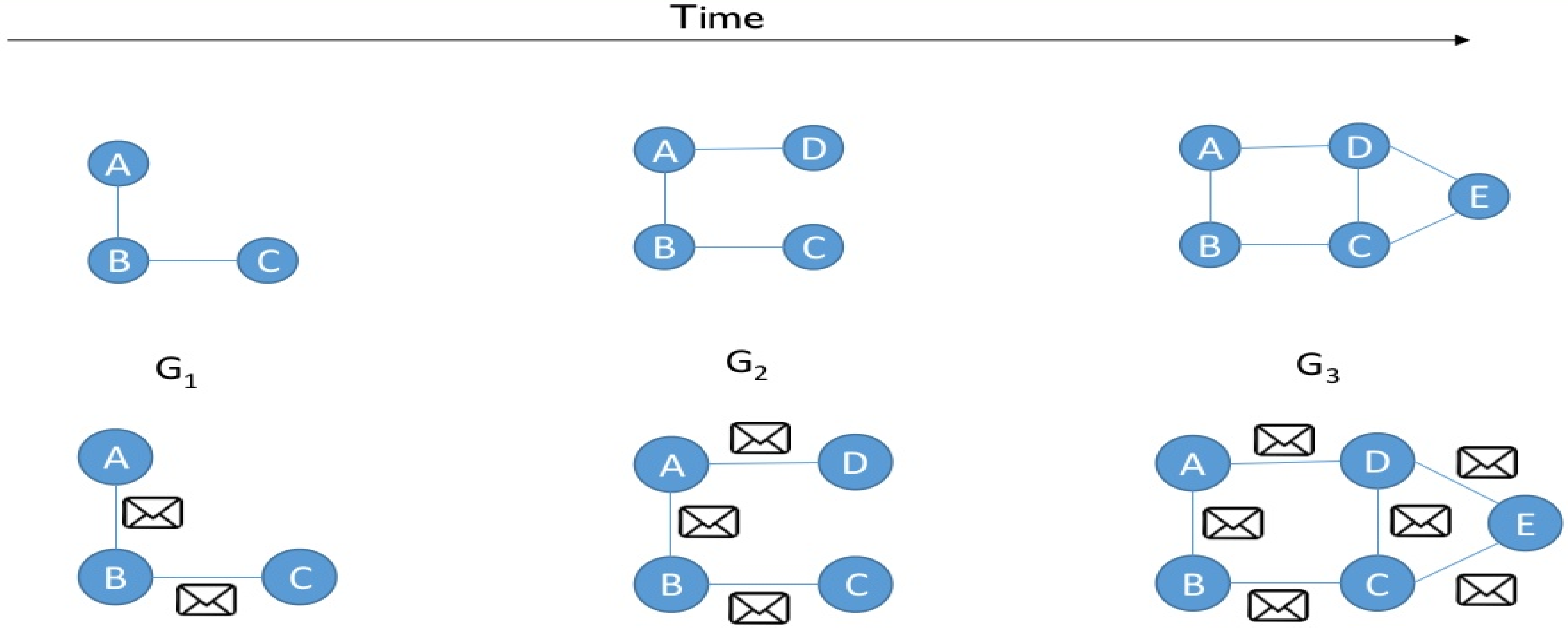
Redundant storage of graph entities over time

Challenge #2: Computation



Wasted computation across snapshots

Challenge #3: Communication



Duplicate messages sent over the network

Share

How do we process time-evolving, dynamically changing graphs efficiently?

