Graph Queries and Analytics on Evolving Data Graphs

Evaggelia Pitoura

Computer Science and Engineering Department

University of Ioannina, Greece

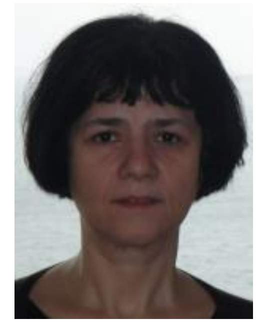
University of Ioannina

Data Lab @ UOI

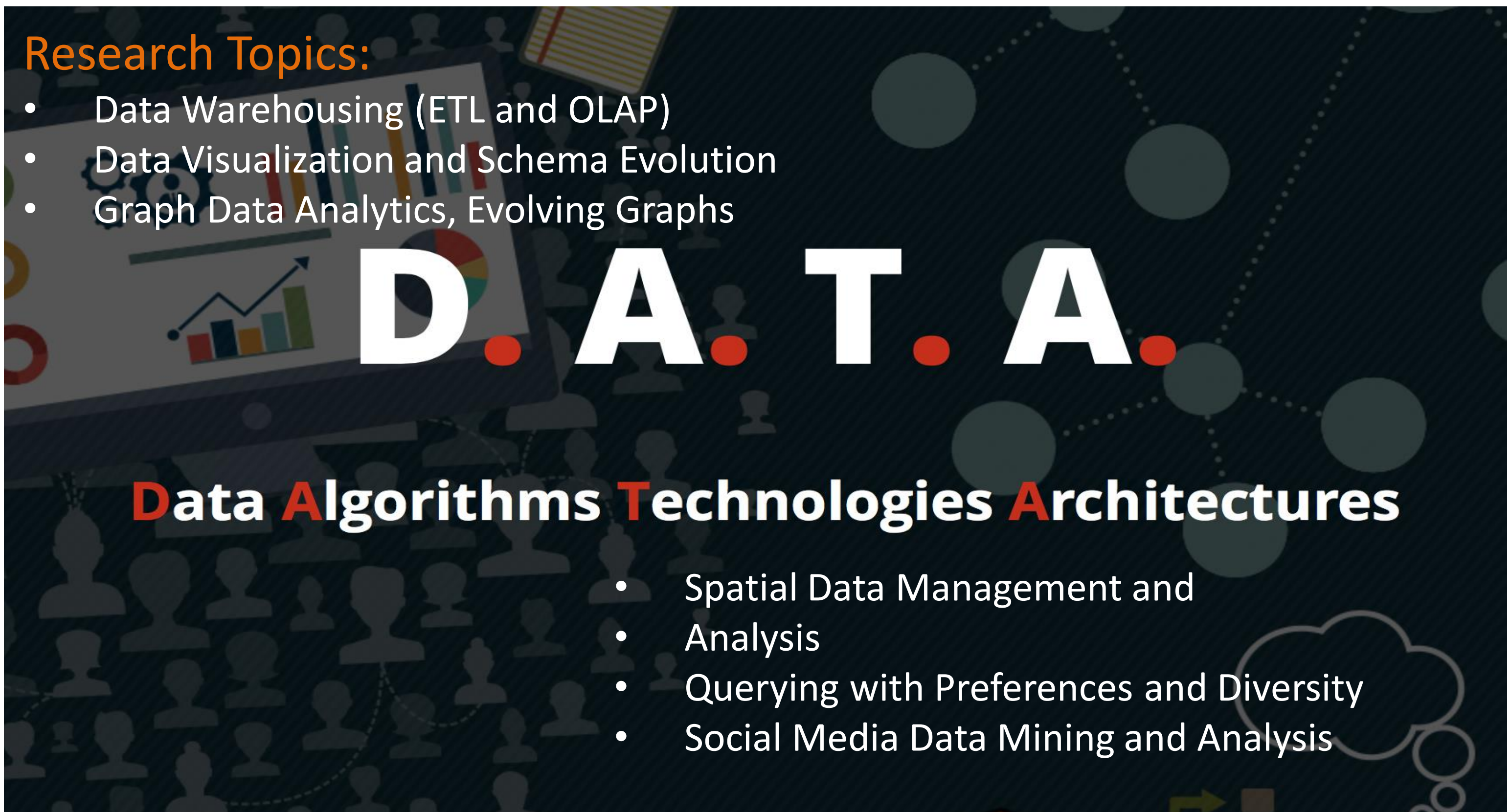

Panos Vassiliadis


Panayiotis Tsaparas


Nikos Mamoulis


Evaggelia Pitoura

…+ 16 students



Research Topics:

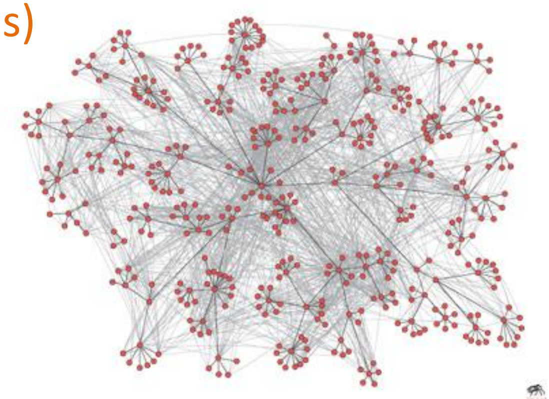
- Data Warehousing (ETL and OLAP)
- Data Visualization and Schema Evolution
- Graph Data Analytics, Evolving Graphs

- Spatial Data Management and
- Analysis
- Querying with Preferences and Diversity
- Social Media Data Mining and Analysis

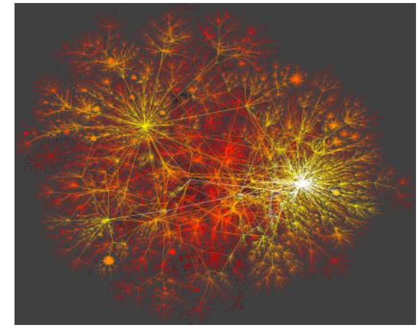
Why Graphs?

Online social networks

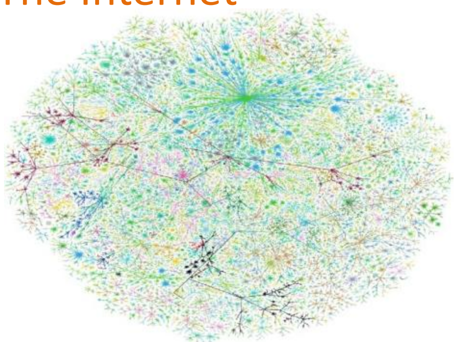
Communication networks (email, phones)



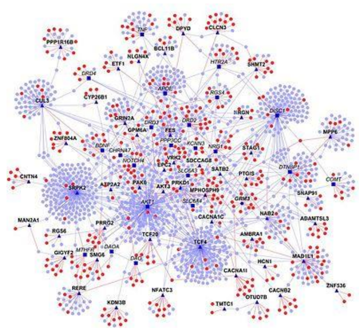
The Web



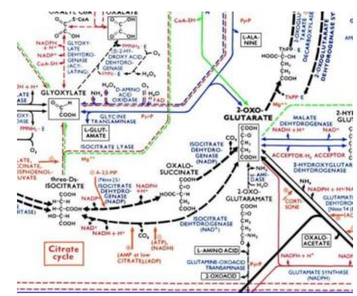
The Internet



Biological networks

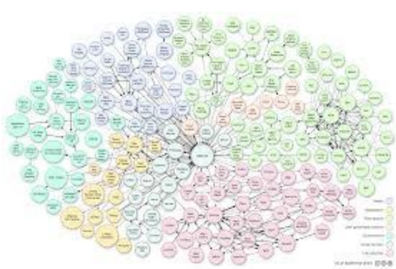


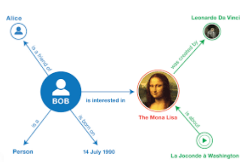
Proteins - interactions



metabolites, enzymes
- chemical reactions

Linked open data, RDF

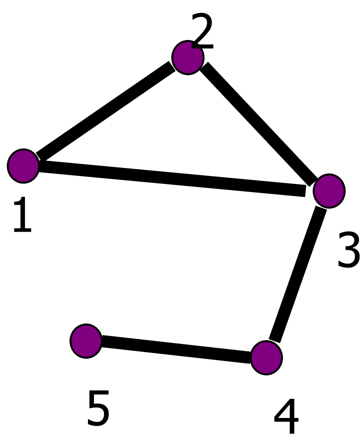




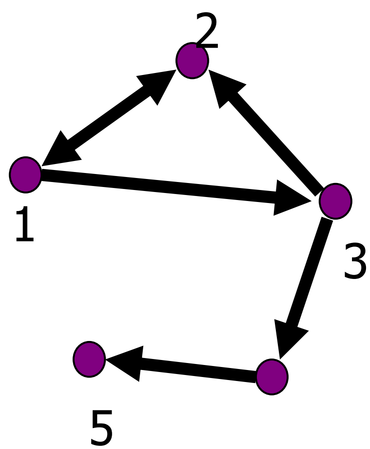
Graph Model (basics)

Graph G=(V,E)

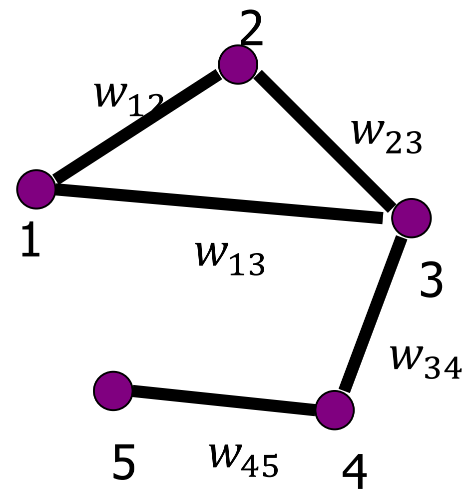
- V = set of vertices (nodes)
- E = set of edges



Undirected graph

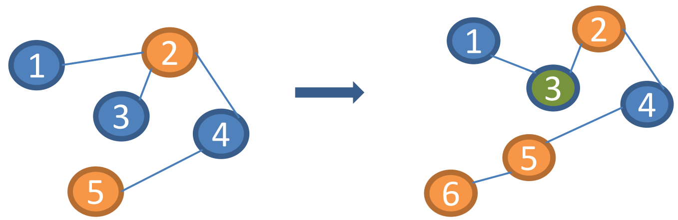


Directed graph



(edge-) Weighted graph
weight: distance/similarity, volume of communication
(node-) weights
Labels or attributes
Properties (key-value pairs)

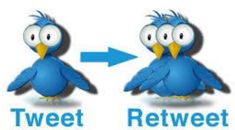
Why Time-Evolving?



Social network

Underlying network

Interaction networks - Who interacts (likes, befriends, reposts, retweets) with whom

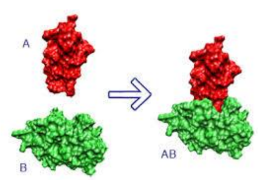


Cooperation network (citation network) Who cooperates with whom

Who talks/communicates with whom

Both

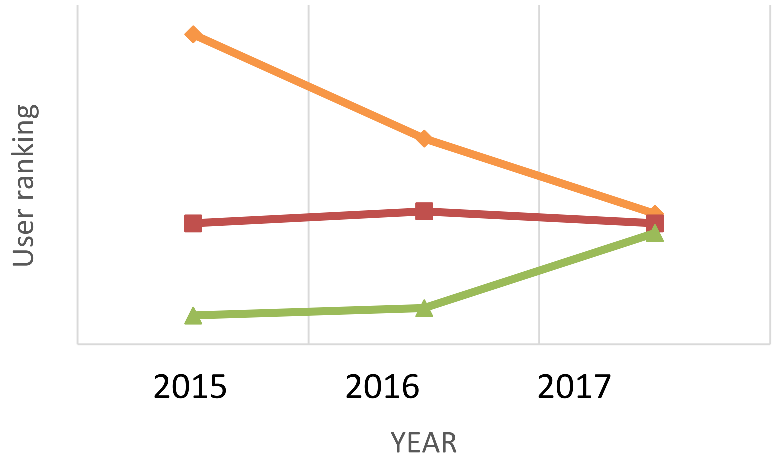
- *Structure* (nodes, edges)
- *Content* (weight, labels, property values)



Protein interactions

Why evolving graphs (simple example)?

We would like to be able to query/analyze the whole history of the graph as the graph evolves – why?



If we look only at 2017, just that the three users are similar

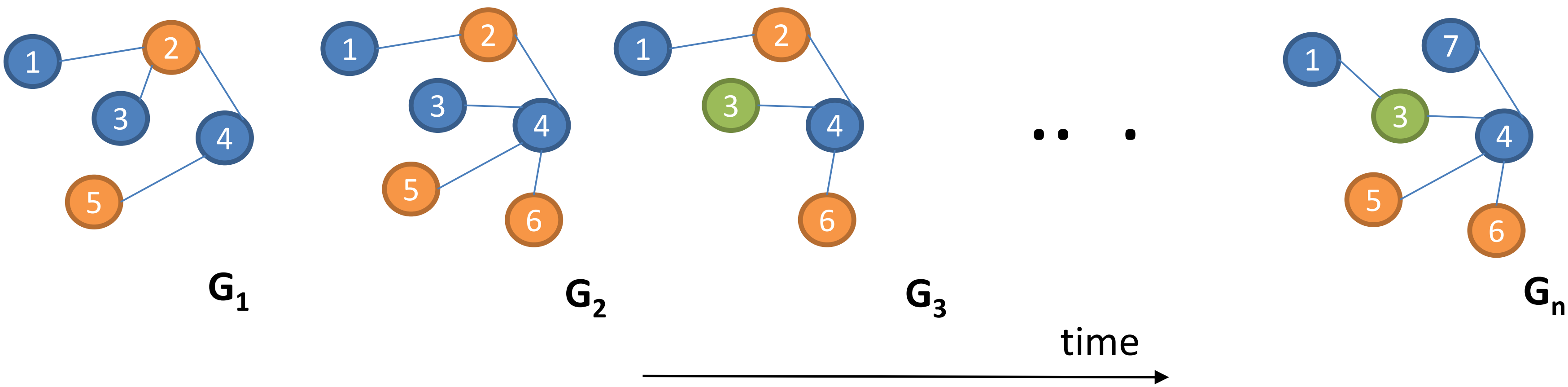
Why evolving graphs?

- Metrics evolve over time
- Knowledge discovery: Understand the network (e.g., social network analysis, biology, etc)
- Useful in predicting the future (link recommendations, marketing, etc)
- Digital forensics (e.g., virus propagation), disease propagation, etc
- Temporal correlations and causality

And of course, recall this morning talk: Not only BIG but also LONG data

Evolving Graph: definition

Time-evolving or *historical graph* is a sequence of graph *snapshots* G_t capturing the state of the graph at time point or instance *t*



Discrete time points correspond to Real time (e.g., minutes)

Granularity (what is the chronon?)
Time (second, minutes, etc) or a new operation happens
Operational (number of operations)

Quiz: Discrete or continuous? Transaction or valid time?

Historical vs Dynamic Graphs

Focus of this talk:

Query/analyze *the full history* of an evolving graph

Dynamic (non static) graphs: Maintain only one snapshot: the current/most recent one
Apply queries *on the most current* snapshot

Example
Given a time-evolving graph, (page)-rank query

- Calculate each vertex's current PageRank (dynamic)

vs

- Analyze the change of each vertex's PageRank for a given time range (historical)

Historical vs Dynamic Graphs

In dynamic graphs

Real-time evaluation (metrics, queries) so that they reflect the current state (efficiency)

Avoid re-computation and support incremental evaluation and update of any data structures

Special cases of dynamic graphs

- Graph streams
  - Graph updates arrive in a streaming fashion
  - *Continuous* evaluation
  - Additional issues
    - Limited memory storage for the updates (cannot store the whole stream)
    - Incremental update of the result
- Online graphs
  - we do not know the whole graph at each time point, but need to probe

Outline

Introduction, problem definition

➡ Taxonomy of historical queries

Part 1 (general techniques)

- Representation, Storage, Processing

Part 2

- Specific Types of Analysis and Queries

Conclusions and Future Work

Graph processing

No standard query language, or analysis

- Offline graph analytics (graph mining)
  - Centrality measures (PageRank, betweenness, etc)
  - Triangle counting, cliques, cores, density
  - Diameter
  - Clustering, community detection
  - Frequent patterns, or motives

- Online query processing
  - Traversals
    - Reachability, shortest, paths,
  - Graph pattern matching
  - …

Graph processing in historical graphs: taxonomy

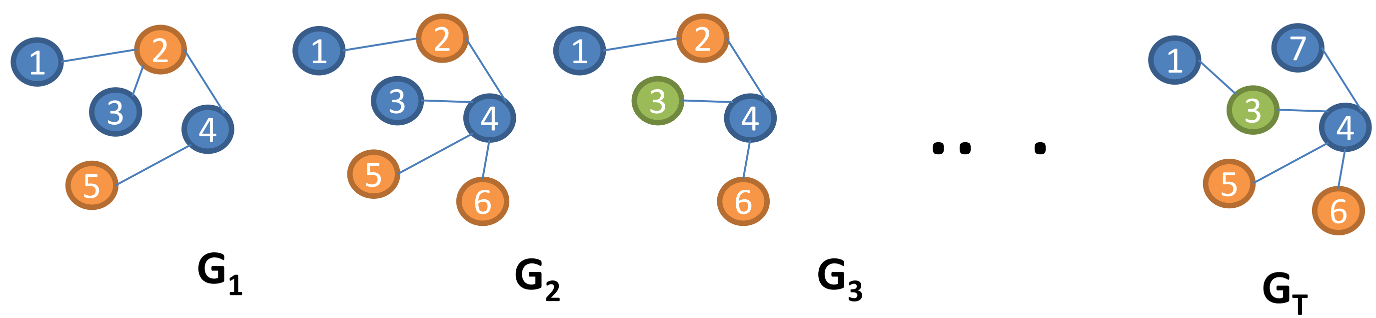
- historical
- durable
- evolution

Graph processing in historical graphs

Historical graph processing: Typical graph query (or, analysis) *Q* applied in some time interval *I* in the past (time travel)

- Single point or interval (time slice) or a time expression (every Sunday)

Example: Pagerank in t_1, Shortest path distance (or, paths) between *node1* and *node3* in [1, 3], Matches of a given pattern in [1, 3]



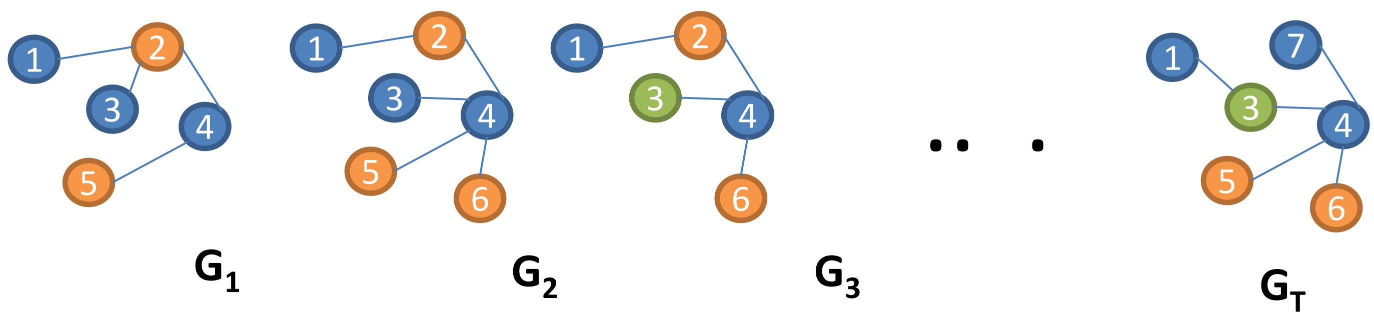
Aggregation semantics when more than one time instance

- Reachability: At all instances, at least one instance, at least-*k*
- Shortest path: the shortest among the paths that exist in (all, one, at least k)? Or, the shortest path may be different at each instance
- Distance: as before, but also, average?

Graph processing in historical graphs

Persistence or durability graph processing: The most persistent results of *Q* in a time interval *I* in the past (that is, the result that appears in the largest number of instances)

Example: The most durable shortest path between *node1* and *node3* in [1, 3]
The most durable match, that is the subgraph that matches input pattern *P* at the largest number of instances in [10, 30]



Semantics

- Contiguous and non-contiguous

Variations

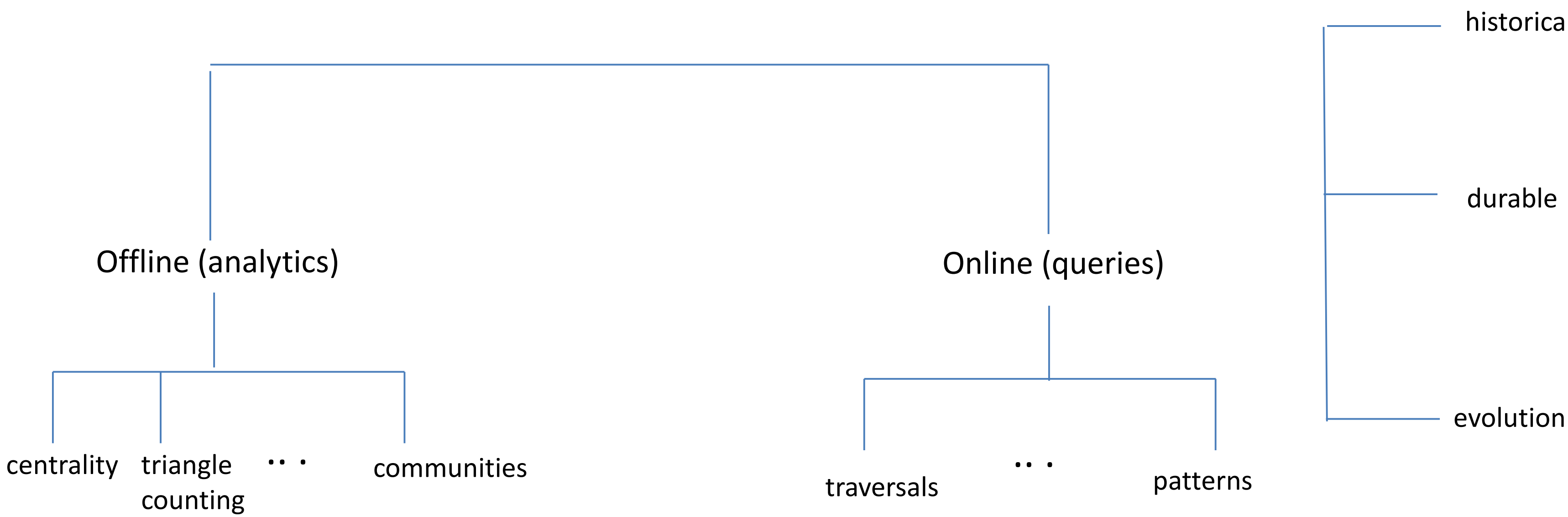
- Top-k most durable
- Results that appear in at least-k instances (to avoid transient results, or, even noise)

Graph processing in historical graphs

Ad-hoc evolution queries

- What is the *first time* that *X* happened (the first time that u and v connected)
- The *maximum time interval* for *X*
- *How many times X* happened
- Patterns of evolution: *What/how* much *X* changed
- Peaks, intensity, etc
- Results similar in evolution

Summary



- All combinations are possible with varying semantics

Example

Find the (twitter) users that liked posts of X and Y in [2009, 2017]

Historical: apply query in past intervals and combine the results

Durable: report the most durable result (not same as all (since all may be empty)

Ad hoc-evolution (how the pattern change over time -> various plots?)

Generality is hard

- There is no single model of large graphs
- There is no single query (declarative) language or API for processing large graphs
- There is no single system for processing large graphs (analysis: GraphX, Giraph, etc, databases: Neo4j, Sparksee, Titan, etc, in memory ad-hoc algorithms)

Outline

Introduction, problem definition

Taxonomy of historical queries

➡ Part 1 (general techniques)

- Representation, Storage, Processing

Part 2

- Specific Types of Analysis and Queries

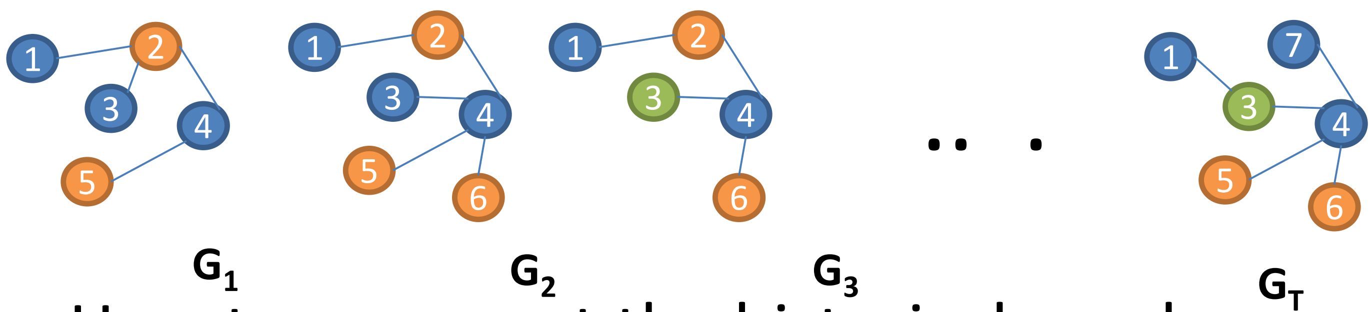
Conclusions and Future Work

Part 1

Representation, Storage, Processing

Representation

Given a historical graph (graph sequence):

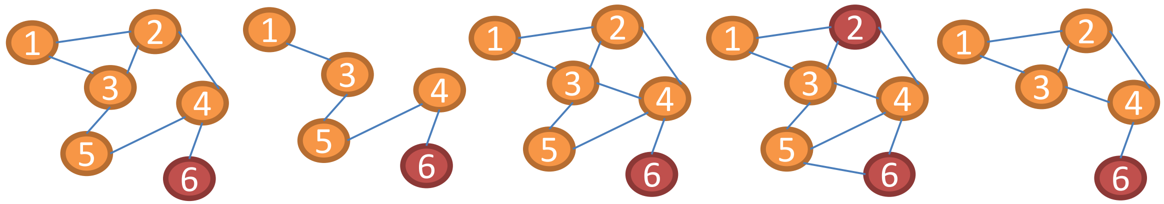


- How to represent the historical graph
- Store
  - On disc or, in memory
  - Partition, or distribute the historical graph
- Processing approaches

Note: only nodes and edges – but also, weights, labels, properties

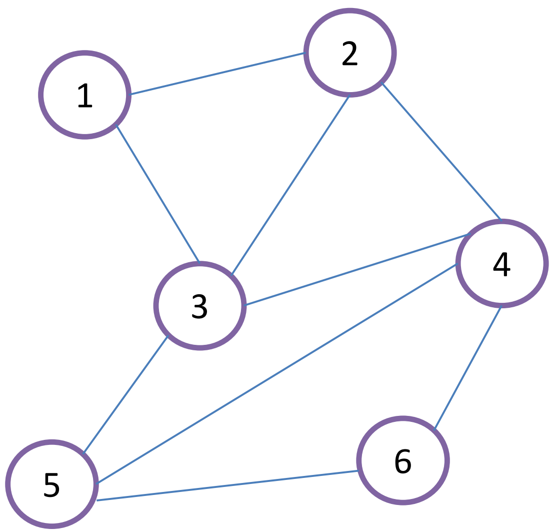
First, *two useful aggregated graphs*

Union Graph



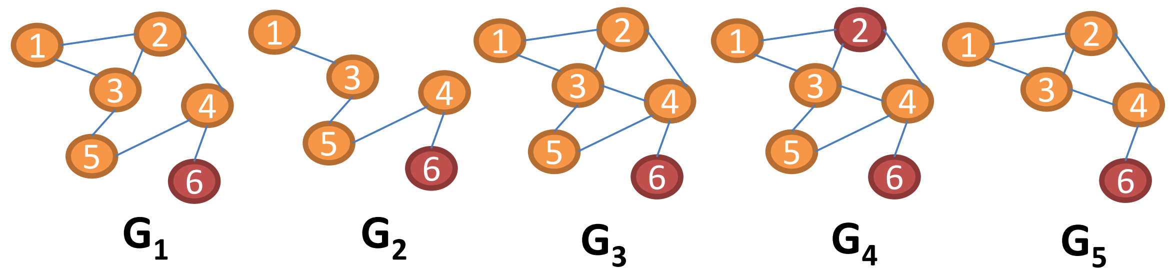
G_1 G_2 G_3 G_4 G_5

G_U

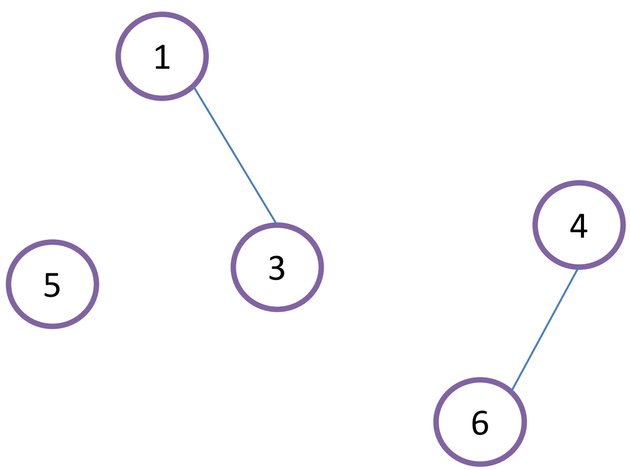


- An element belongs to the union graph, if it belongs to any of the snapshots
- Time information is lost

Intersection Graph

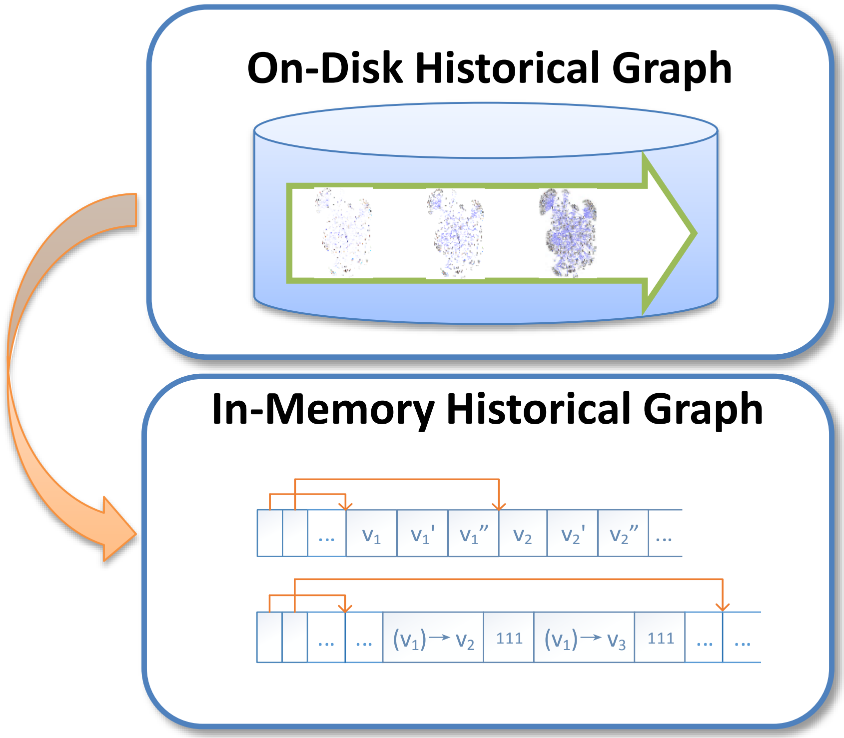


$G_\cap$



- An element belongs to the intersection graph, if it belongs to all snapshots
- Transient elements are lost

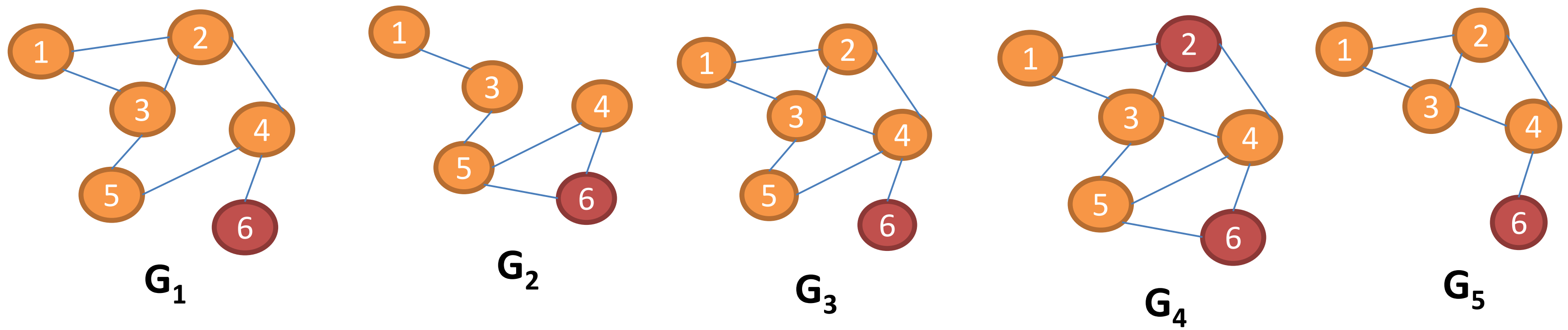
Overview: on disk or in memory



All snapshots in
- Files
- DBMS (relational or graph database)

Selected snapshots

Copy and Log representation



Two straw man approaches

- COPY: Store every snapshot (G_1, … G_5)
- LOG: Store only operations – delete-node(2), delete-edge(2, 1), delete-edge(2, 3), add-edge(5, 6) – snapshot3: add-node(1, 2), etc

Tradeoffs: redundant storage vs performance time

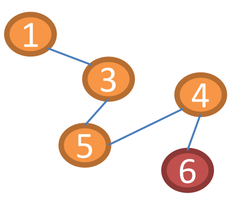
Hybrid representation: deltas

Store:
(1) selected graph snapshots
(2) operational deltas (logs) Δ from selected snapshots

- To create any snapshot G_t: apply deltas on other materialized snapshots

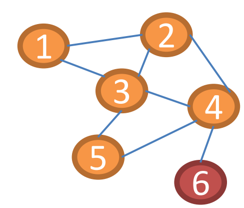
materialized G_2

delta log



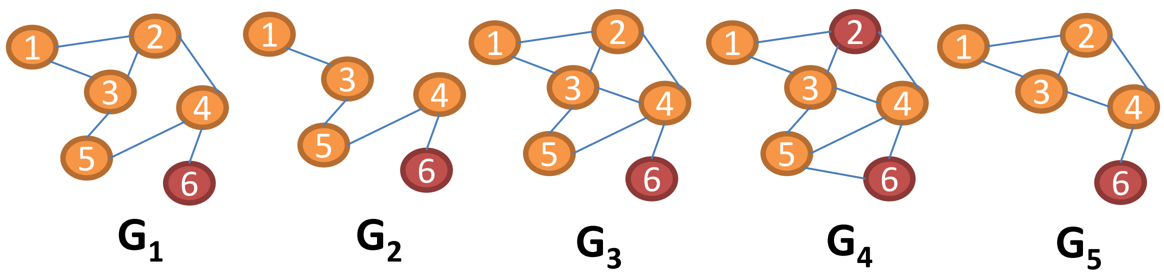
G_2

Δ = add(node(2)) add(edge(2,1)), add(edge(2,3)) add(edge(2,4)), add(edge(3,4))

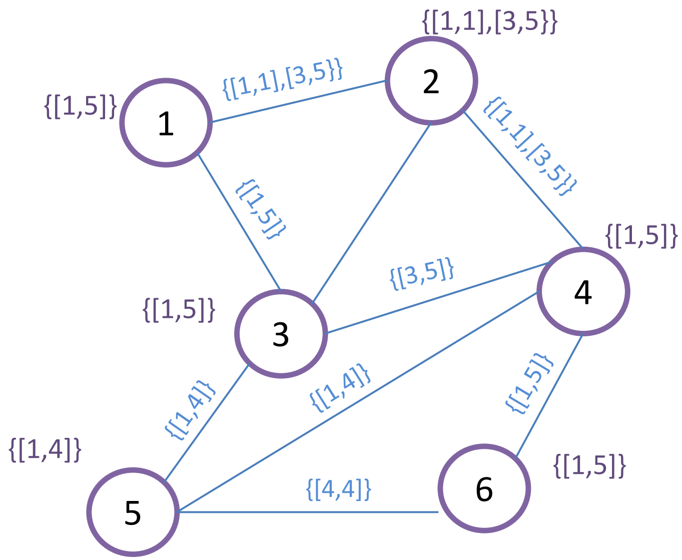


G_3

Hybrid: Versioning

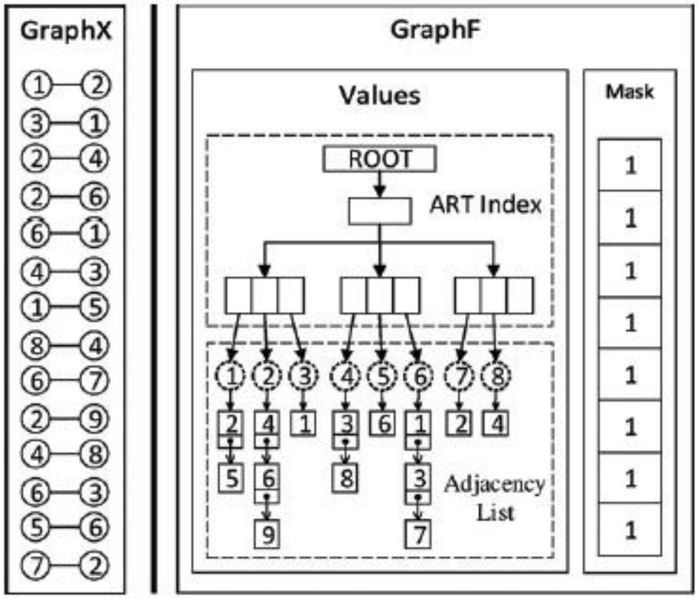


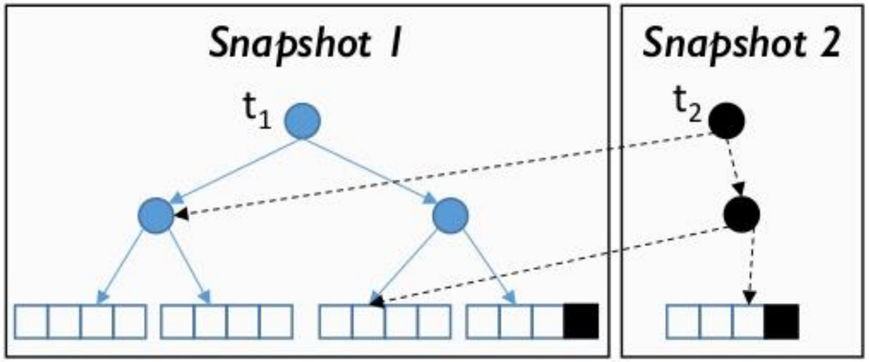
VG



- Keep the union graph
- Each graph element is annotated with its *lifetime* (lifespan)
- *Sets of intervals (Quiz: how is this called?)* to allow the deletion and the re-insertion of an element
- A version graph for all, *or subsets* of the sequence e.g., one for G_1, G_2 and one for G_3, G_4, G_5

Hybrid: Indexing [SIAMCSE17]

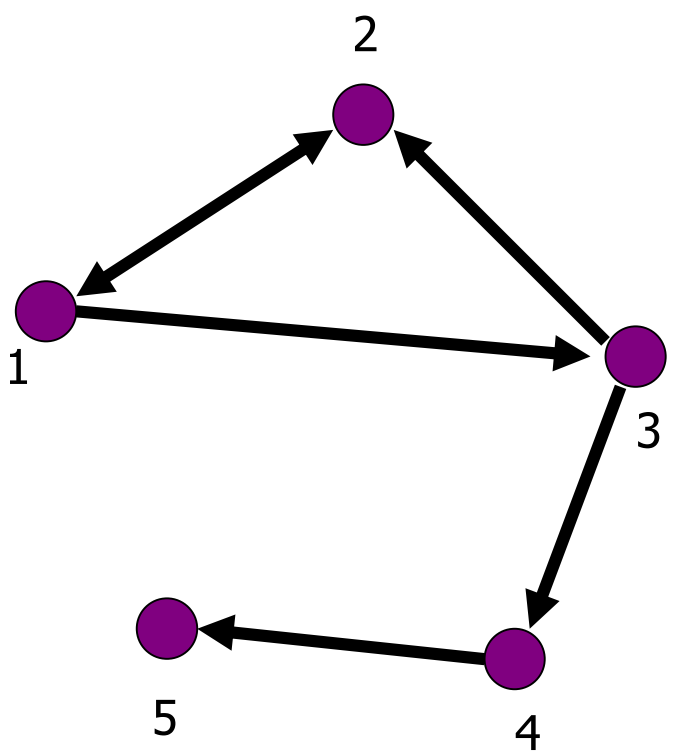




Persistent adaptive radix tree

(Static) Graph Representation

- Adjacency Matrix
  - unsymmetric matrix for undirected graphs

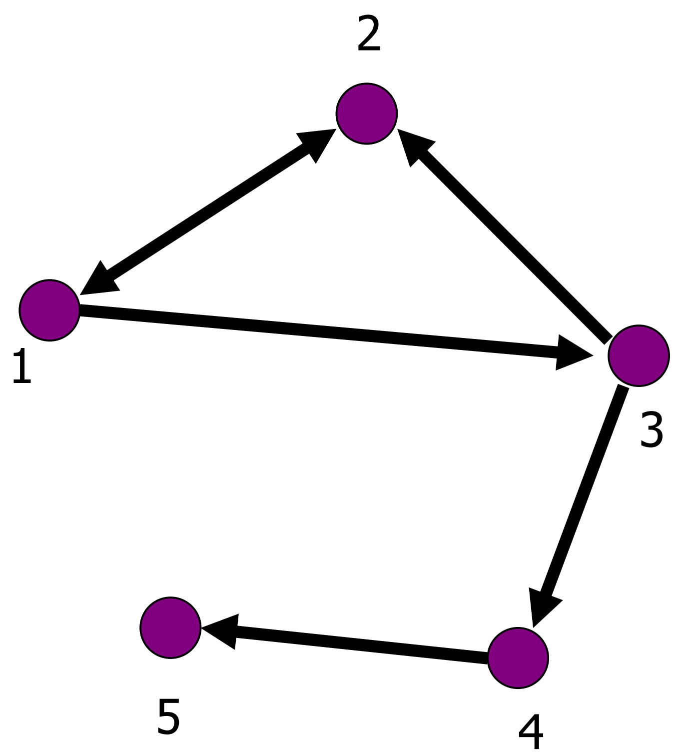


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Various compression techniques

(Static) Graph Representation

- Adjacency List
  - For each node keep a list of the nodes it points to

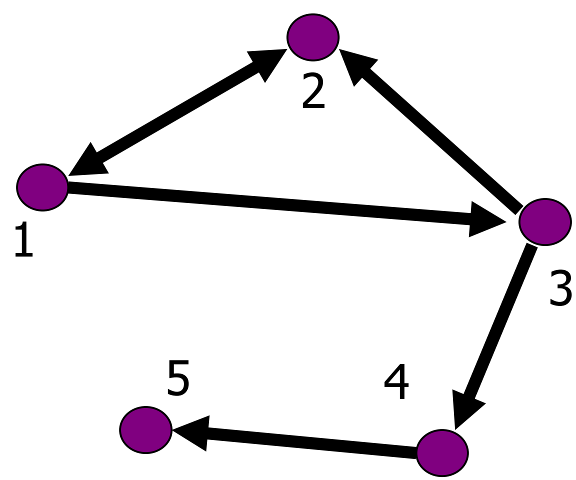


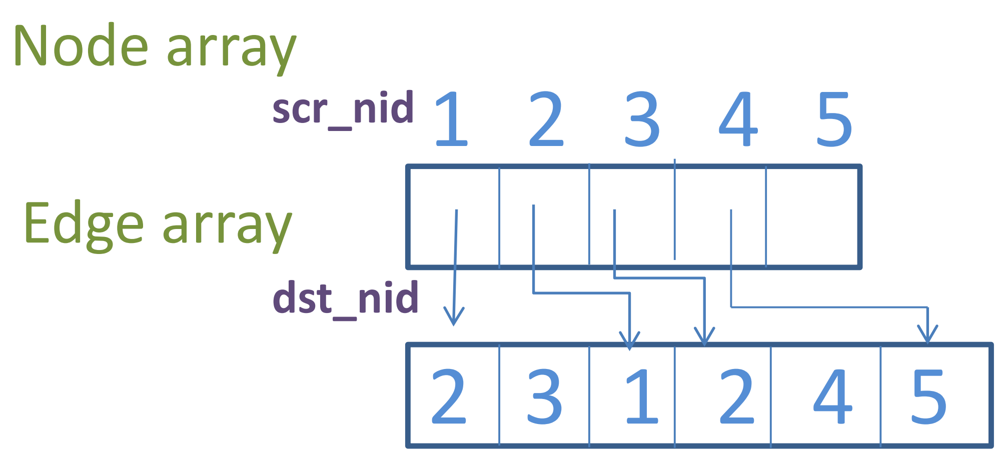
1: [2, 3]
2: [1]
3: [2, 4]
4: [5]
5: [null]

Common in-memory

(Static) Graph Representation

- Compressed Sparse Row (CSR) format
  - Keep nodes and edges in separate arrays with array indexed correspondingly to the node id
  - Node array stores offsets into the edge array (first edge)
  - Edge array sorted first by source of each edge then by destination