Storage

Communication

Computation

Tegra

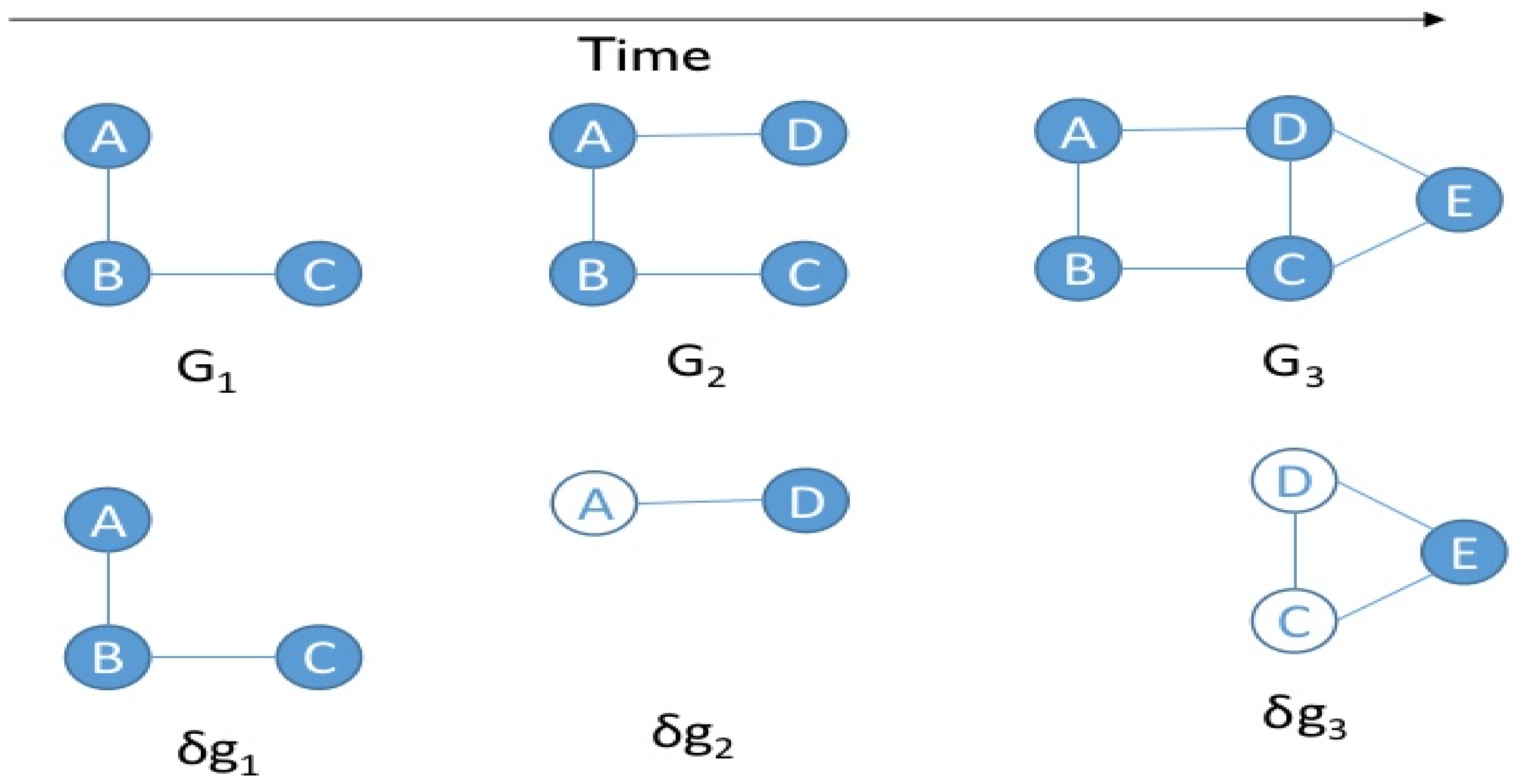
How do we process time-evolving, dynamically changing graphs efficiently?

Share

Storage
Communication
Computation

Tegra

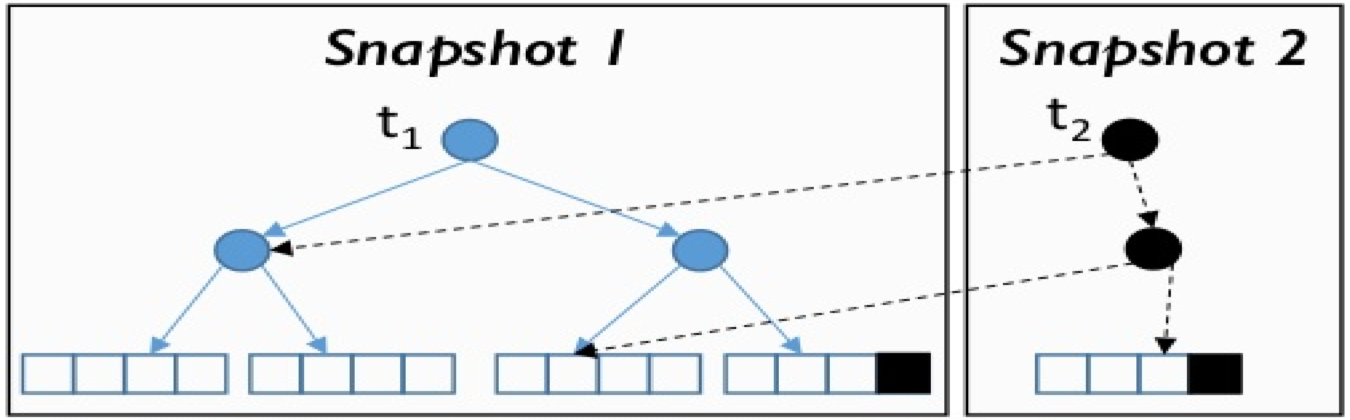
Sharing Storage



Storing deltas result in the most optimal storage, but creating snapshot from deltas can be expensive!