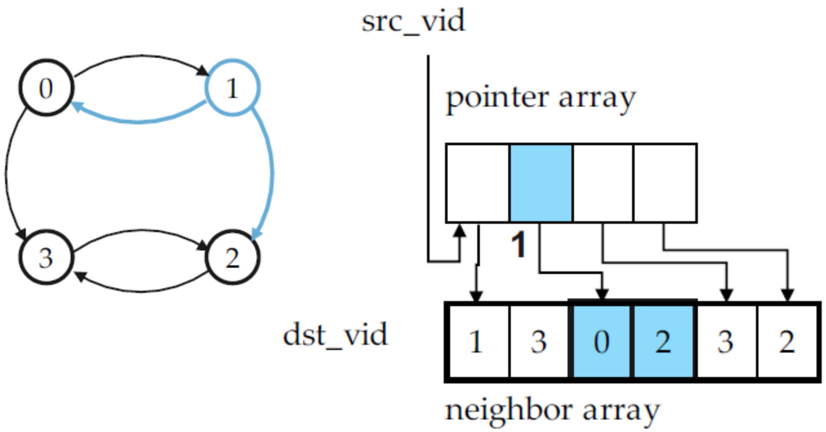


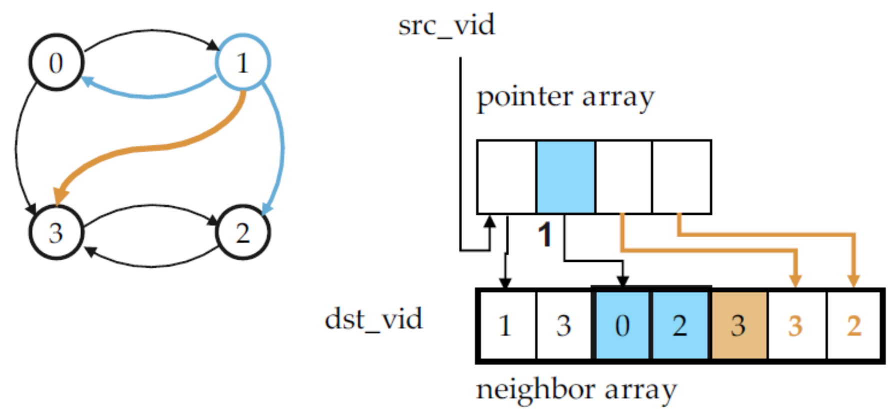


- In memory -- Minimizes memory use to O(n + m)

(Static) Graph Representation

- Compressed Sparse Row (CSR) format (mutability)

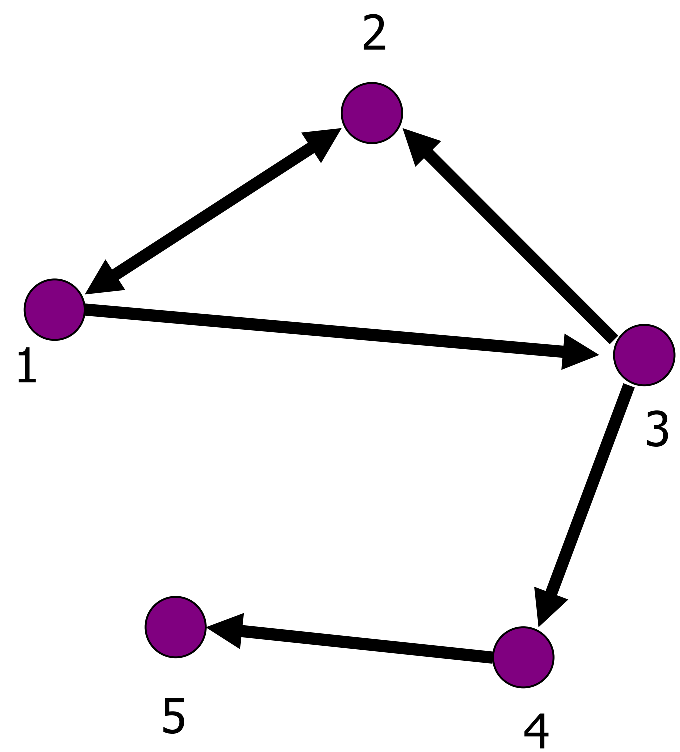




memory

(Static) Graph Representation

- List of Edges
  - Keep a list of all the directed edges in the graph

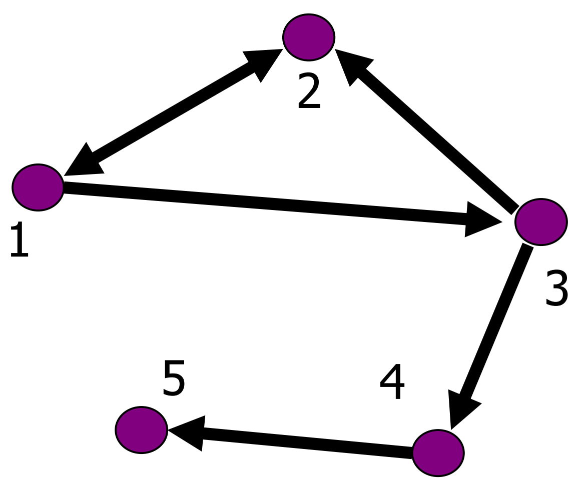


(1,2)
(2,1)
(1,3)
(3,2)
(3,4)
(4,5)

Common in disk (files)

(Static) Graph Representation

- Relational database
  - A vertex and an edge table



Disk storage

Vertex Table

id	name	value	…
1	N1		
2	N2		
3	N3		
4	N4		
5	N5		

Edge Table

src	dst
1	2
1	3
2	1
3	3
3	4
4	5

Or a separate table with vertex and edge properties

Path from 1 to 4?

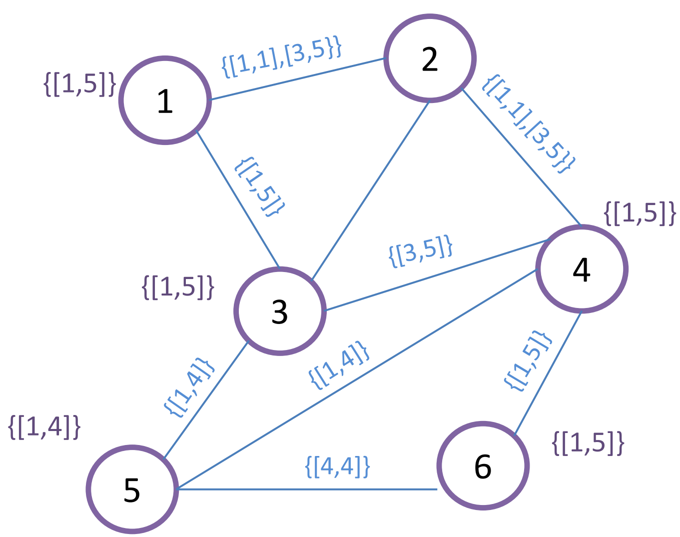
Dynamic Graph Representation

COPY approach: one static graph representation for each snapshot

LOG/Delta approach: static graph representation for select snapshots – ***special structures for the deltas***

Versioning approach: Extend the structures to "code" the lifespan of each element

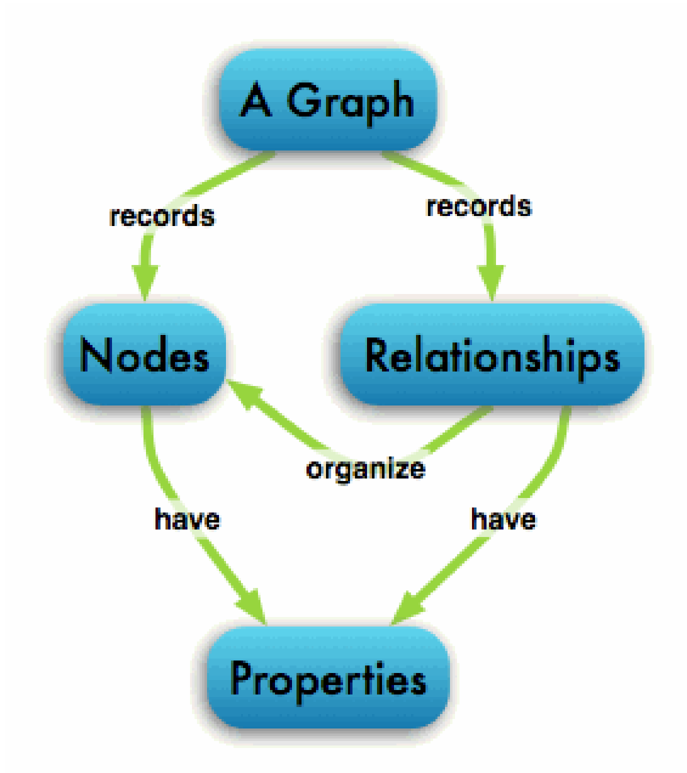
VG



Edge Table

src	dst	lifespan
1	2	
1	3	
2	1	
3	3	
3	4	
4	5	

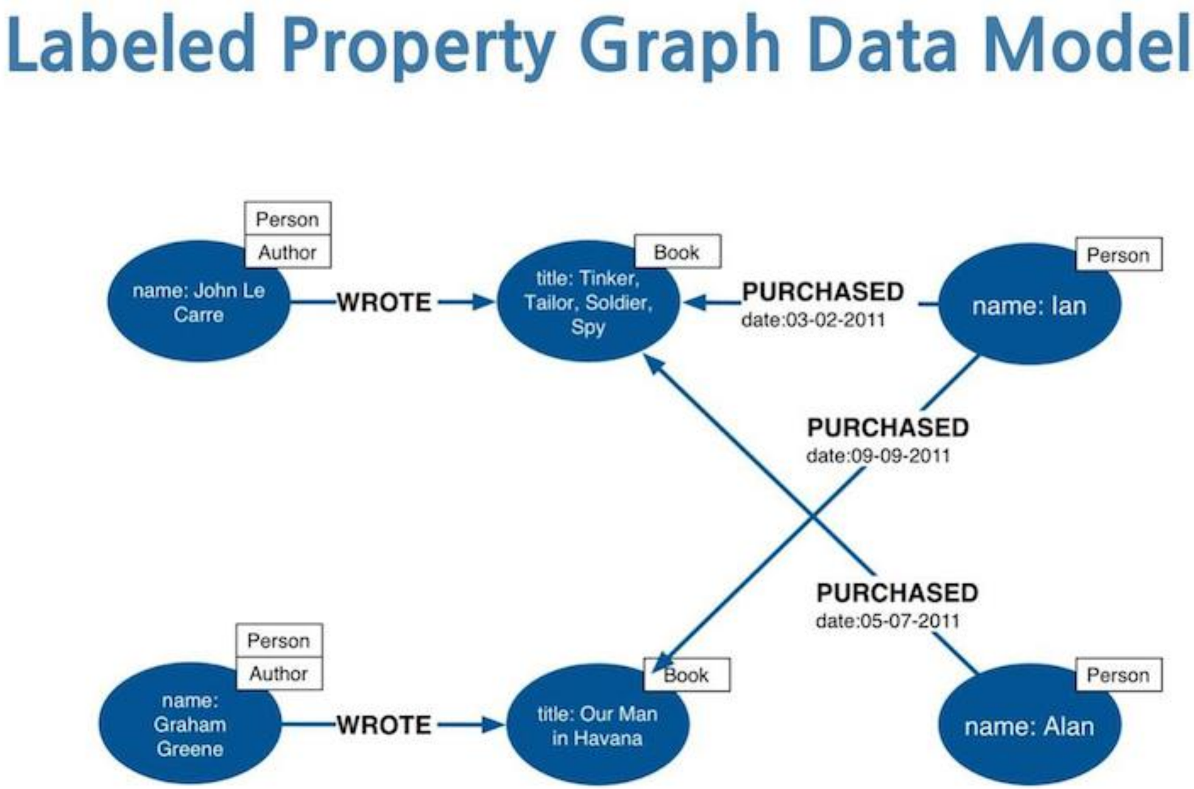
(Static) Property Graph Model (native graph database)



In disk -- separate stores for nodes, relationships, properties

Example from Neo4j

(Static) Property Graph Model (native graph database)



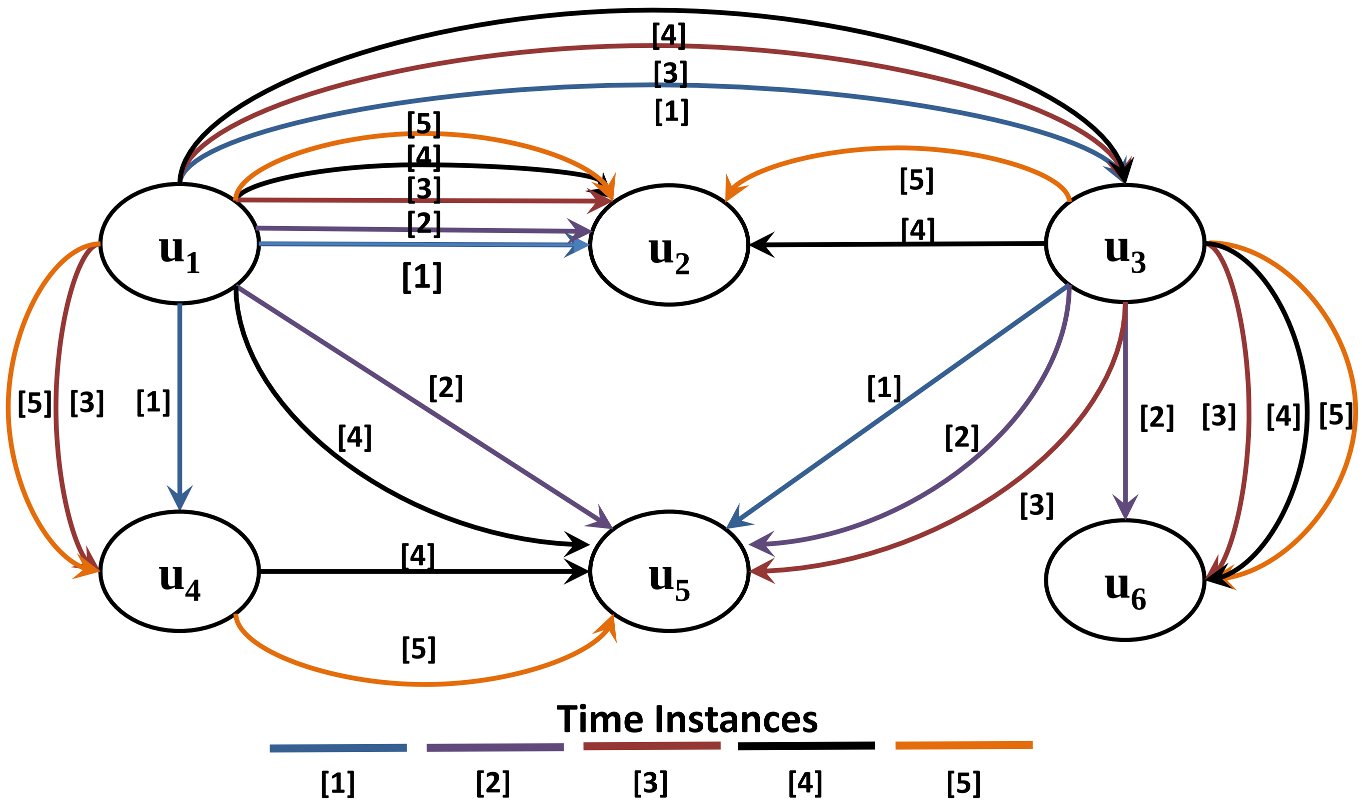
Example from Neo4j

Graph database (historical)

Store information about the snapshots [ADBIS17]

Multi-edge
Single-edge

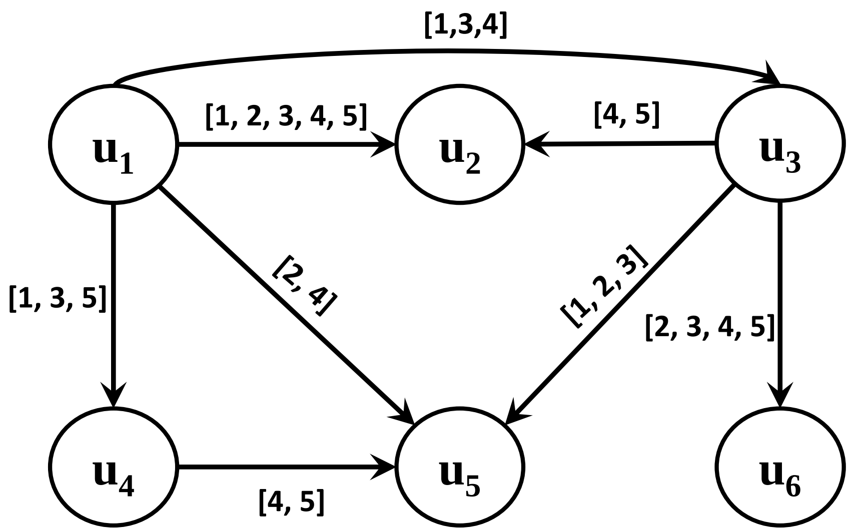
Graph database: Multi-edge



- A *different edge type (label)* between two nodes u and v for each time instance of the lifespan of the edge ($u \ -> \ v$).
- Provides an efficient way of retrieving the graph snapshot G_t corresponding to time instance t.

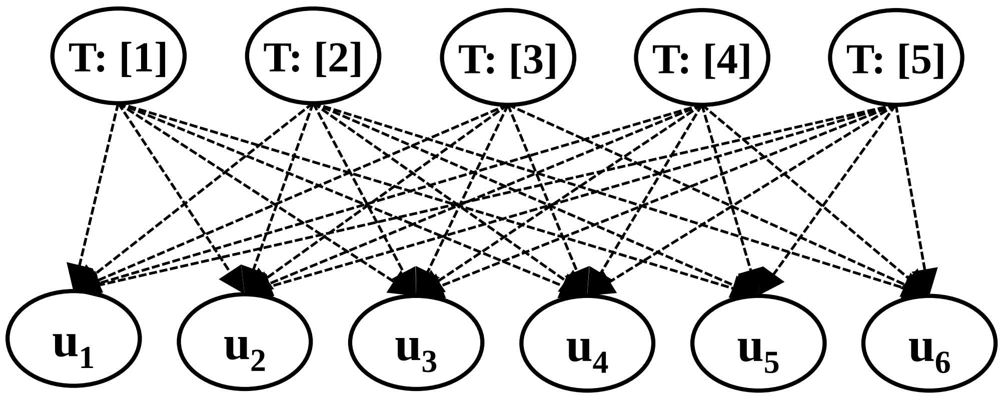
Graph database: Single-edge

SETP



- A *single edge*
- Lifespan as a property of the edge
- How to represent lifespans? (e.g.,, list of timepoints)
- Storage efficient, but may slow-down traversals

Graph database: Index



Time index

- Nodes of type of T, where each node of the given type has a *unique value* that corresponds to a specific time instance.
- Add edge to alive nodes

Next: Processing

So far,
Different ways to store a graph (in files, databases, main memory)
Adapt them for historical graphs

Now,
generic ways to do processing , mainly historical (or, time travel) queries

Processing

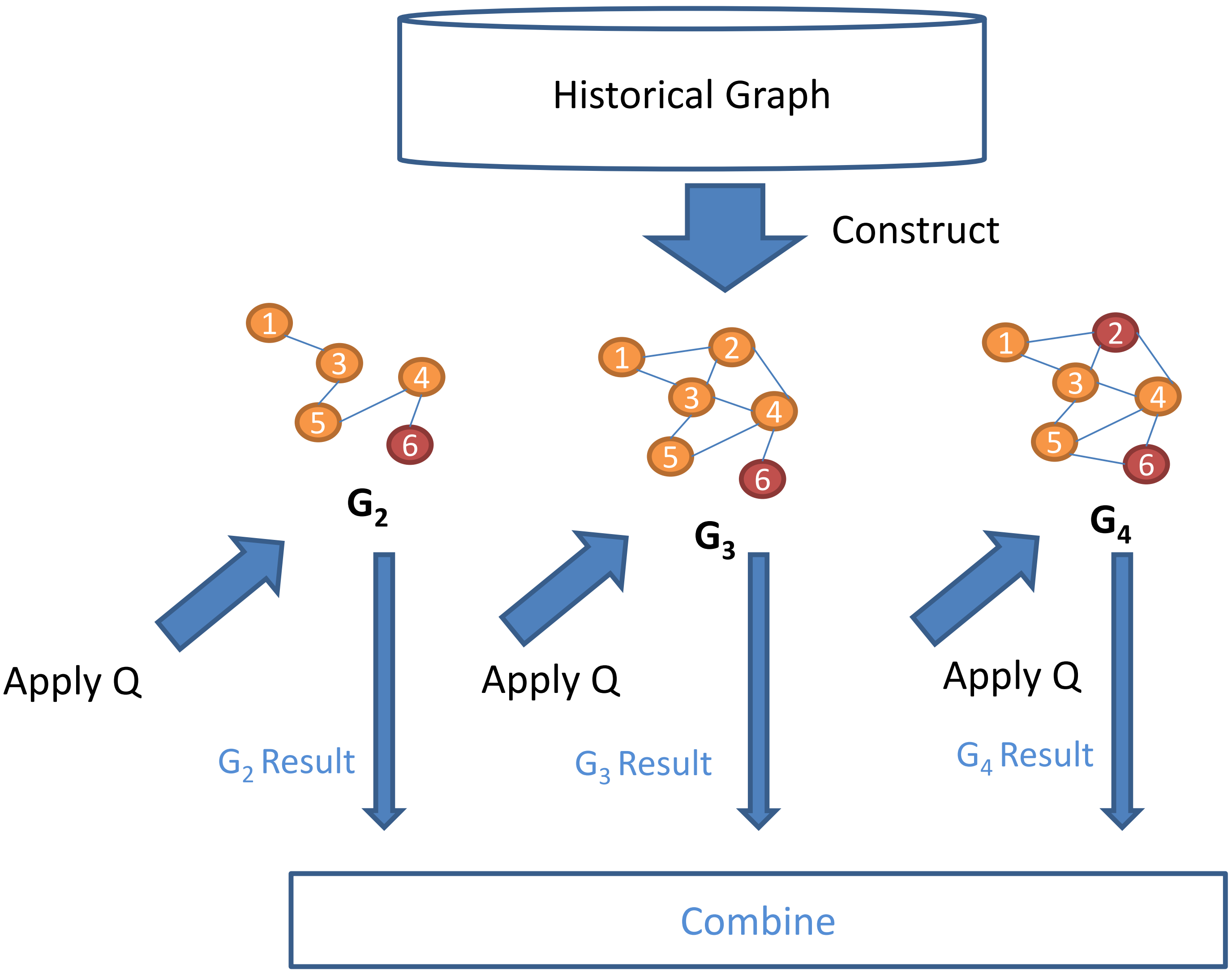
Simple 2-Level Strategy

1. Construct the required snapshots (e.g., apply the deltas, or (use a time-index to) project from the version graph the live elements)
2. Apply best known static algorithms at each snapshot
3. (optional) Combine the results

Example of this approach: Delta Graph

2P Processing

Q in [2, 4]



Delta Graph [ICDE13, EDBT16]

Scope: historical queries

2-level: Access past snapshots of the graph and perform static graph analysis on these snapshots

Focus: compact storage and efficient retrieval of snapshots

Hybrid Approach

- Materialize selected snapshots
- Maintain Eventlists: log of events (insert, deletes, etc)