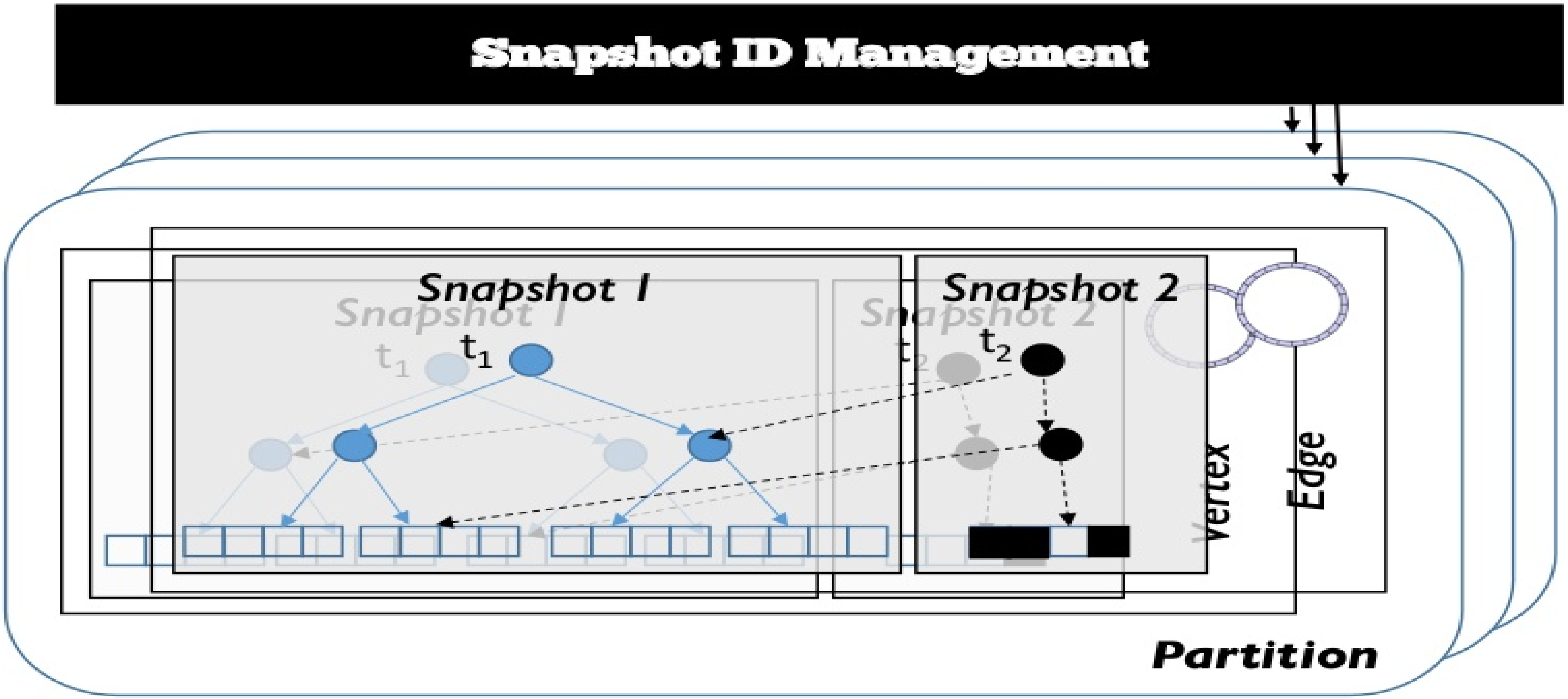
A Better Storage Solution

Use a persistent datastructure



Store snapshots in Persistent Adaptive Radix Trees (PART)

Graph Snapshot Index



Shares structure between snapshots, and enables efficient operations

How do we process time-evolving, dynamically changing graphs efficiently?