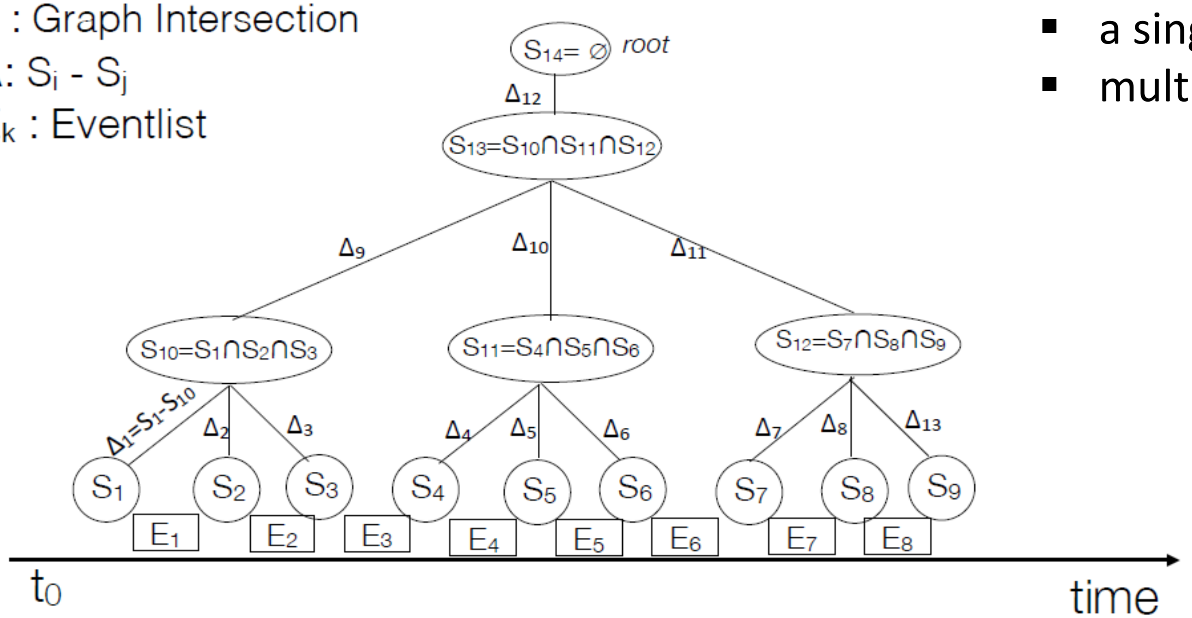
Two main components

- Temporal Graph Index: Delta Graph
- Graph Pool: in memory data structure

Delta Graph: Index

Leaves: snapshots (not necessarily materialized), bidirectional leaf event lists
Internal nodes: graphs constructed by combining the lower level graphs (not necessarily corresponding to any actual snapshot)
Edge deltas: information for reconstructing the parent node for the child node

S_i : Graph Snapshot
$\cap$: Graph Intersection
Δ: S_i - S_j
E_k : Eventlist

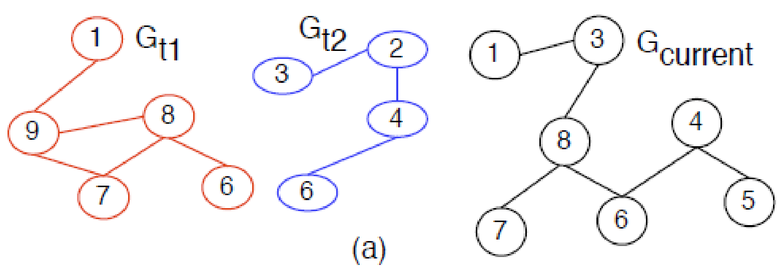


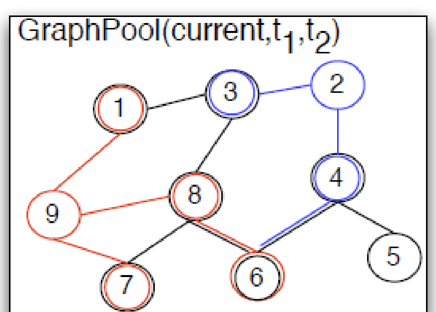
Used to construct:

- a single snapshot
- multiple snapshots

Delta Graph: Graph Pool

- In memory data structure
- Union of
  - the *current graph* reflecting the current state
  - the *historical snapshots*, retrieved from the past
  - *materialized graphs* corresponding to internal or leaf nodes of the Delta Graph
- Each element associated with a bitmap indicating which of the active graphs include the element





GraphId-Bit Mapping Table

Bit	GraphID	Graph	Dep
2,3	34	Hist. Graph	-
4	4	Mat. Graph	-
5	41	Mat. Graph	-
6,7	35	Hist. Graph	4

2P Processing: Extensions

- Targeted reconstruction (partial views)
- No Reconstruction

Targeted reconstruction: Partial Views
[GRADES13]

- Snapshot construction is expensive
- Many queries refer to only part of the graph
- Restrict snapshots reconstruction around a specific node (partial views)

Local queries or node-centric queries

Traverse only a specific subgraph of G

Examples: Queries similar to Facebook graph search

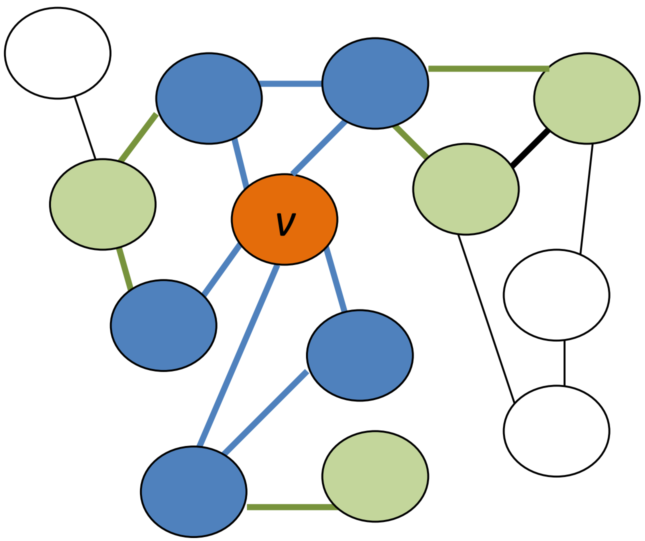
- Find my friends that live in Brussels
- Find the friends of my friends that are interested in graph management, etc…

Partial Views

- Partial views modeled as *extended egonets*
- Egonet(v, R, t)
  - Node *v* center of the egonet
  - *R* radius of the induced subgraph
  - *t* time point at which the egonet is valid (i.e. egonet is a subgraph of SG_t)

Radius extension

- Egonet of v with R=1
- Egonet of v with R=2



Time extension

Partial Views

- Model *local queries* as *egonets* similar to partial views
- Given a query *Q*, construct the partial view required by the query (not the whole snapshot)
  - view construction: for example, apply only the *related* parts of the log file
- Evaluate the query on the derived partial view

Partial Views: Can we reuse materialized views?

- View subsumption between partial views:

Given two partial views, EG_1 and EG_2, EG_1 *subsumes* EG_2, if the result of the evaluation of any local query Q on EG_2 is equal to the result of evaluating Q on EG_1.

- View selection

Given a query workload W, an estimation of the construction cost, a storage budget C

Select a set S of egonets, size(S) < C, to materialize

Such that the total evaluation cost of the query workload W is minimized.

Partial Views: View selection

- Group egonets according to their center
- At each iteration
  - For each group
    - Select the egonet with the *largest construction cost*
    - Re-evaluate the total construction cost of the group
    - Compute the benefit from materializing the egonet
  - Select the *group with the largest benefit*
  - Update all costs
  - Proceed to next iteration until storage limit is met

No snapshot reconstruction [WOS12]

- *Delta-only* query plan
  - The query is evaluated directly on the delta
- *Hybrid query* plan
  - Use the delta and the current snapshot

Is this possible?

Yes, for specific type of queries

No snapshot reconstruction

Time \ Graph		Local	Global
Point		the degree of u_i at t_k	the diameter of G at t_k
Interval	Evolution	how much the degree of u_i changed in $[t_k, t_l]$	how much the diameter of G changed in $[t_k, t_l]$
	Historical (Aggregate)	average degree of u_i in $[t_k, t_l]$	average diameter of G in $[t_k, t_l]$

Query Types		**Query Plans**		
		Two-Phase	Delta only	Hybrid
Point	Local	✓		✓
	Global	✓		
Interval evolution	Local	✓	✓	✓
	Global	✓		
Interval aggregate	Local	✓		✓
	Global	✓		

Processing

Can we avoid running the same algorithm *to all* snapshots?

Idea: apply the algorithm to *representative* snapshots

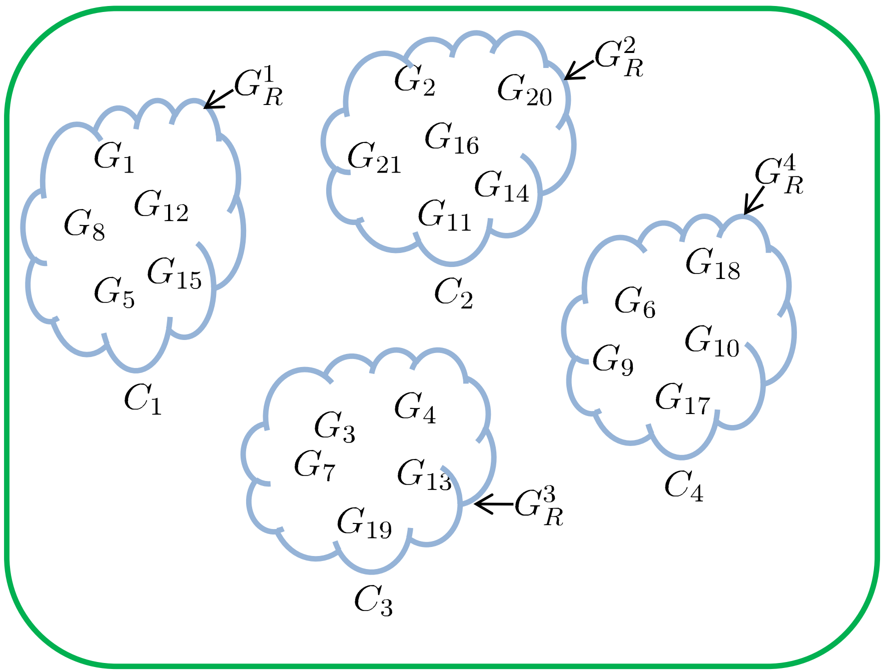
Find-Verify-and-Fix [VLDB11, IS17]

Find-Verify-and-Fix (FVF) Processing Framework

1. Preprocessing
    - cluster similar snapshots
    - extract representatives from each cluster
2. Apply query to each representative *(find)*
3. For each graph snapshot G^t, *verify* the solution
4. If not verified, apply query on G^t *(fix)*

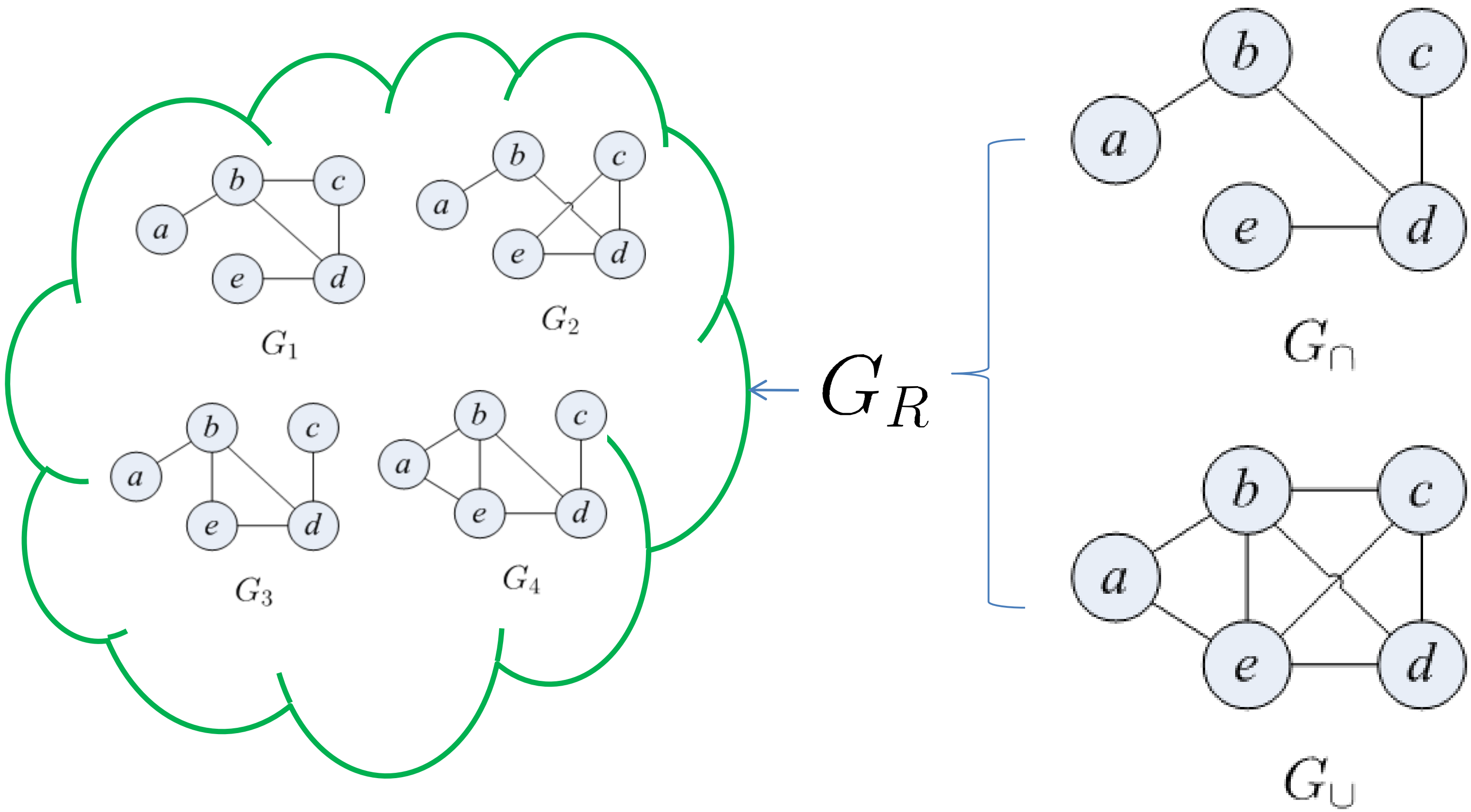
Find-Verify-and-Fix: Preprocessing

Graphs gradually evolving, many edges in common
Exploit graph redundancy by clustering

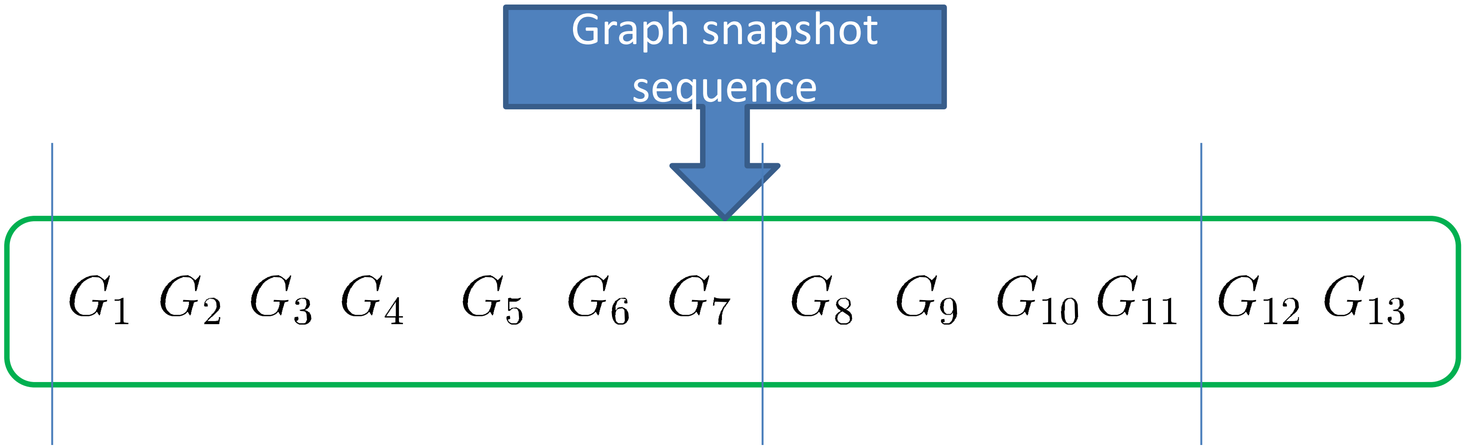


Find-Verify-and-Fix: Preprocessing

For each cluster maintain two representatives: $G_\cap$ and $G_\cup$



Find-Verify-and-Fix: Preprocessing



Segmentation clustering algorithm:

- A cluster consists of *successive* snapshots
- A cluster satisfies:

$$ges(G_{\cap}, G_{\cup}) \geq \alpha \quad \left(ges(G_a, G_b) = \frac{2|E(G_a \cap G_b)|}{|E(G_a)| + |E(G_b)|}\right)$$

Find-Verify-and-Fix: Preprocessing

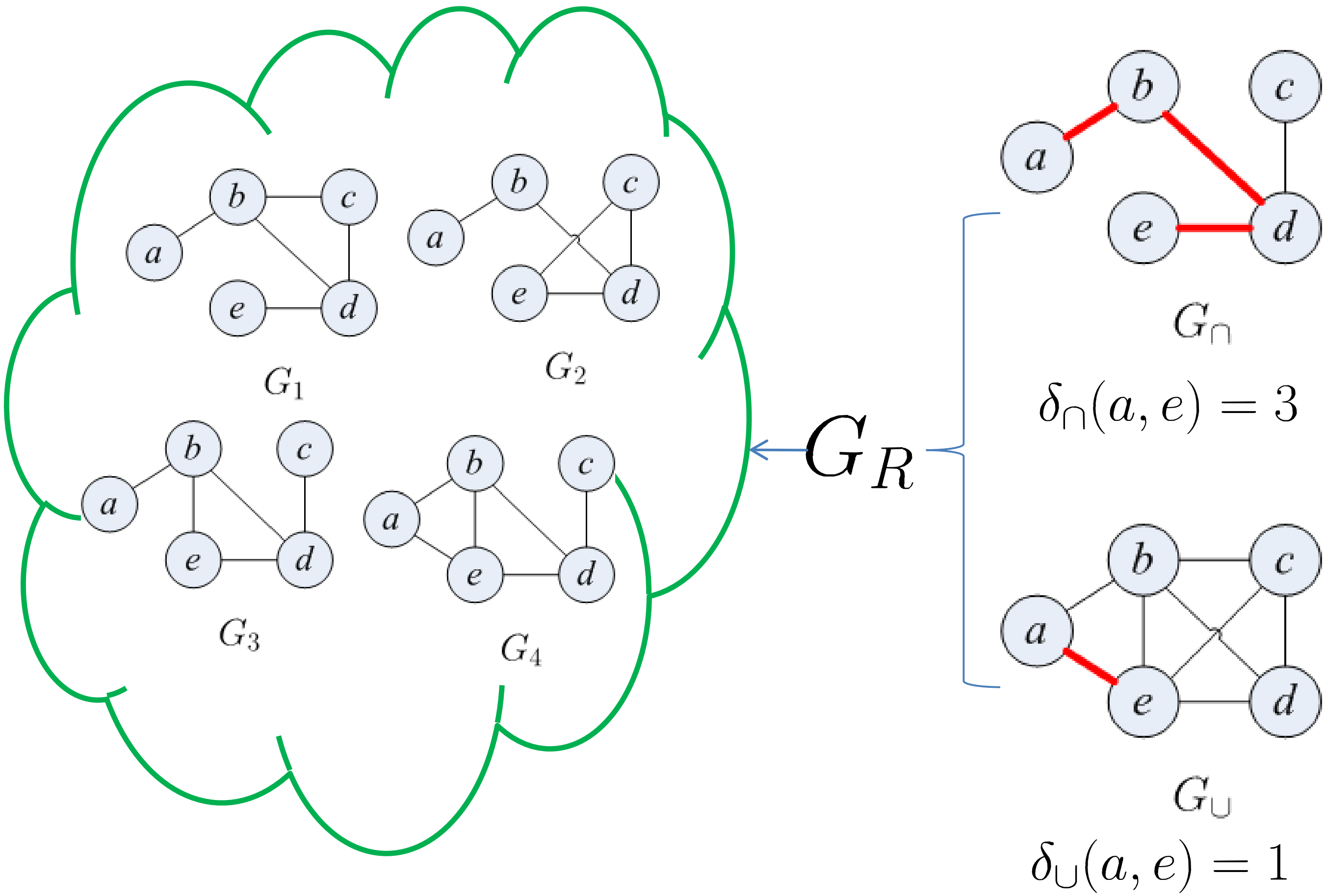
For each cluster maintain (in memory):

- $G_{\cap}$
- $G_{\cup}$
- $\Delta(G_{i,}\ G_{\cap})$

Find-Verify-and-Fix: Find

Find: Apply query on the cluster representatives

Shortest path query: Find shortest path between *a* and *e* in all snapshots

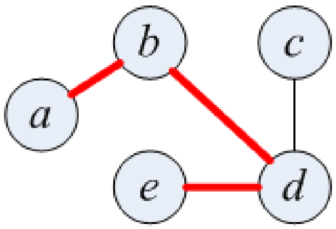


Find-Verify-and-Fix: Verify

Verify: Is the result correct on all snapshots?
Depends on the query

Bounding property:

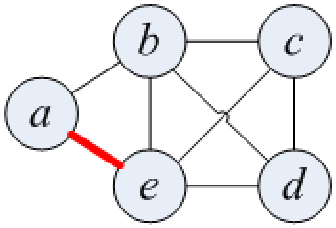
$$\delta_{\cup}(a,e) \leq \delta_i(a,e) \leq \delta_{\cap}(a,e)$$



$G_{\cap}$

$\delta_{\cap}(a,e) = 3$

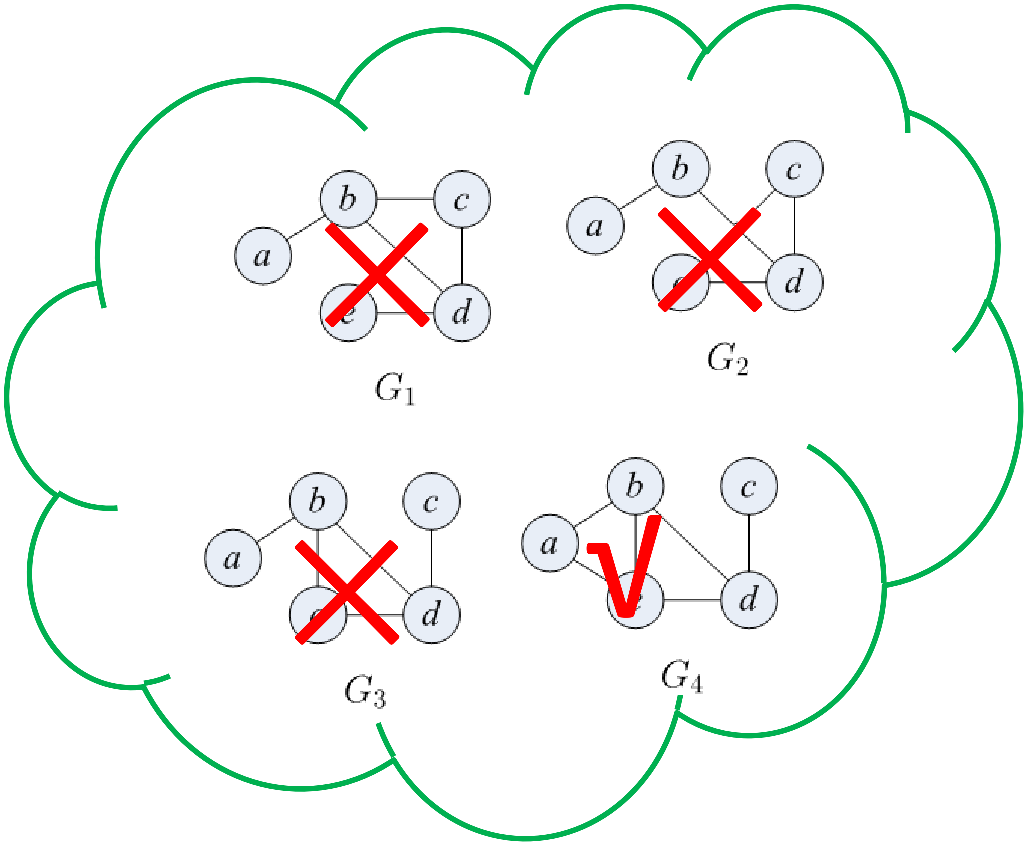
Lemma 1: If $\delta_{\cup}(a,e) = \delta_{\cap}(a,e)$, then $\widetilde{P}_{\cap}(a,e)$ (a shortest path for $G_{\cap}$) is a solution for any $G_i \in C$