Share

Storage

Communication

Computation

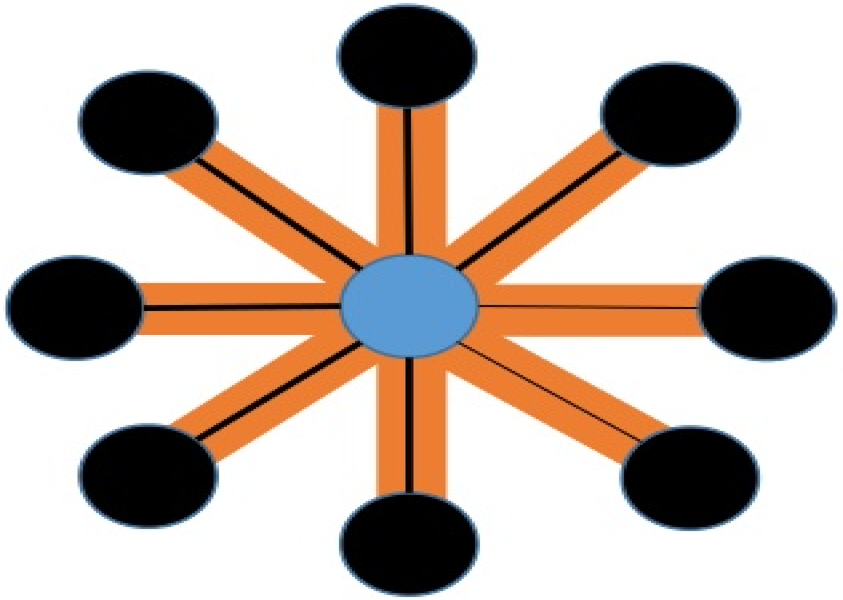
Tegra

Graph Parallel Abstraction - GAS

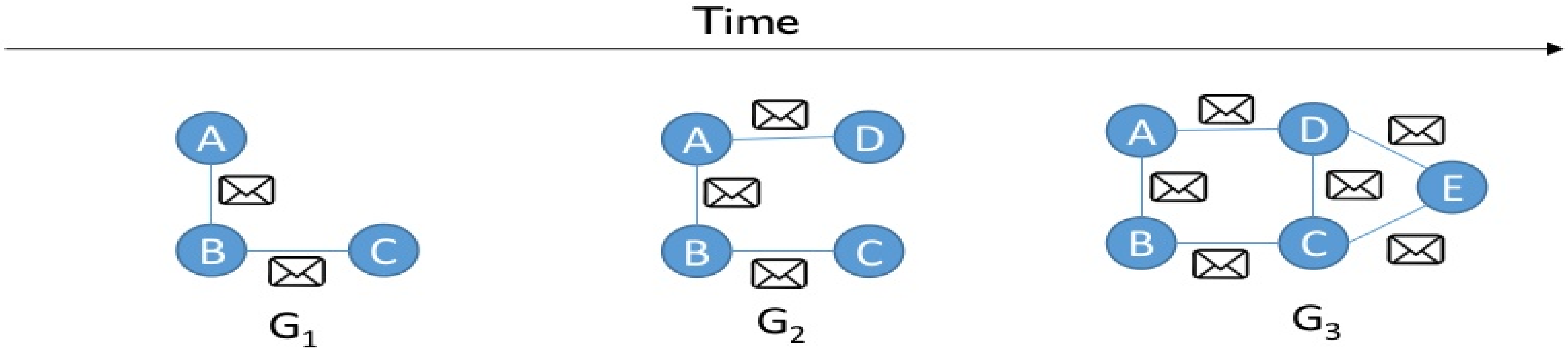
Gather: Accumulate information from neighborhood

Apply: Apply the accumulated value

Scatter: Update adjacent edges & vertices with updated value



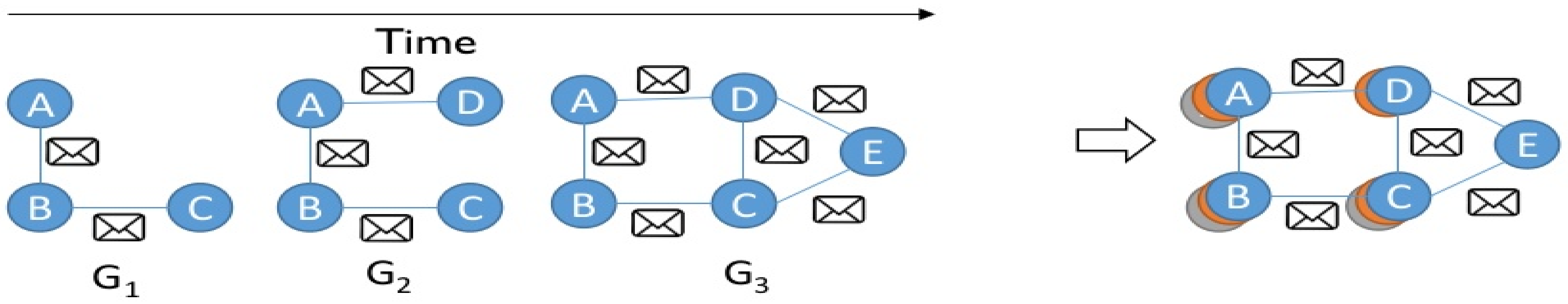
Processing Multiple Snapshots




```
for (snapshot in snapshots) {
    for (stage in graph-parallel-computation) {…}
}
```

Reducing Redundant Messages

```
for (step in graph-parallel-computation) {
      for (snapshot in snapshots) {…}
}
```



Can potentially avoid large number of redundant messages

How do we process time-evolving, dynamically changing graphs efficiently?

Share

Storage
Communication
Computation

Tegra

Updating Results

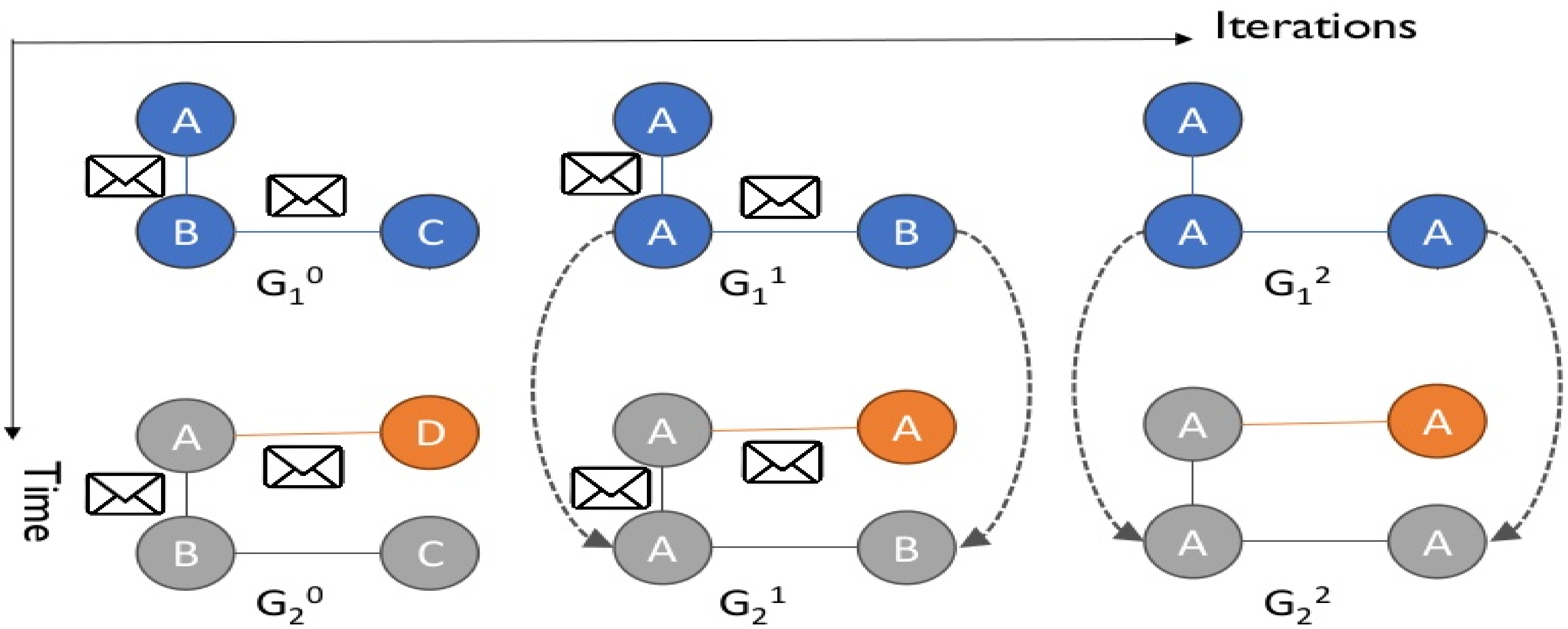
- If result from a previous snapshot is available, how can we reuse them?
- Three approaches in the past:
  - Restart the algorithm
    - Redundant computations
  - Memoization (GraphInc[1])
    - Too much state
  - Operator-wise state (Naiad[2,3])
    - Too much overhead
    - Fault tolerance