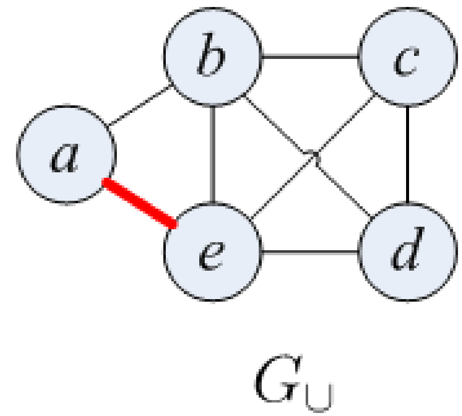


$G_{\cup}$

$\delta_{\cup}(a,e) = 1$

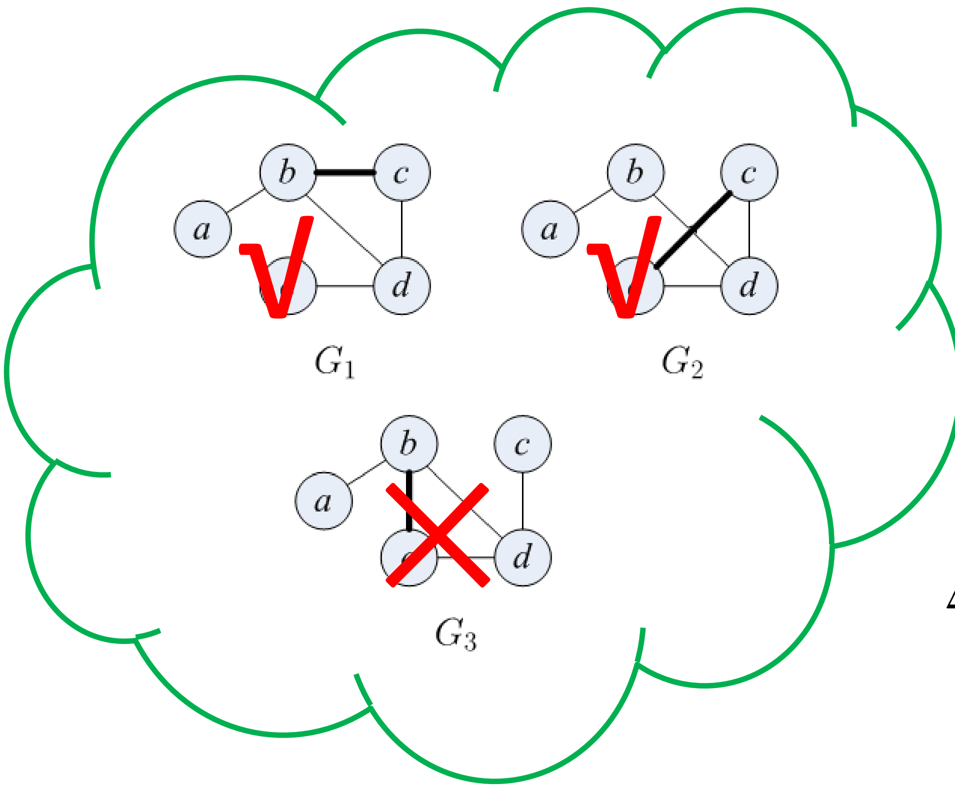
Find-Verify-and-Fix: Verify

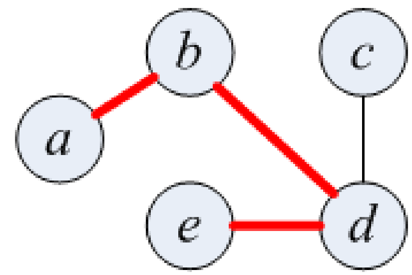




Lemma 2: If $\widetilde{P}_\cup(a, e)$ (a shortest path for $G_\cup$) exists in G_i, then $\widetilde{P}_\cup(a, e)$ is a solution for G_i

Find-Verify-and-Fix: Verify





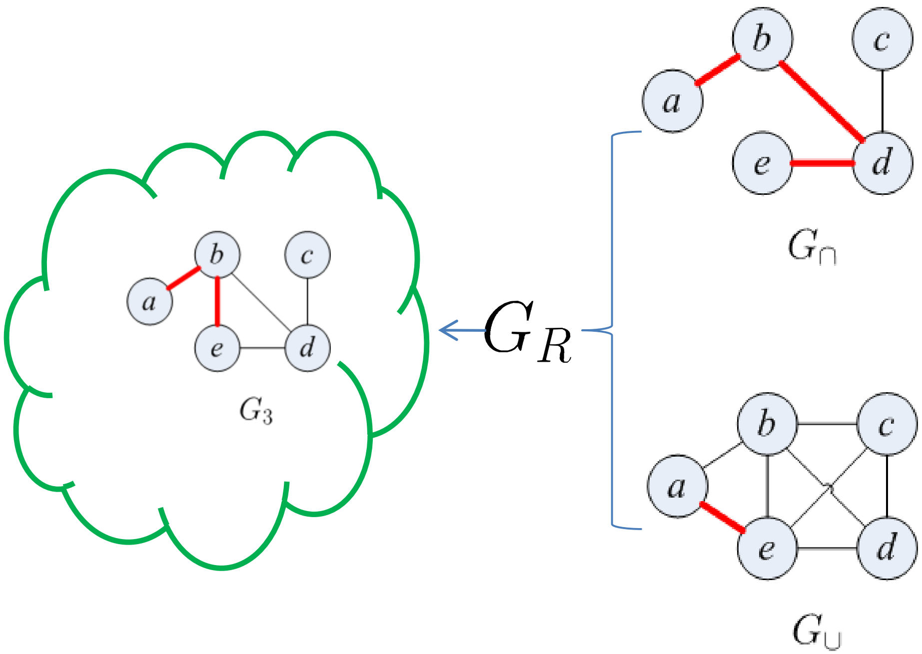
$G_\cap$

$\Delta(G_i, G_\cap)$ is bolded.

Idea of Lemma 3: If no edges in $\Delta(G_i, G_\cap)$ can give a path shorter than $\widetilde{P}_\cap(a,e)$, then $\widetilde{P}_\cap(a,e)$ is a solution for G_i

Find-Verify-and-Fix: Fix

Fix: Run shortest path queries for the snapshots that cannot be verified



Idea of Lemma 4: Fix solutions for $G_\cap$ and $G_\cup$ based on $\Delta(G_i, G_\cap)$.

Processing methods (so far)

2-phase Processing

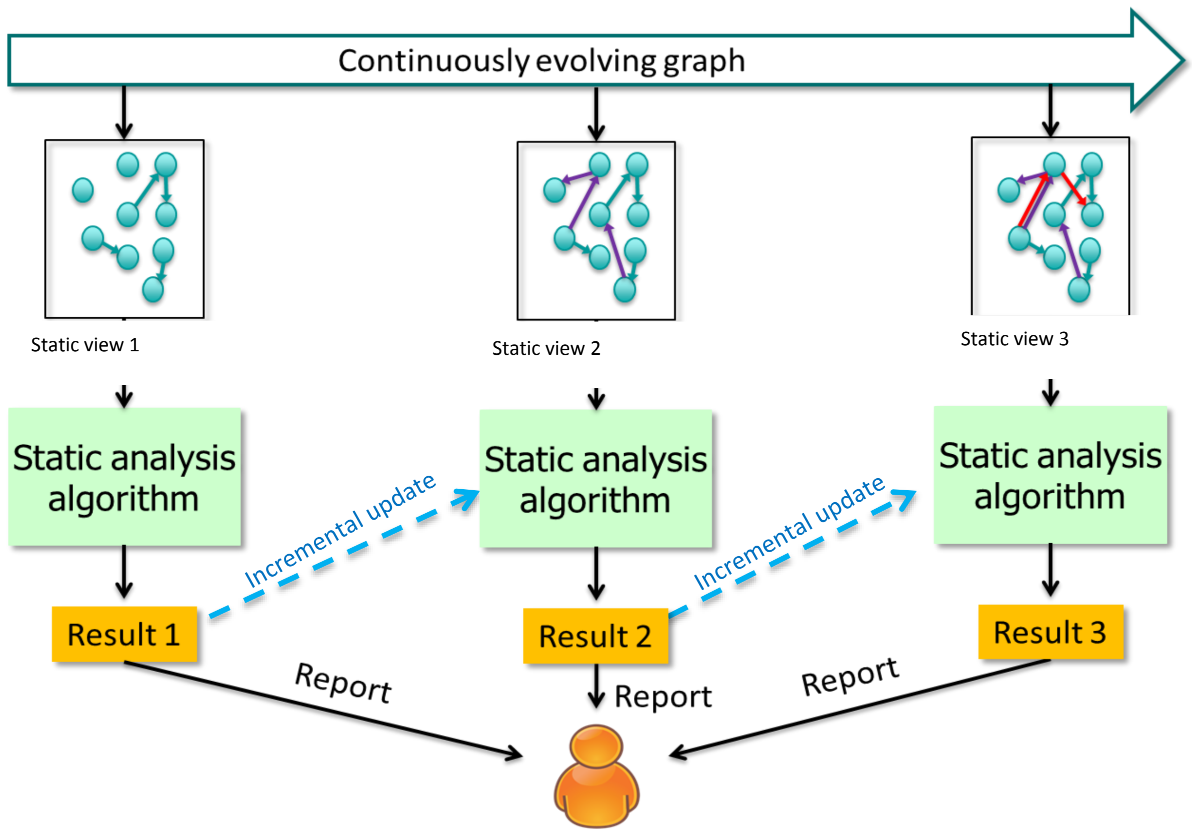
1. Re-construct snapshot
   - either the whole graph or a subgraph
2. Apply static algorithm in each of them

FVF Processing

- Find: apply query to cluster representatives
- Verify: check if the result is correct for each snapshot $G_{i,}$ or can easily be modified
- Fix: apply query to all snapshots that cannot be verified

Processing Models

Incremental (more applicable to dynamic graphs)



Slide from sigmod2016 tutorial

Batch (Iterative) Processing

2 Phase and FVF high redundancy:
The same static algorithm is applied many times

Can we avoid this by exploiting *time locality*?

Batch Computation across time (snapshots) => Chronos

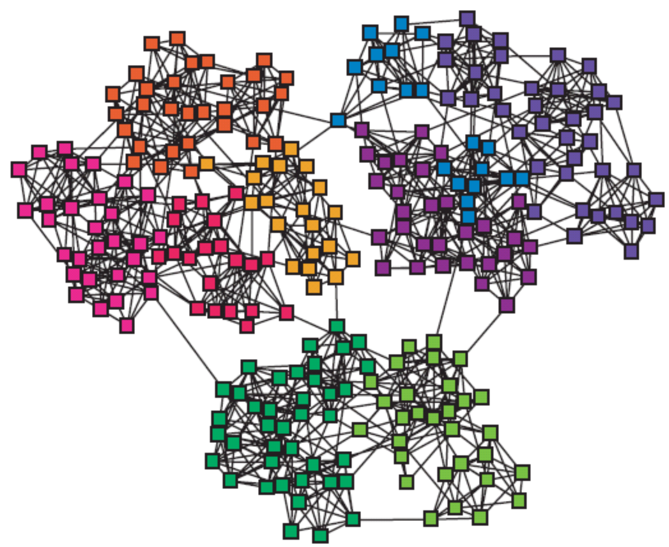
Requires a specific node layout based on graph partitioning

Partitioning: why?

Applications

- parallel or distributed computation, assign a different partition to a core or machine
- (in memory) caching: storage layout
- computation (propagate information among nodes)

Graphs low (structural) locality

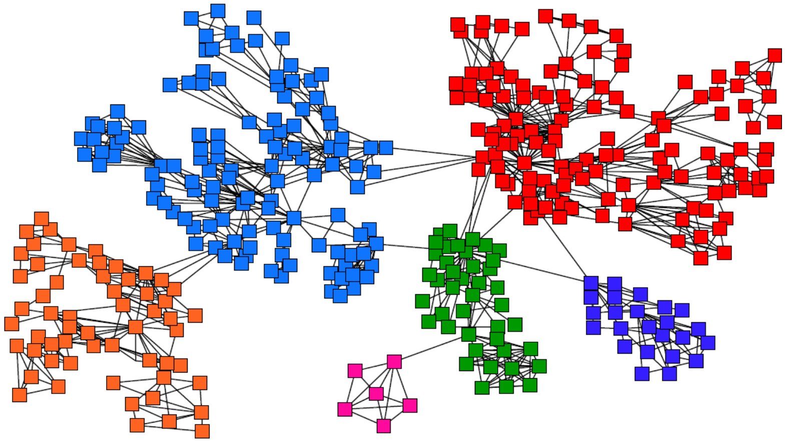


Partitioning (background)

Partitioning as an optimization problem:

Partition the nodes in the graph such that

- nodes *within clusters* are *well interconnected* (high edge weights), and
- nodes *across clusters* are *sparsely interconnected* (low edge weights)



In static graphs

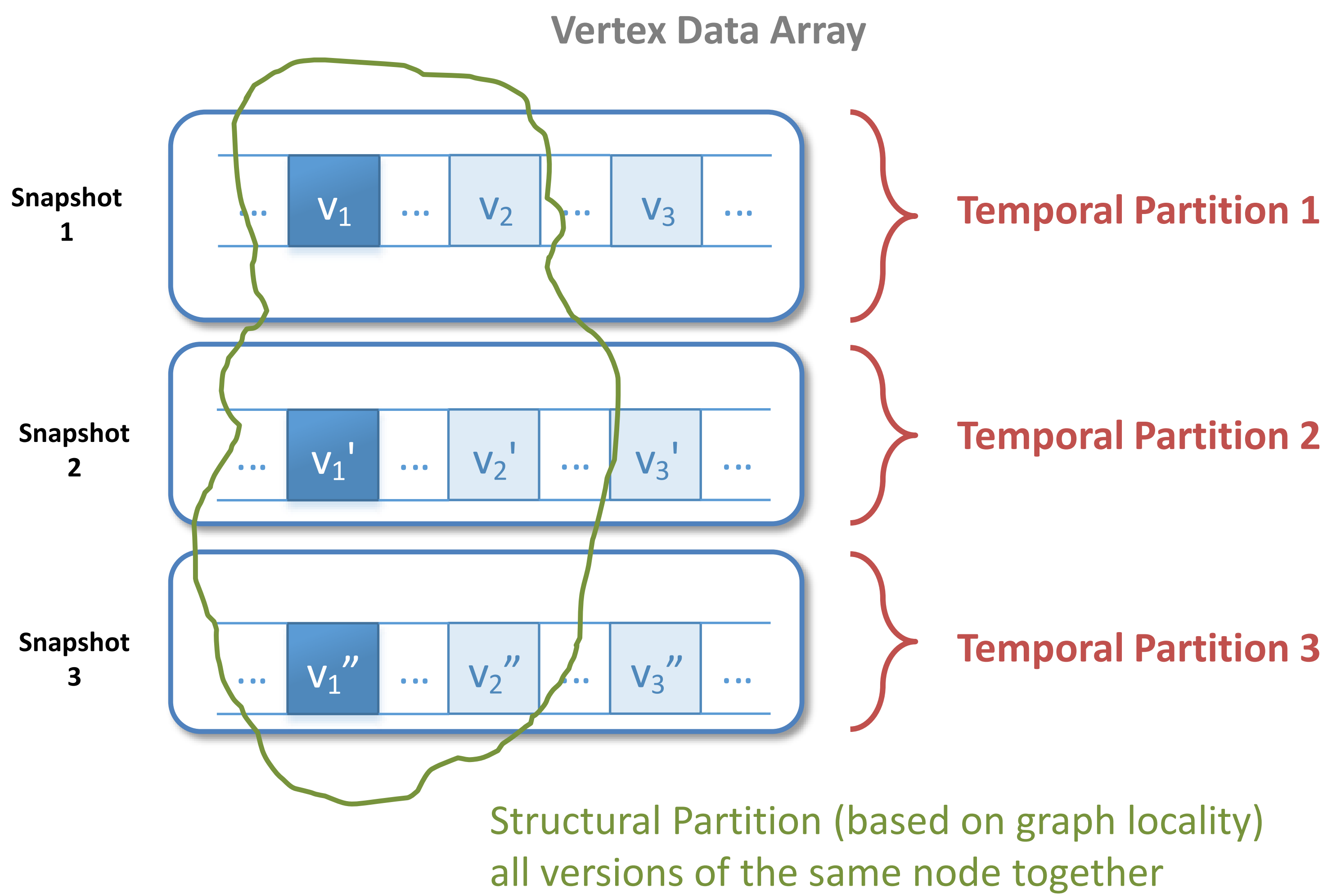
- Random (hash on nodes) (load balancing)
- Structural locality ((normalize) edge cut, max flow, METIS, modularity, spectral clustering)

Partitioning historical graphs

Two levels of locality (at a high level):

- Structural: partition by node (as in static)
- Temporal (or time): partition by time e.g., every 10 snapshots

Partitioning (high level)



Chronos [EuroSys14, ACM ToS, 2015]

In main memory, multi core graph engine

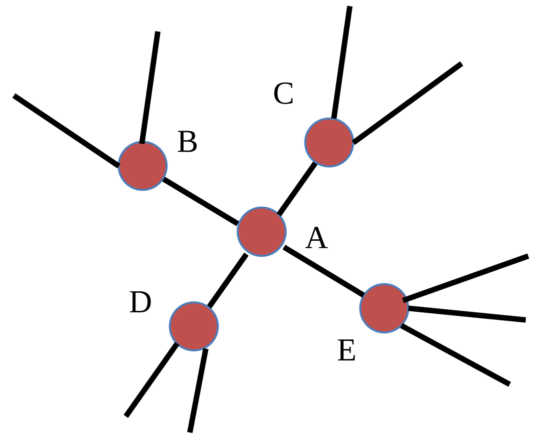
Scope: multi-snapshot historical analytical queries

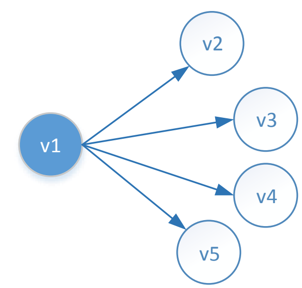
Think like a vertex (background)

Typical graph operation (GAS)

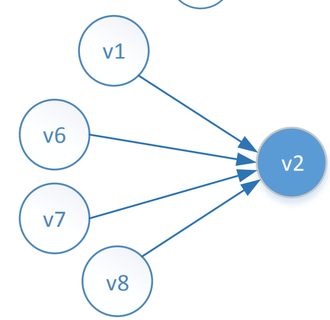
Works in iterations

- Each vertex assigned a value
- In each iteration, each vertex:
  - Gathers values from its immediate neighbors (vertices who join it directly with an edge). E.g., @A: B→A, C→A, D→A,…
  - Applies some computation using its own value and its neighbors values.
  - Updates its new value and scatters it out to its neighboring vertices. E.g., A→B, C, D, E
- Graph processing terminates after: (i) fixed iterations, or (ii) vertices stop changing values