[1] *Facilitating real- time graph mining, CloudDB '12*
[2] *Naiad: A timely dataflow system, SOSP '13*
[3] *Differential dataflow, CIDR '13*

Key Idea

- Leverage how GAS model executes computation
- Each iteration in GAS modifies the graph by a little
  - Can be seen as another time-evolving graph!
- Upon change to a graph:
  - Mark parts of the graph that changed
  - Expand the marked parts to involve regions for recomputation in every iteration
  - Borrow results from parts not changed

Incremental Computation



Larger graphs and more iterations can yield significant improvements

API

```
val v = sqlContext.createDataFrame(List(
      ("a", "Alice"),
      ("b", "Bob"),
      ("c", "Charlie")
)).toDF("id", "name").indexed()

val e = sqlContext.createDataFrame(List(
      ("a", "b", "friend"),
      ("b", "c", "follow"),
      ("c", "b", "follow)
)).toDF("src", "dst", "relationship").indexed()

val g = GraphFrame(v, e)

val g1 = g.update(v1, e1)
```

API: Incremental Computations

```
val g = GraphFrame(v, e)

val result = g.triangleCount.run()

val g1 = g.update(v1,e1)

val result1 = g1.triangleCount.run(result)
```

API: Computations on Multiple Graphs

```
val g = GraphFrame(v, e)
val g1 = g.update(v1,e1)
val g2 = g1.update(v2,e2)
val g3 = g1.update(v3,e3)
val results =
g3.triangleCount.runOnSnapshots(start, end)
```

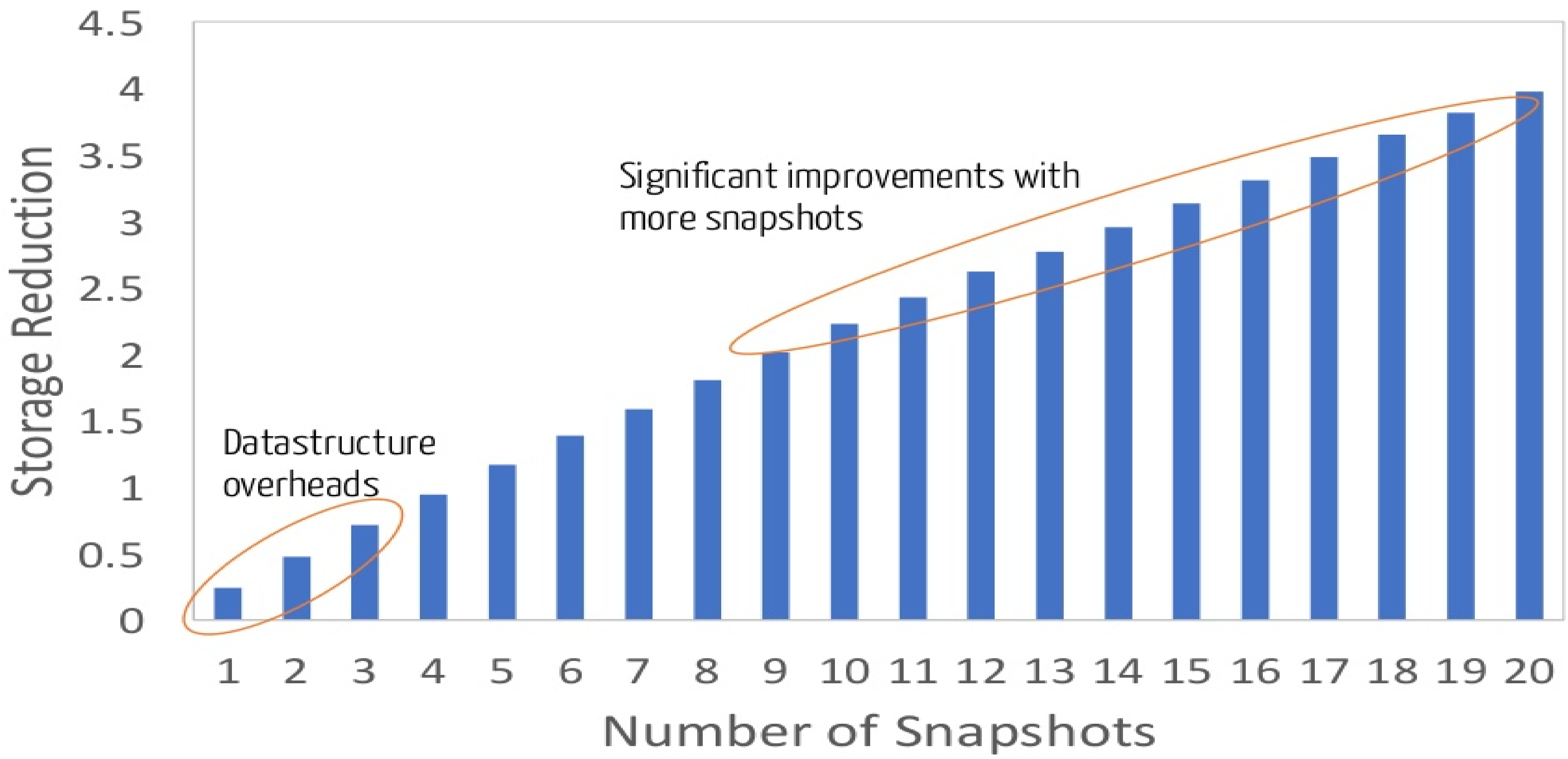
API

```
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]
}
```