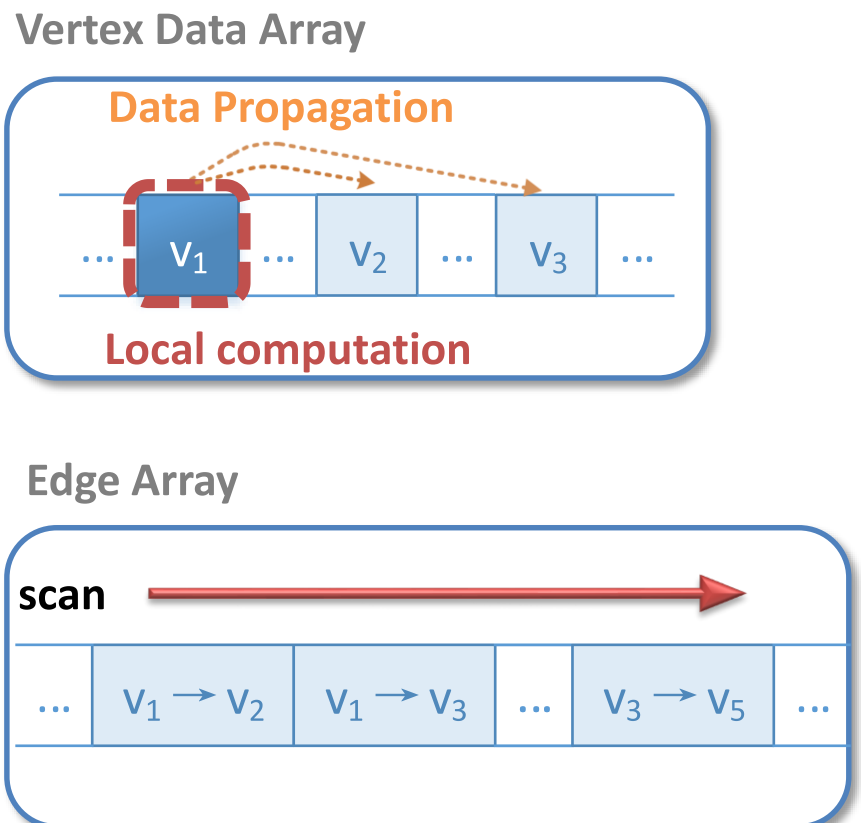
Push Mode

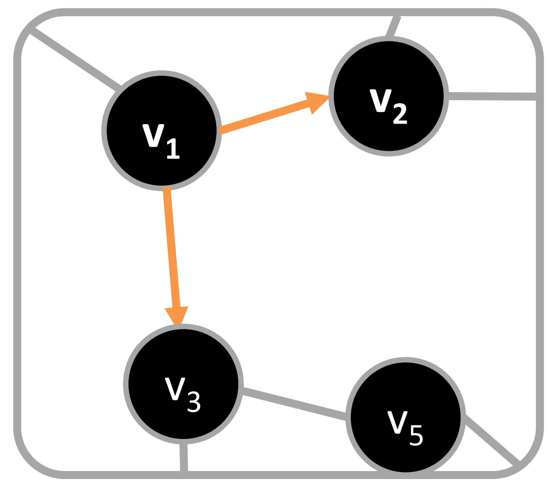


Pull Mode

Chronos: Revisit Static Graph Analysis

Propagation (vertex) based graph computation model

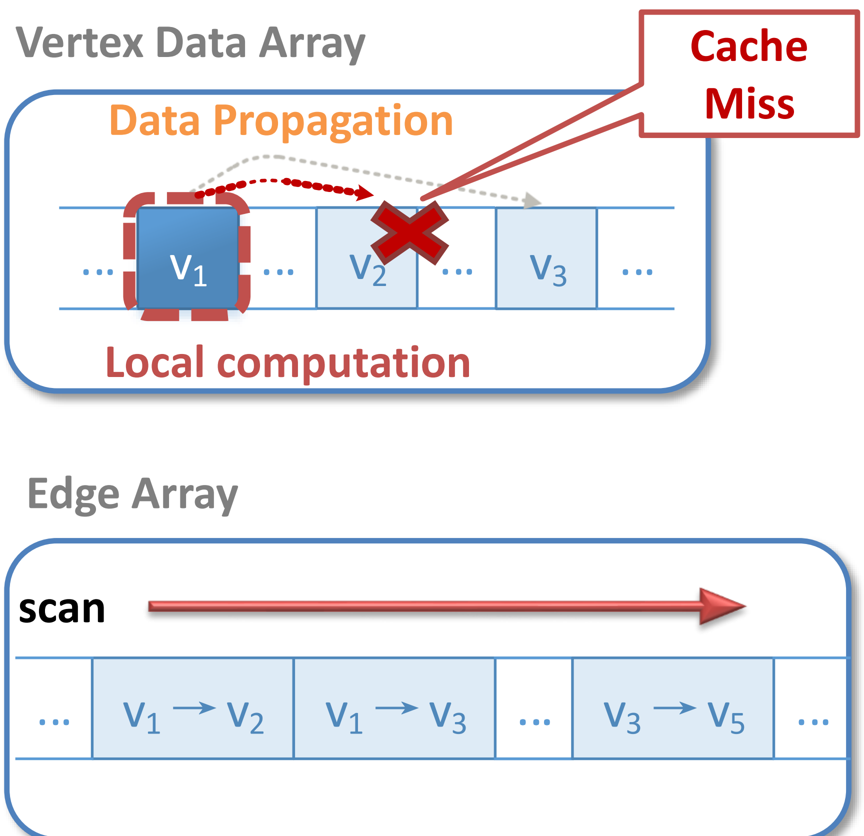


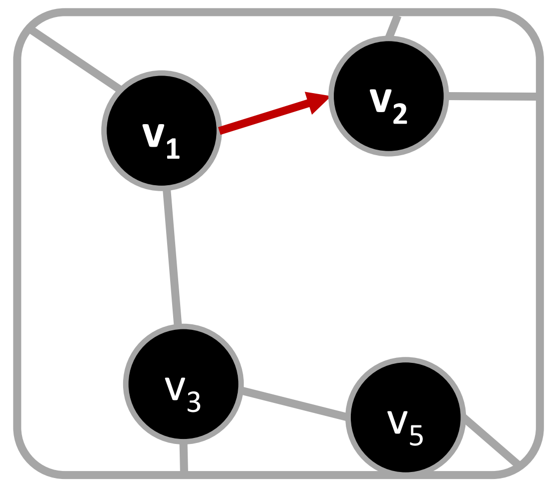


slides from EuroSys14 presentation

Chronos: Revisit Static Graph Analysis

Propagation (vertex) based graph computation model

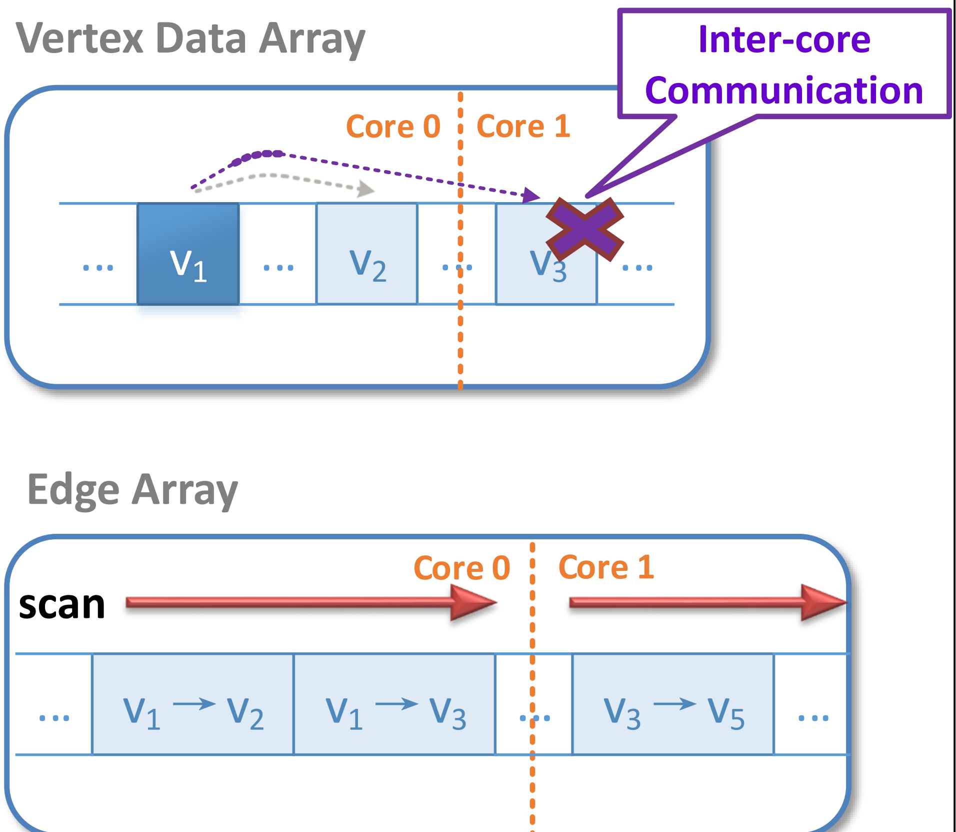


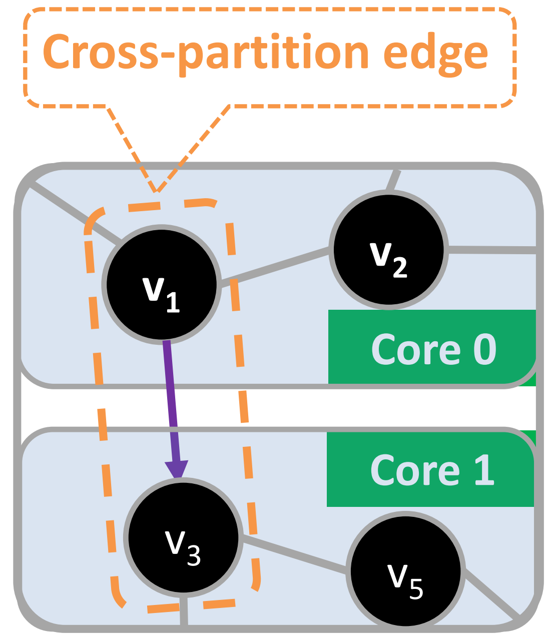


slides from EuroSys14 presentation

Chronos: Revisit Static Graph Analysis

In parallel: **partition** graph & computations among CPU cores

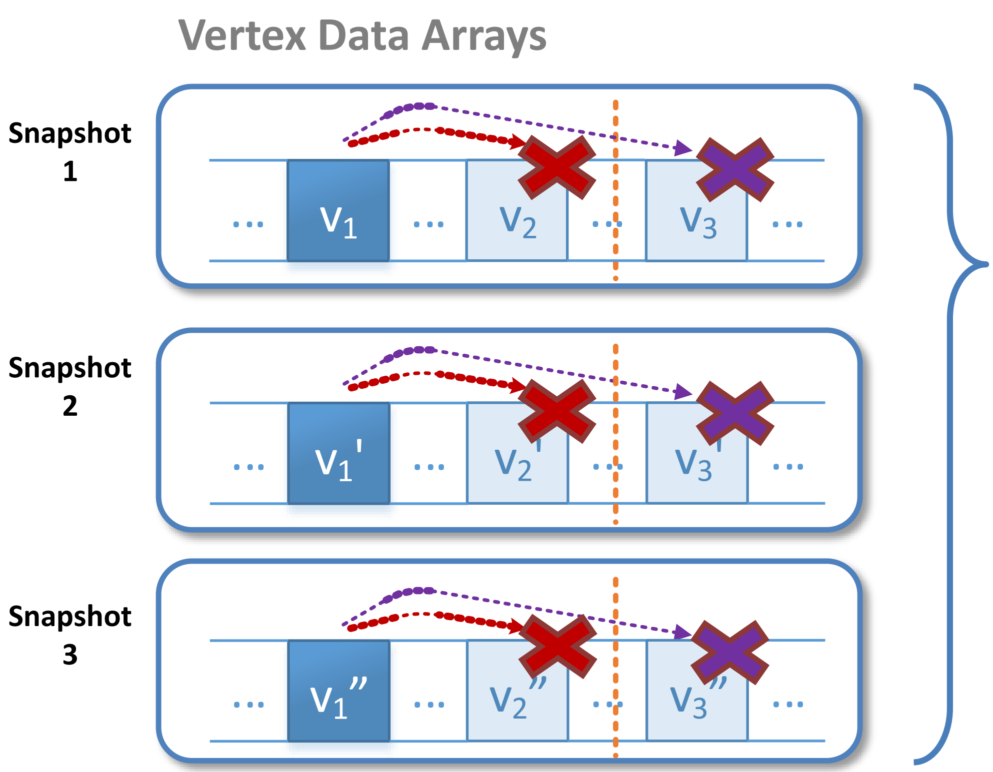




slides from EuroSys14 presentation

Chronos: snapshot by snapshot (2phase) QP

Computation on **multiple** graph snapshot – **multiple** cost

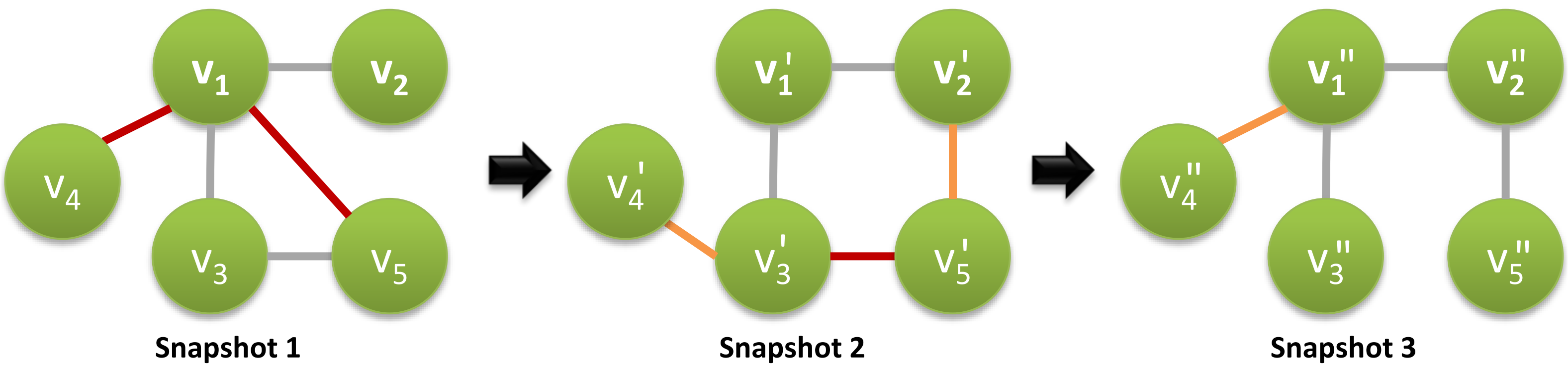


N snapshots
⇒ N cache misses
⇒ N inter-core comm.

slides from EuroSys14 presentation

Chronos observation: Time locality

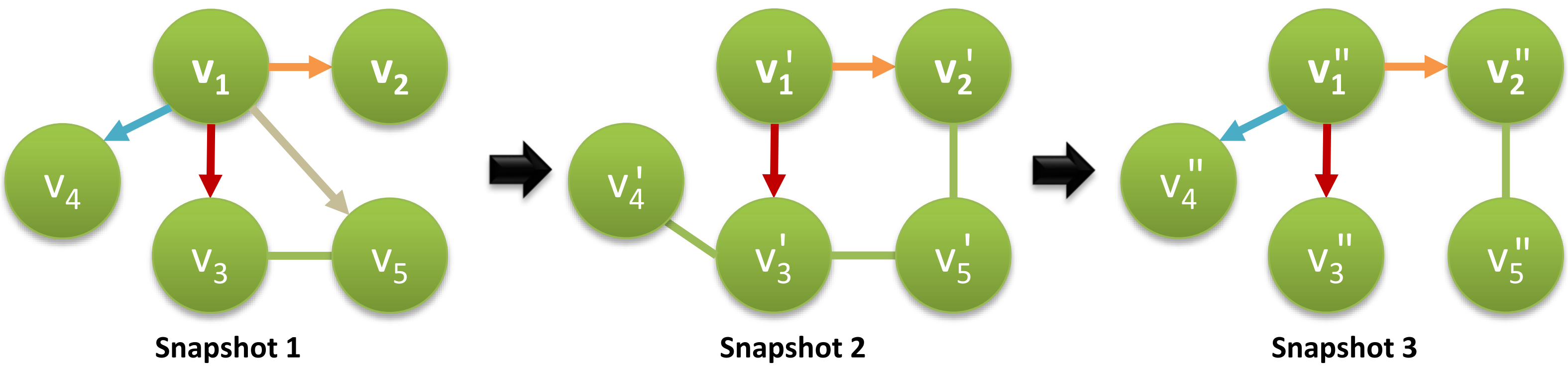
Real-world graph often evolve gradually (similar snapshots)



slides from EuroSys14 presentation

Chronos observation: Time locality

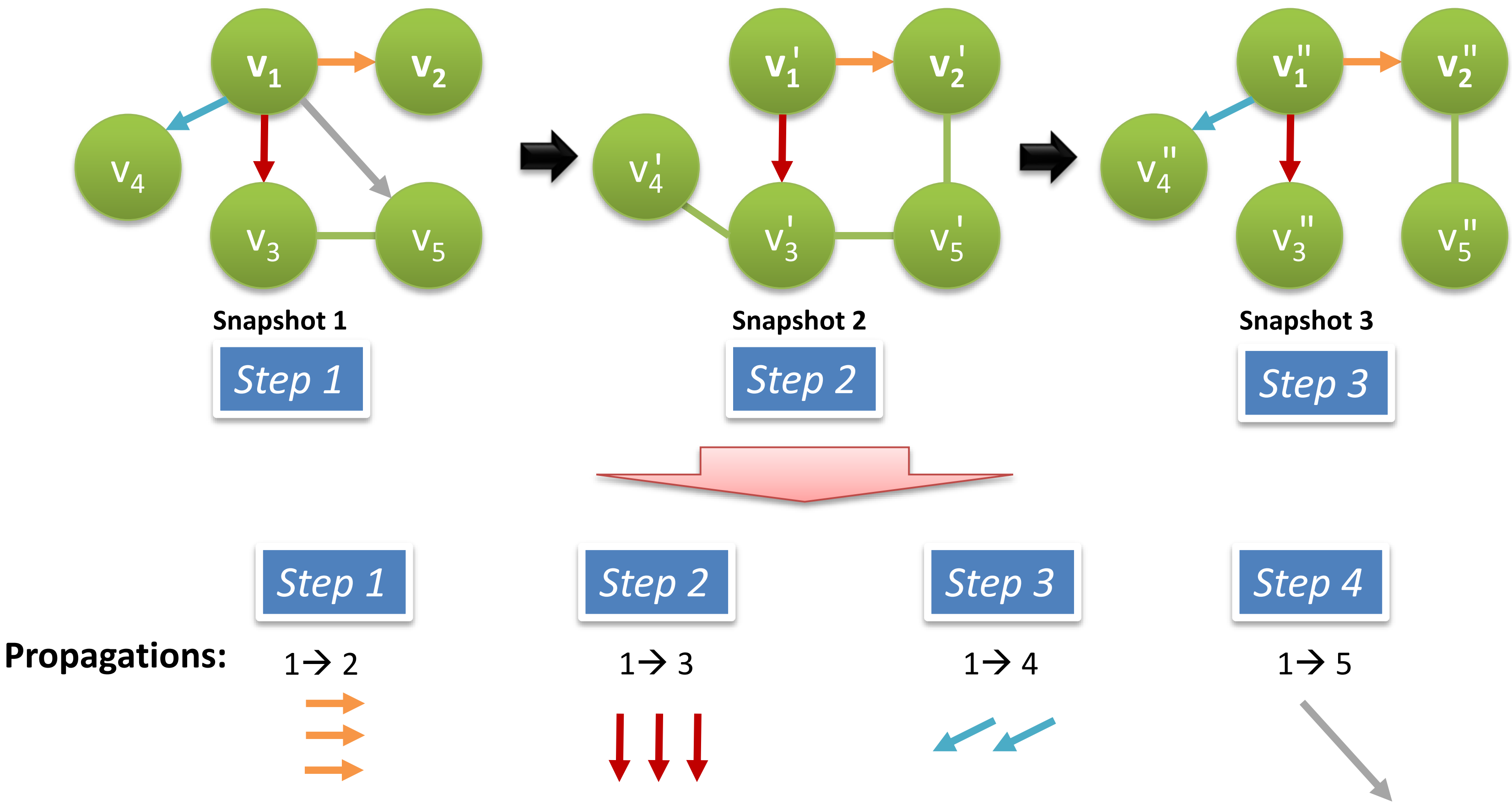
Similar propagations across snapshots



slides from EuroSys14 presentation

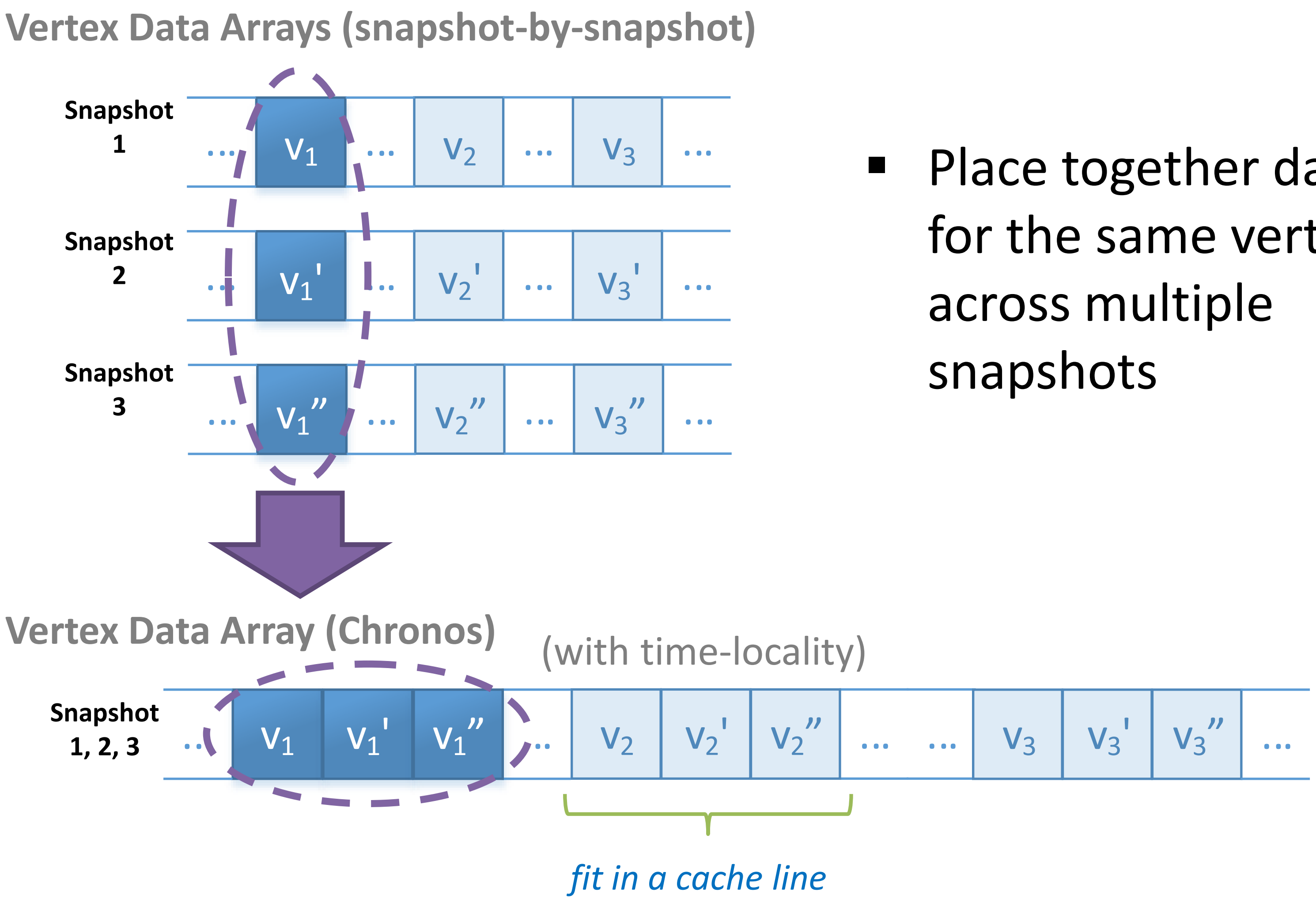
Chronos

Group propagations by source & target, not by snapshot



slides from EuroSys14 presentation

Chrono: Data Layout

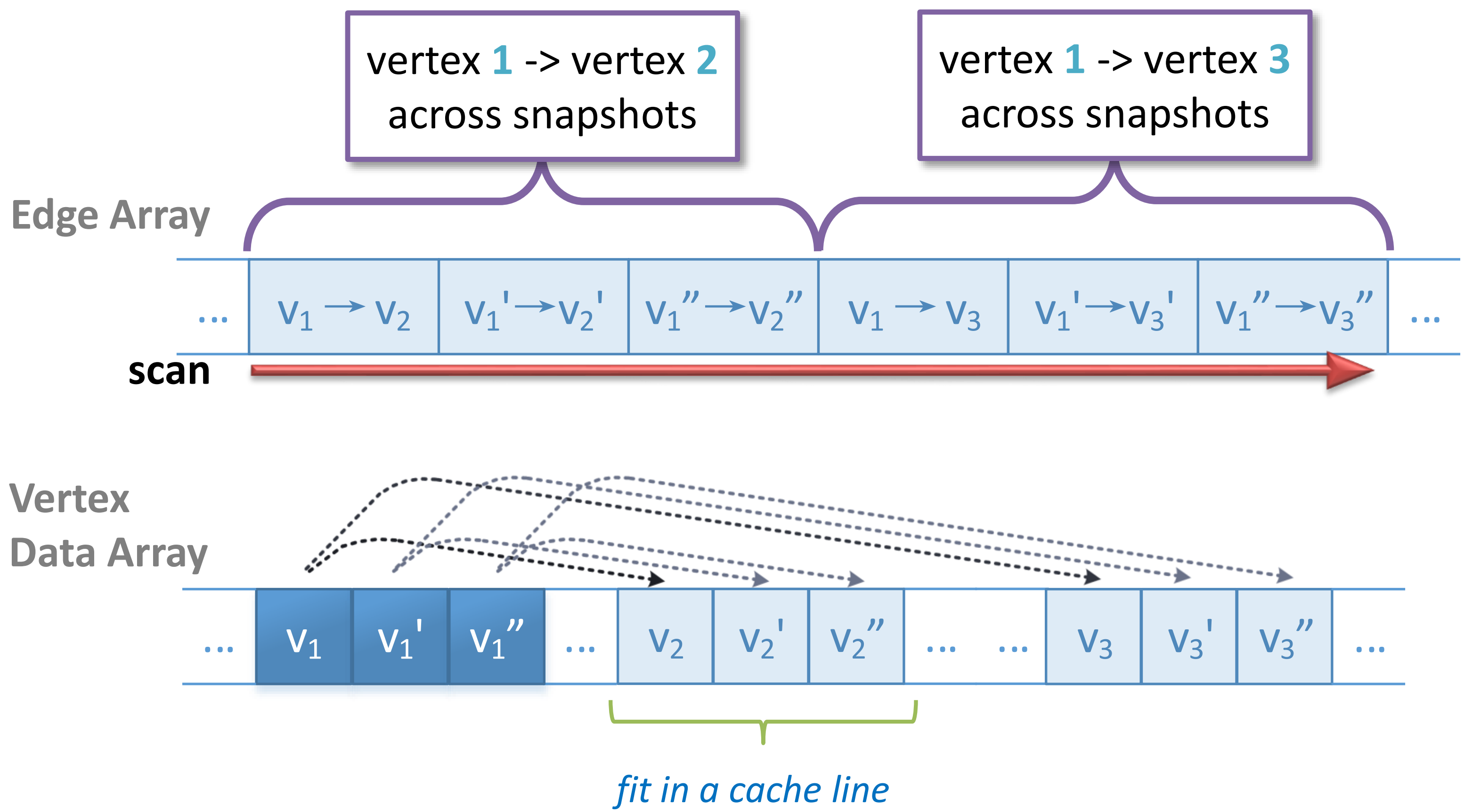


- Place together data for the same vertex across multiple snapshots

slides from EuroSys14 presentation

Chronos: Propagation Scheduling

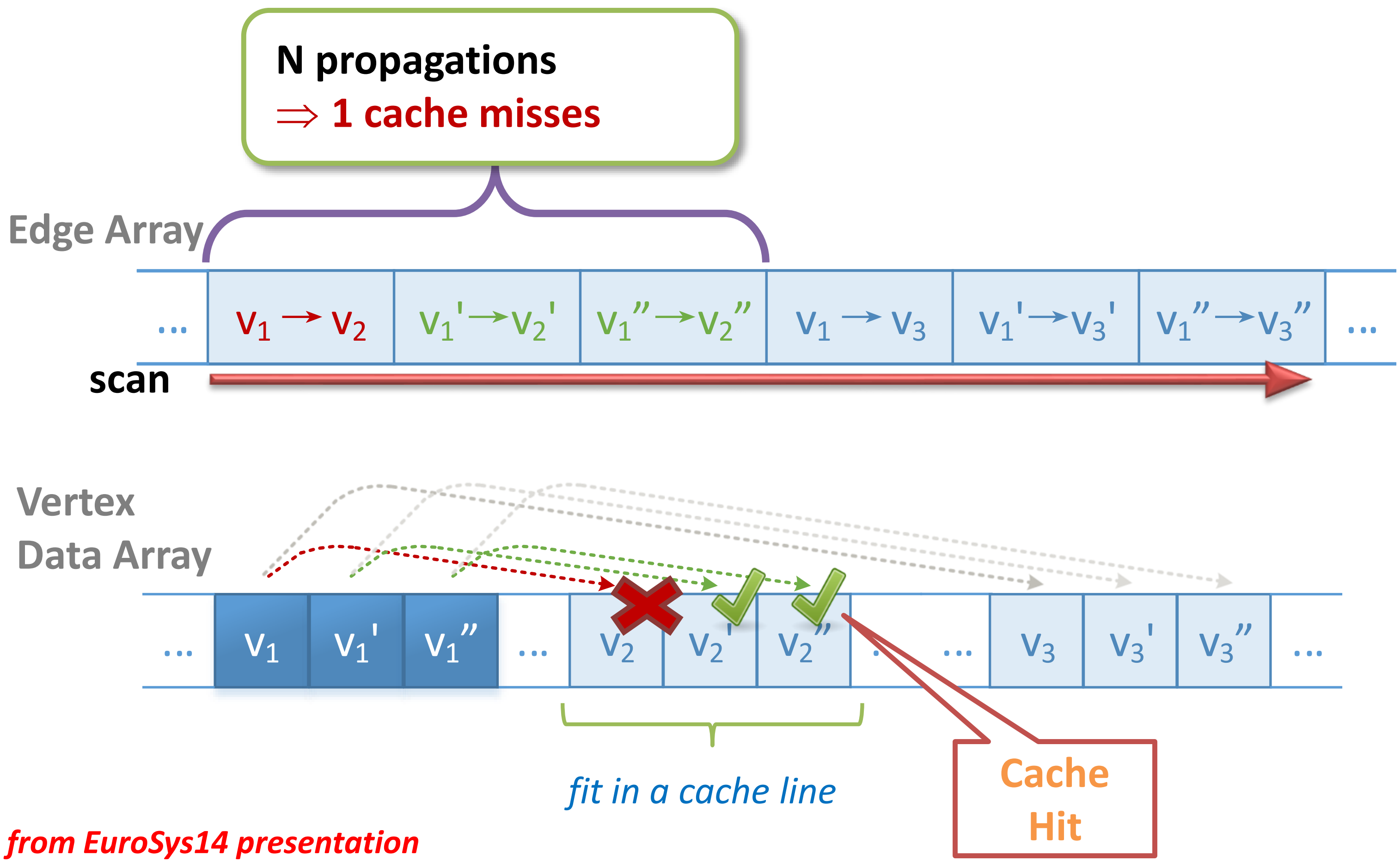
- **Locality Aware Batch Scheduling (LABS):**
  - Batching propagating across snapshots



slides from EuroSys14 presentation

Chronos: Propagation Scheduling

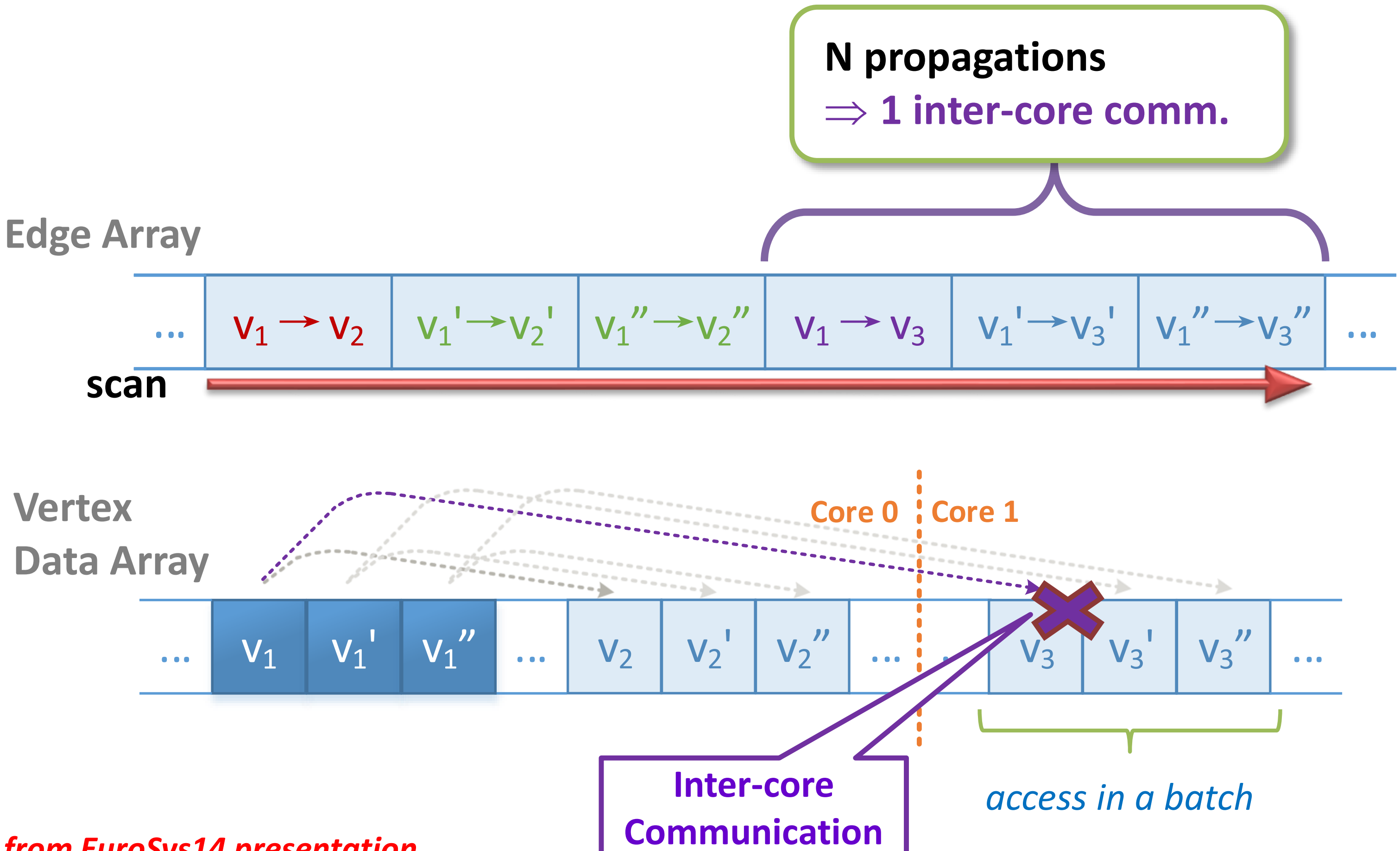
- **Locality Aware Batch Scheduling (LABS):**
  - Batching propagating across snapshots



slides from EuroSys14 presentation

Chronos: Propagation Scheduling

- **Locality Aware Batch Scheduling (LABS):**
  - Batching propagating across snapshots

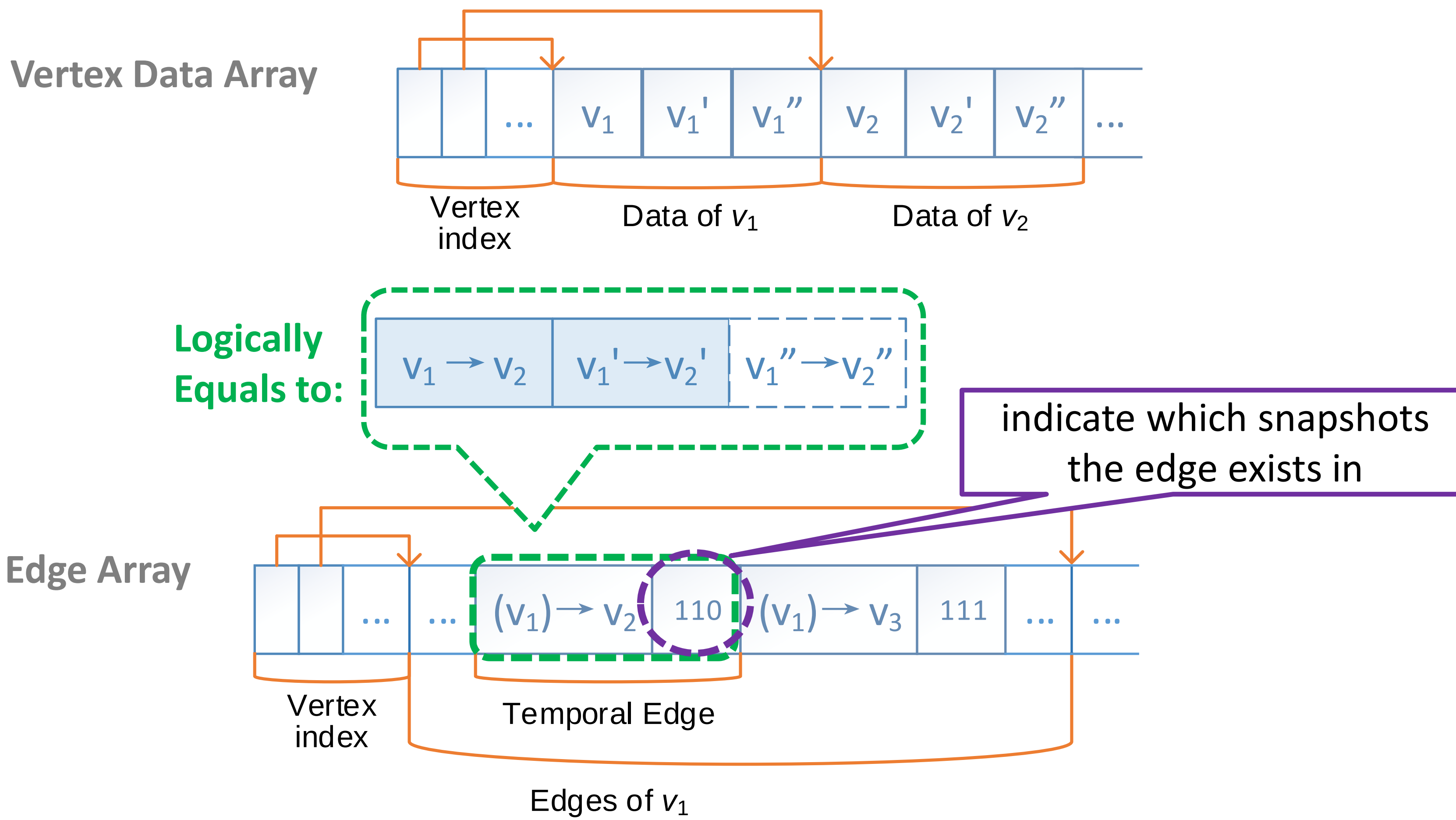


slides from EuroSys14 presentation

Chronos: Key Points

- A graph layout
  - Place together nodes/edge data across snapshots
- QP mechanism
  - Batch propagations across snapshots

Chronos: In main memory



slides from EuroSys14 presentation

Chronos: Parallelization Summary

Good partitioning: Num. of intra-partition edge > Num. of inter-partition edge

	Partition Parallelism	Snapshot Parallelism	LABS-Parallelism
Cache Miss			
Inter-core Communications			

?

Snapshot by snapshot

LABS

Partition-Parallelism: Computing partitions of the same snapshot in parallel
Snapshot-Parallelism: Computing snapshots in parallel
LABS-Parallel: Computing LABS-batched partition in parallel

slides from EuroSys14 presentation

SAMS [PVLDB17]

Same idea with Chronos

Scope: multi-snapshot historical analytical queries

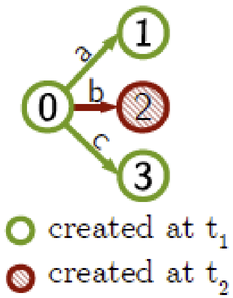
Single Algorithm Multiple Snapshots (SAMS): same algorithm many snapshots

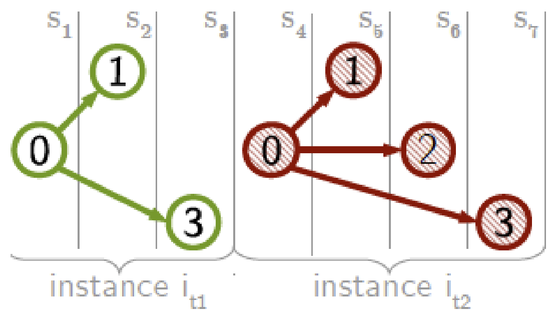
But Chronos is vertex-centric, while SAMS propose *automatic transformation* of graph algorithms and also not only for GAS computation

Two basic transformations

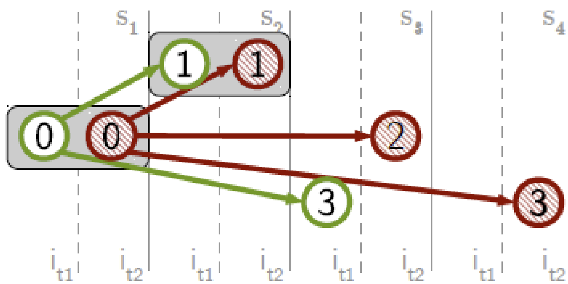
- Program instance interleaving
- Synchronization of graph accesses

SAMS: example





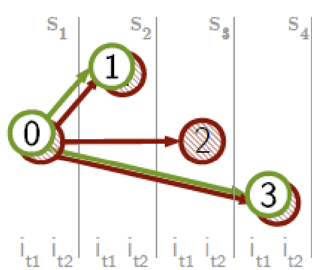
One snapshot at a time



Interleaving: automatically transform an algorithm so that all its instances concurrently execute the same statement

$$i_{t_1} : G^S.neighbors(v_0) = [v_1, v_3]$$

$$i_{t_2} : G^S.neighbors(v_0) = [v_1, v_2, v_3]$$



Synchronization: ensures that all active instances process the same graph element (an instance is active for a statement, if the single snapshot would execute statement)

works for for-loops over nodes and neighbors sets

Processing Models (summary)

2 Phase

- execute static algorithm at each snapshot
  - snapshot parallelism
  - partial snapshots
  - no snapshots

FVF

- cluster similar snapshots
- execute static algorithm on cluster representatives
- verify results
- execute static algorithm on non verifiable snapshots

Incremental

- use results on snapshot at time t to compute result on snapshot at time t+1

Iterative (or batch)

- concurrently execute all instances of the algorithm

Recency-based processing

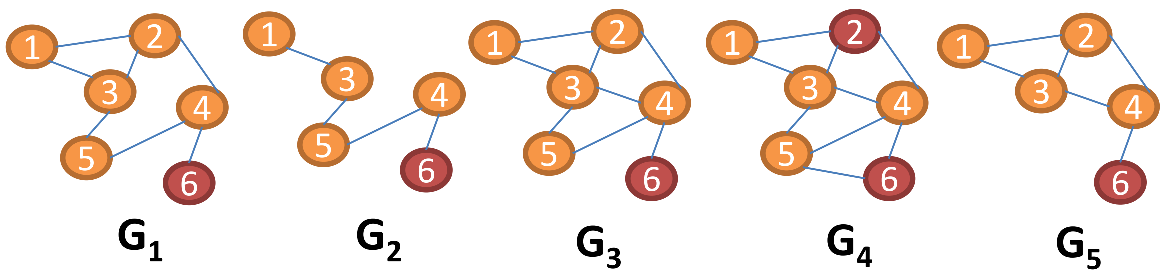
- So far, in historical graphs, all snapshots consider equal
- In dynamic graphs, only the *recent* one
- Introduce *aging* or *decay*, to favor recent snapshot

Example: TIDE [ICDE2015]

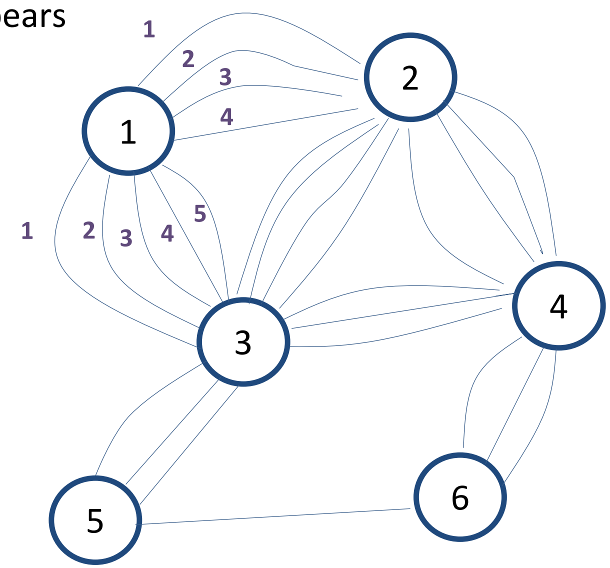
TIDE [ICDE15]

- Target query: continuously deliver analytics results on a dynamic graph
- Model social interactions as a dynamic interaction graph
  - New interactions (edges) continuously added
- Probabilistic edge decay (PED) model to produce static views of dynamic graphs
  - Intuition: sample edges from each snapshot with probability that decreases with the time of the edge so that older edges have a smaller probability to be included in the static view than newer edges

TIDE



Aggregate graph:
Union graph where each edge appears many times



Let τ be the current time
Sample each edge e with probability
$P^f(e) = f(\tau - \text{timestamp}(e))$

f non-increasing decay function – as the edge ages probability remains the same or drops
Every edge e

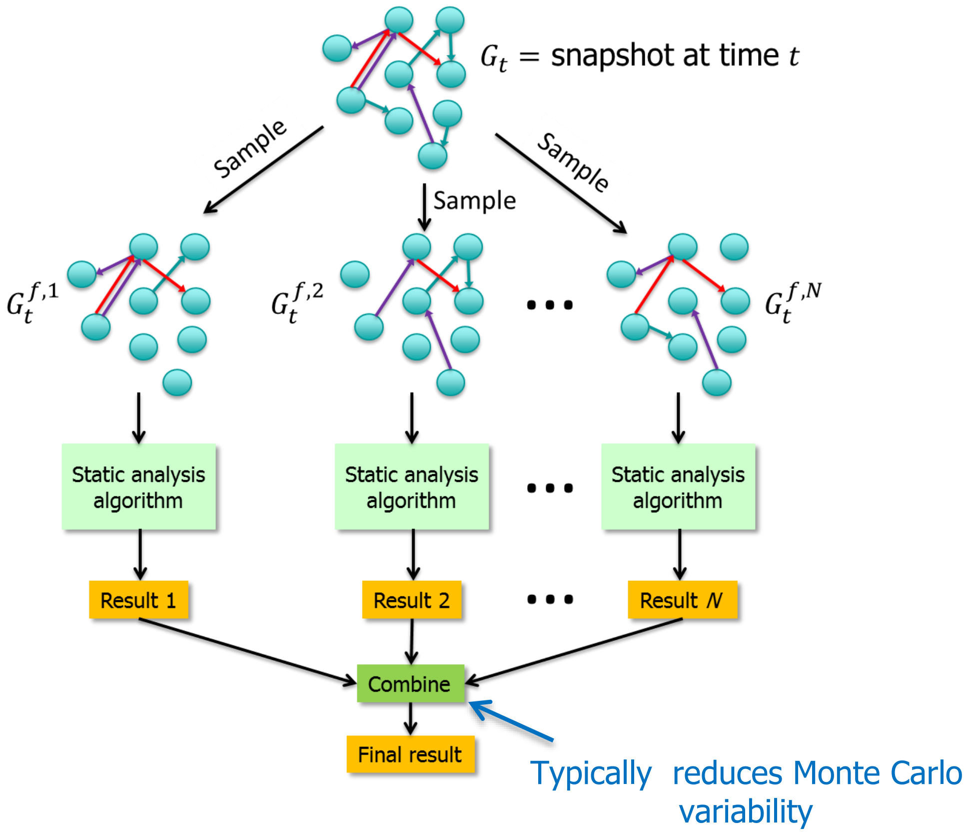
- has a non-zero chance of being included in the analysis (continuity)
- change becomes increasingly unimportant over time, so that newer edges are more likely to participate (recency)

TIDE: PED

G_t: aggregate graph at t

Edge color – time instance

Create N independent sample graphs



Processing Models (summary)

2 Phase

apply static algorithm at each snapshot

FVF

cluster similar snapshots
apply static algorithm on cluster representative
verify results
execute static algorithm on non verifiable snapshots

Iterative (or batch)

concurrently execute all instances of the algorithm

Incremental

use results on snapshot at time t to compute result on snapshot at time t+1

Recency-based

create one (or more) sample static graphs by sampling the aggregate graph
apply static algorithm on the samples
combine the results

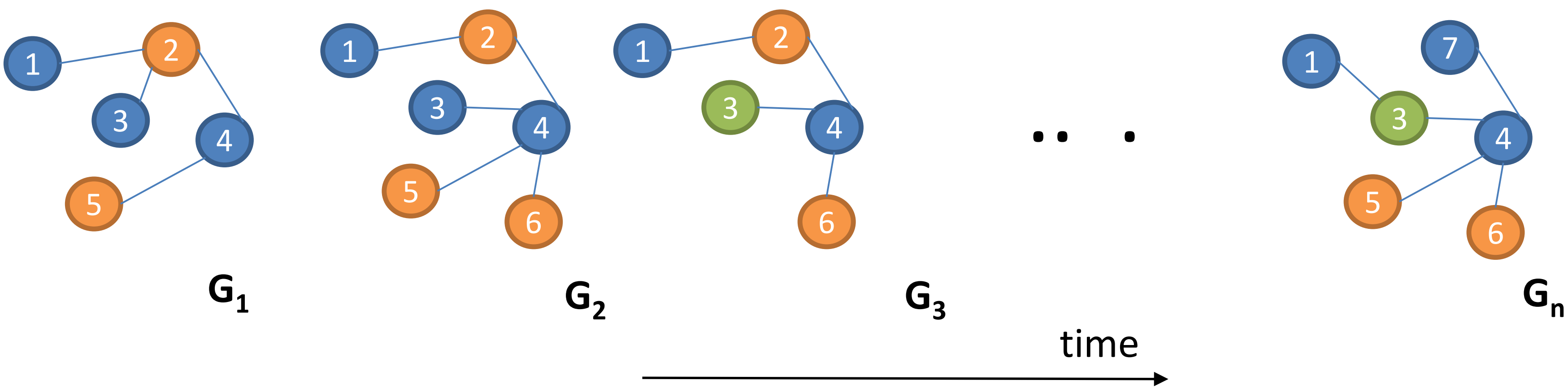
End of Part 1
break!