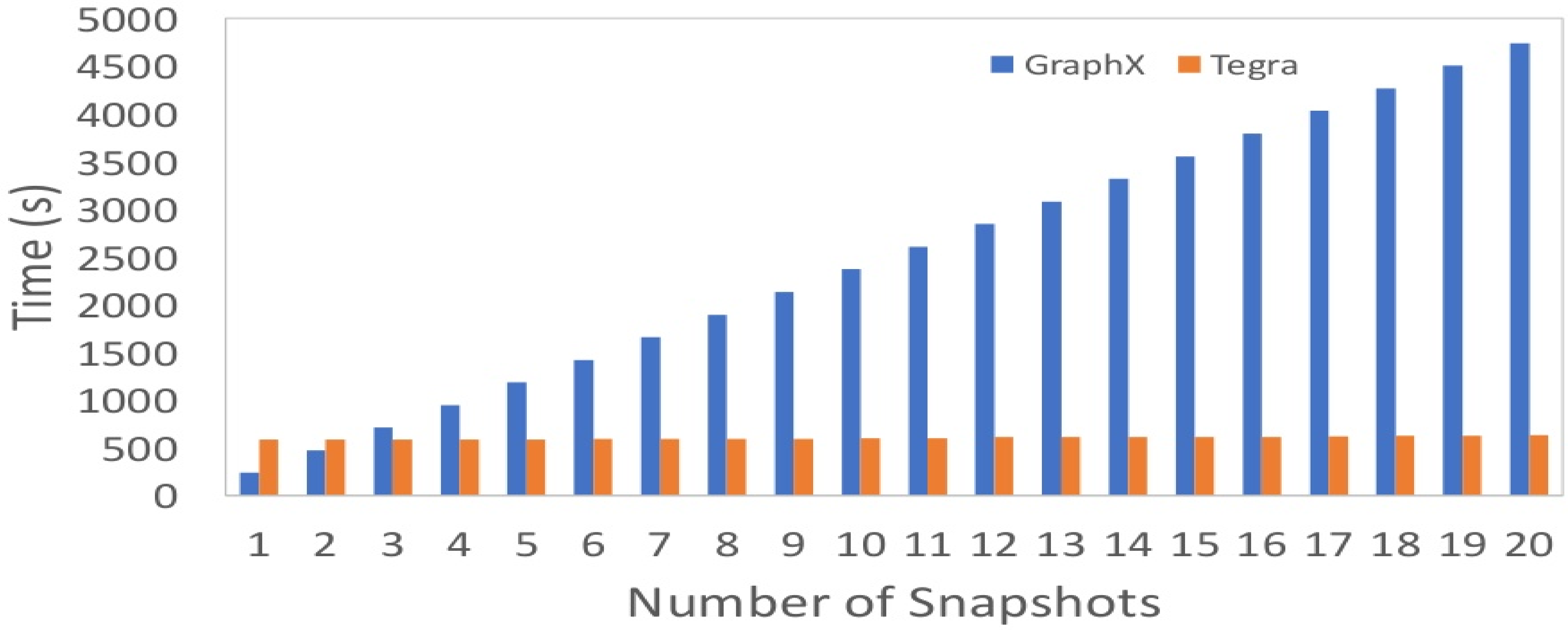
Implementation & Evaluation

- Implemented on Spark 2.0
  - Extended dataframes with versioning information and iterate operator
  - Extended GraphX API to allow computation on multiple snapshots
- Preliminary evaluation on two real-world graphs
  - **Twitter:** 41,652,230 vertices, 1,468,365,182 edges
  - **uk-2007:** 105,896,555 vertices, 3,738,733,648 edges

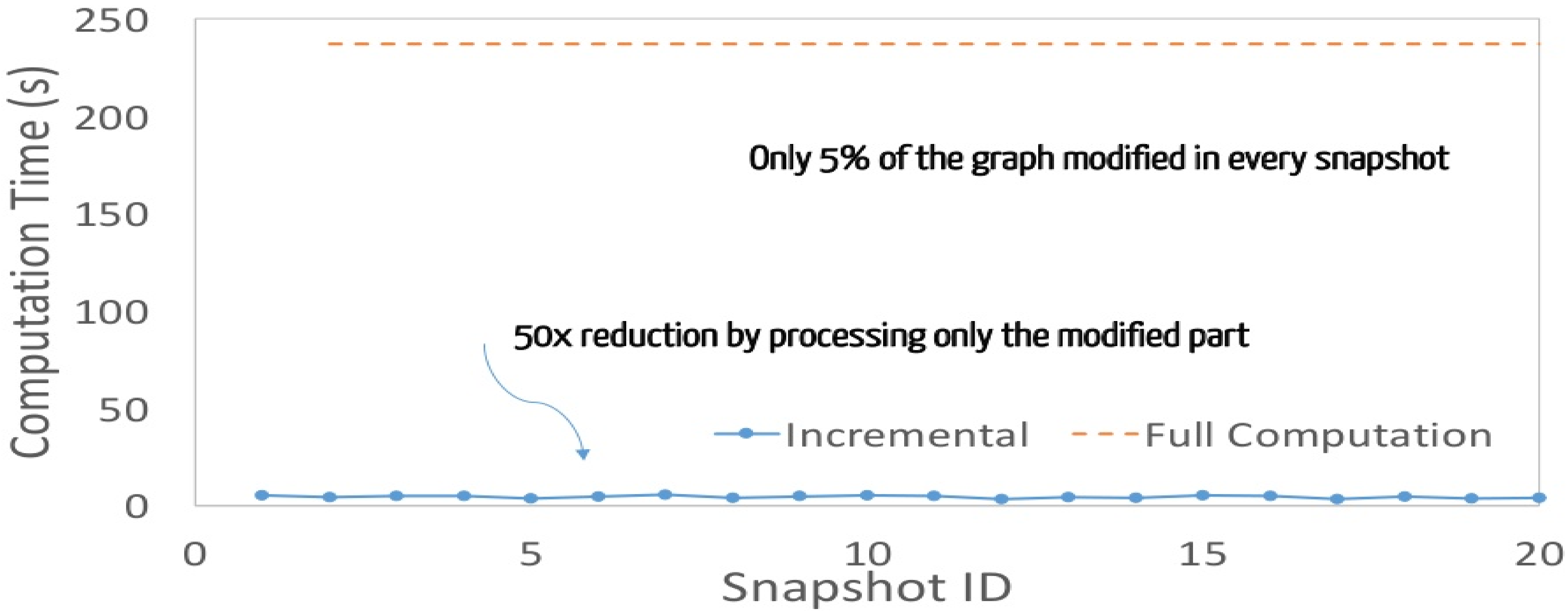
Benefits of Storage Sharing



Benefits of sharing communication



Benefits of Incremental Computing



Ongoing/Future Work

- Tight(er) integration with Catalyst
  - Tungsten improvements
- Code release
- Incremental pattern matching
- Approximate graph analytics
- Geo-distributed graph analytics

Summary

- Processing time-evolving graph efficiently can be useful
- Sharing storage, computation and communication key to efficient time-evolving graph analysis
- We proposed Tegra that implements our ideas

Please talk to us about your interesting use-cases!

api@cs.berkeley.edu

www.cs.berkeley.edu/~api