Part 2

Queries: navigation (longer part), pattern matching

Evolving Graph (recap)

Time-evolving or *historical graph* is a sequence of graph *snapshots* G_t capturing the state of the graph at time point or instance *t*

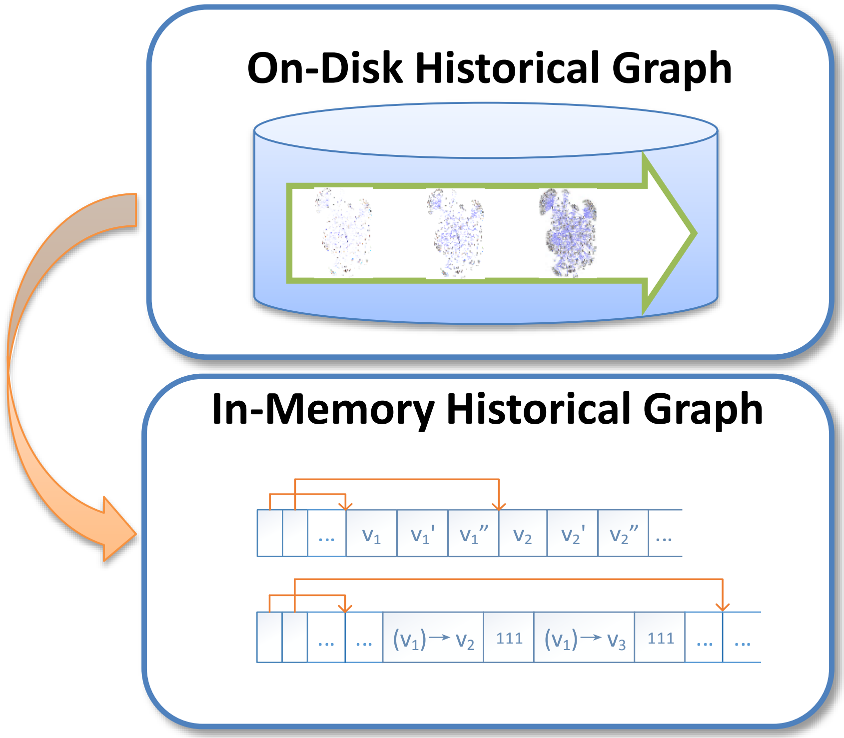


Processing (recap)

Queries on time-evolving graphs

- Historical: Apply query at past snapshots
- Durable: Return the results that hold for the longest time
- Evolution: Ad hoc exploration – eg find patterns with similar evolution

Representation, storage (recap)



All information in

- Files
- DBMS (relational or graph database)

COPY: materialize all snapshots
LOG: maintain operations
HYBRID: materialize selected snapshots
VERSIONING

Selected snapshots
CSR format
Adjacency lists
+ versioning

Processing Models (recap)

2 Phase

apply static algorithm at each snapshot

FVF

cluster similar snapshots
apply static algorithm on cluster representatives
verify results
execute static algorithm on non verifiable snapshots

Iterative (or batch)

concurrently execute all instances of the algorithm

Incremental

use results on snapshot at time t to compute result on snapshot at time t+1

Recency-based

create one (or more) sample static graphs by sampling the aggregate graph
apply static algorithm on the samples
combine the results

Next

Look into specific graph queries

Navigational
- reachability
- shortest paths

Patterns (briefly)

Conclusions

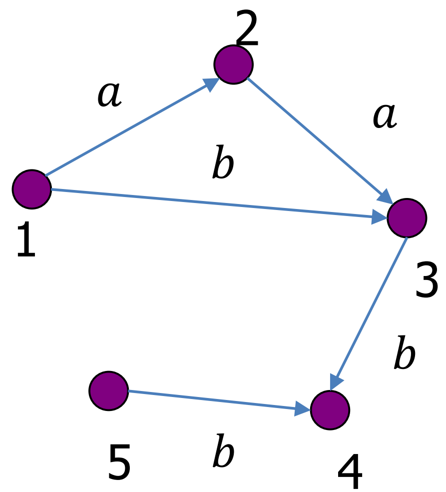
Navigational Queries

Shortest path queries
Reachability queries

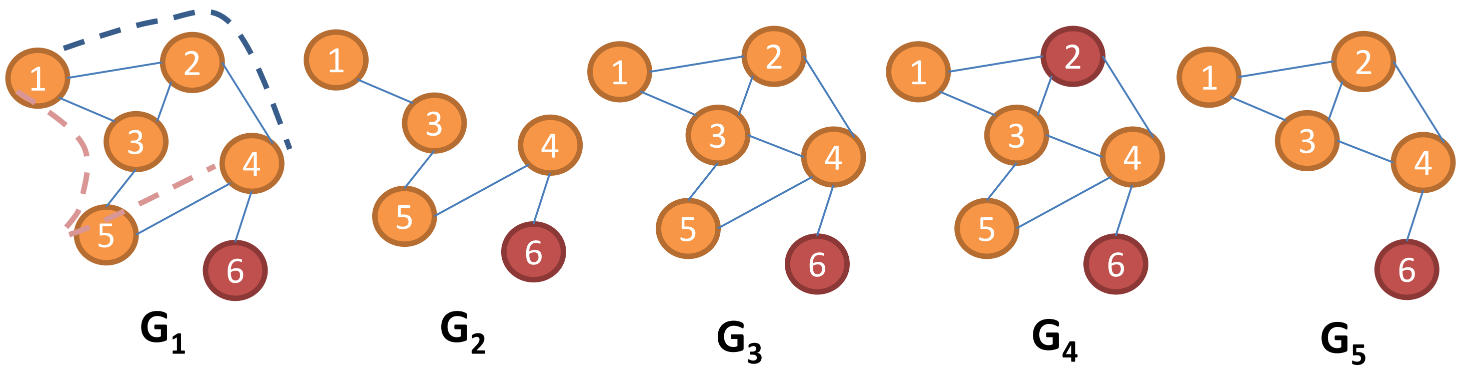
Navigational queries

- Allow navigating the topology of a graph
  - Find the friends of Maria
  - Find all people connected to Maria
- Simplest form: path queries

  P: x $\rightarrow_{C}$ y
    - Source x
    - Target y
    - *C* specifies conditions on the paths (when labels or properties)
      - Regular Path Queries, when C is a regular expression
- Reachability queries
  - ask for the existence of the path
- Shortest path queries
  - Length: no weights (number of edges)
  - also defines the distance between two nodes



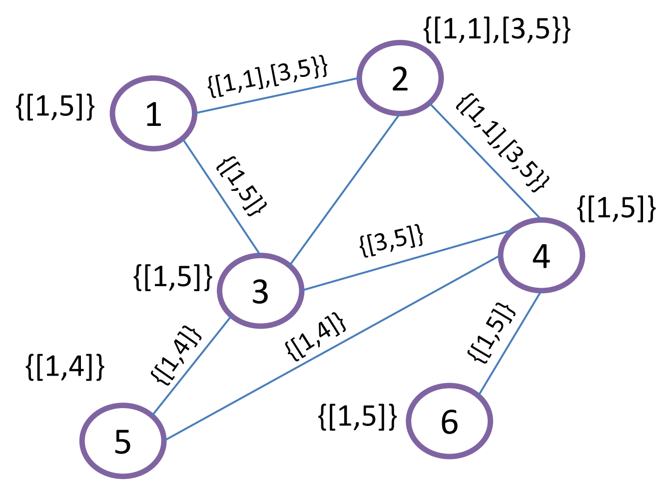
Paths in historical graphs



Find paths from 1 to 4?

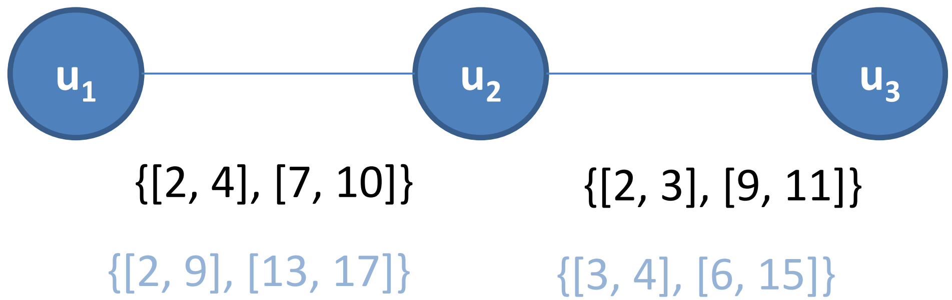
- Assume the *versioning approach* (without lack of generality)
- Assume that each edge (node) is augmented with its *lifespan*

VG



Paths in historical graphs

What is the lifespan of path $u_1u_2u_3$?



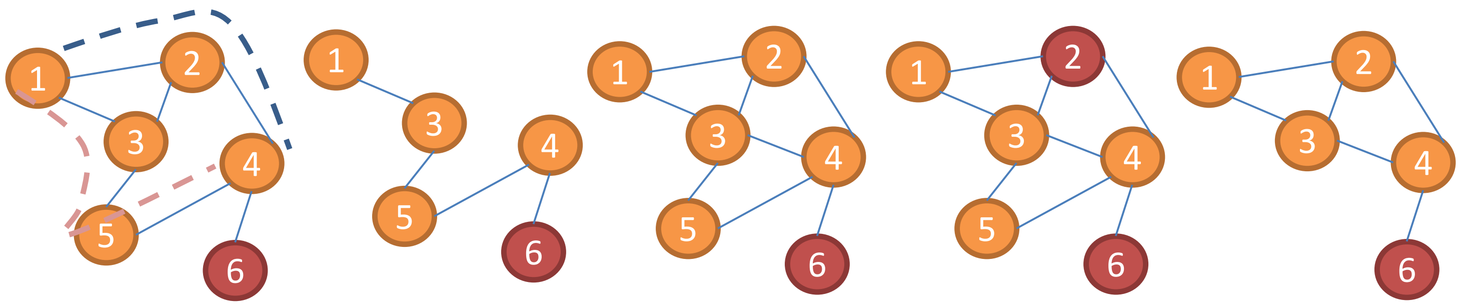
Time Join

$$I \oplus I' = I \cap I' = \{t \mid (t \in I) \wedge (t \in I')\}$$

Central operation in traversals

{[2, 4], [7, 10]} $\oplus$ {[2, 3}, [9, 11]} = {[2, 3], [9, 10]}

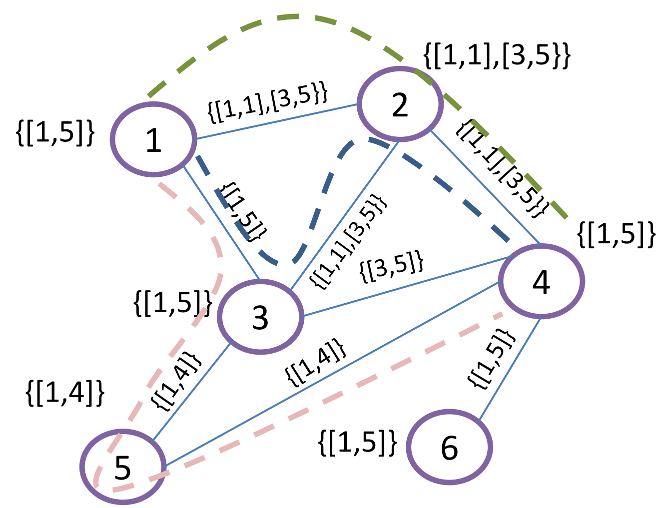
Paths in historical graphs



G_1 G_2 G_3 G_4 G_5

Paths from 1 to 4?

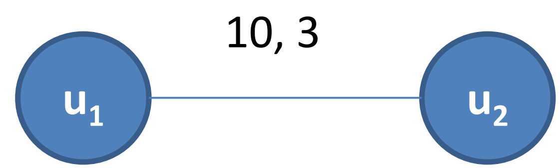
VG



Comparison with Temporal Graphs

Each edge (u, v) two values (t, λ)

- t starting (departure) time
- λ traversal (duration) time
- t + λ ending (arrival) time



Represented as (u, v, t, λ)

Applications

- *Phone call or Short Message Service networks*: start of the call and duration of the call
- *Flight graphs* (and in general transportation): departing time and flight duration

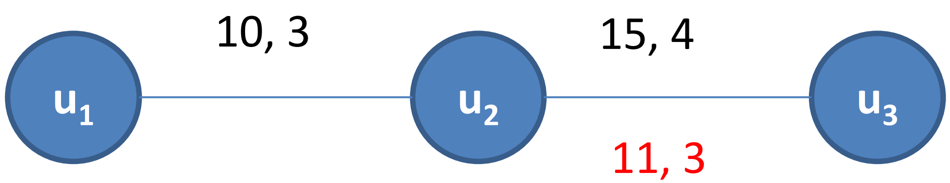
Multiple edges between two nodes (more than one interaction)

Paths in Temporal Graphs [PVLDB14]

Temporal path (must follow chronological order)

Each edge $u_i u_j$ in the path

$start(u_j) \geq end(u_i)$ $(start(u_j) \geq start(u_i) + \lambda)$



Let P be a path

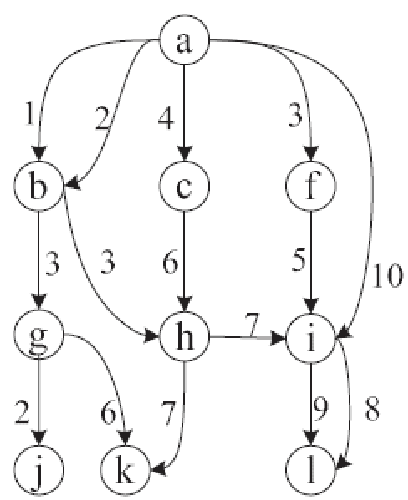
- duration(P) = end(P) – start(P)
- distance(P) = $\sum \lambda_i$

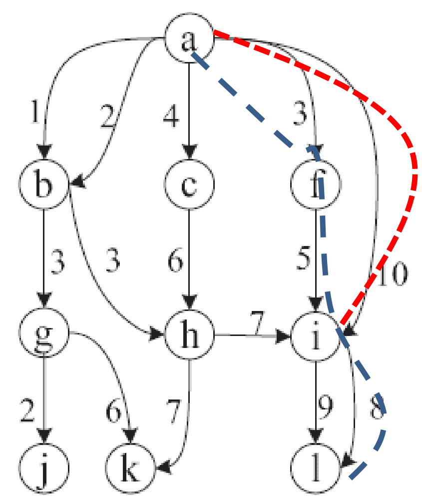
Example

a -> l

Showing starting times,

assume all durations 1





Minimum Temporal Paths [PVLDB14]

Minimum Temporal Path from u to w in interval [t1, t2]

All temporal paths P' from source *u* to target *w* in interval *[t1, t2]* with *start(P')* $\geq$ *t1* and *end(P')* $\leq$ *t2*

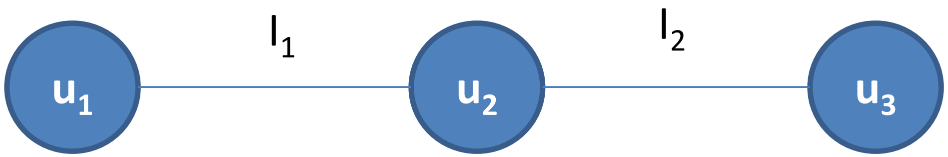
Look for path P such that

- Earliest-arrival path: end(P) = min{end(P')}
- Latest-departure path: start(P) = max{start(P')}
- Fastest path: duration(P) =min{duration(P')}
- Shortest path: dist(P) =min{dist(P')}

Temporal Paths vs Paths in Historical Graphs

- Temporal paths additional constraints to model a sequence of events or a journey
- Combine
  - Historical temporal paths
  - Most durable or historical communications

Representing Path lifespans

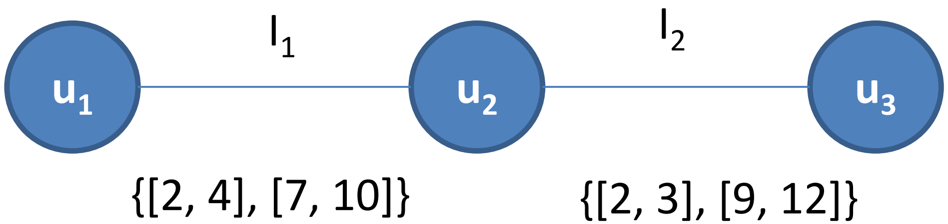


{[2, 4], [7, 10]}   {[2, 3], [9, 12]}

{[2, 4], [7, 10]} $\oplus$ {[2, 3}, [9, 12]} = {[2, 3], [9, 10]}

- Intervals as ordered list of time points, l_1 = {2, 3, 4, 7, 8, 9, 10} l_2 = {2, 3, 9, 10, 11,12}
  - Seldom connected, fast, few snapshots
- Intervals as a minimal ordered list of intervals:
  - non-overlapping, overlap [2, 7] [6, 9]
  - Non-continuous, continuous [2, 8] [9, 10]
  - very few deletes, continuous connections

Representing Path lifespans



{[2, 4], [7, 10]} $\oplus$ {[2, 3}, [9, 12]} = {[2, 3], [9, 10]}

- Using bit-arrays

l_1 = 011100111100000

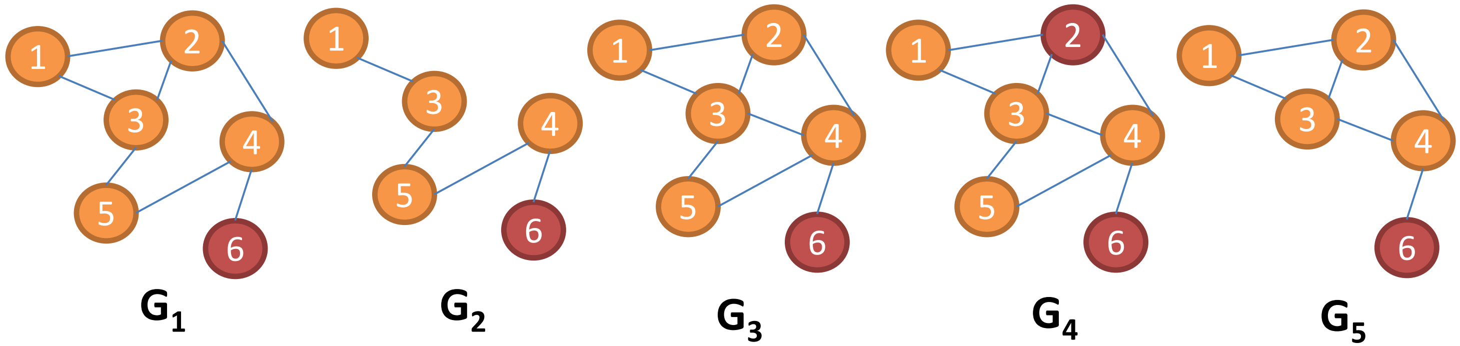
l_2 = 011000001111000

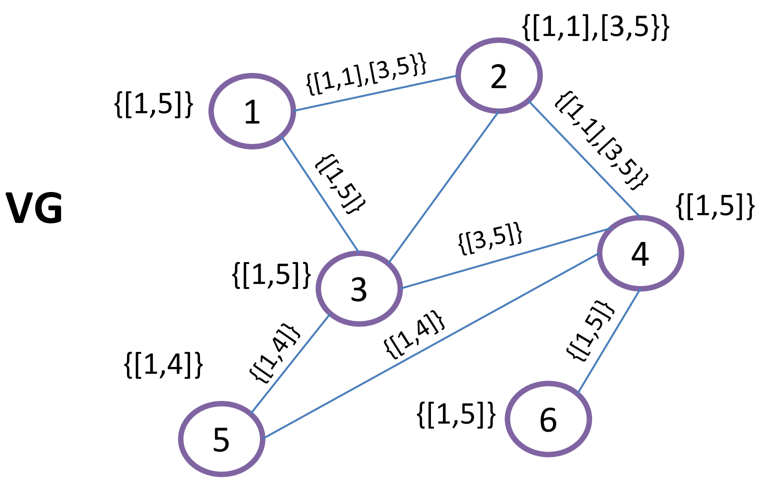
Very fast time join

011000001100000

Predefined maximum size – but can use additional arrays as time evolves

Version Graph





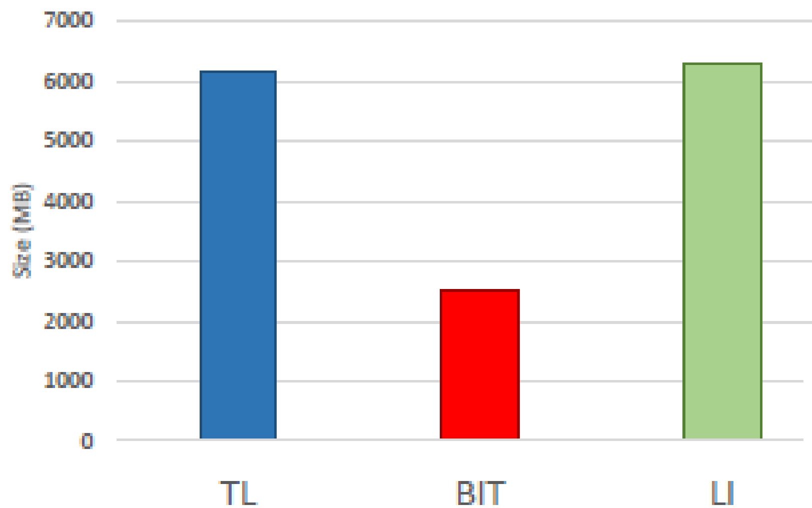
- Bit array representation
  - Example *I* = {[1, 3],[5, 10], [12, 13]}, T = 16, **1110111111011000**
- In-memory storage

Representing Path lifespans: comparison

- ordered list of time points (TL)
- minimal ordered list of intervals (TI)
- bit-arrays (BIT)

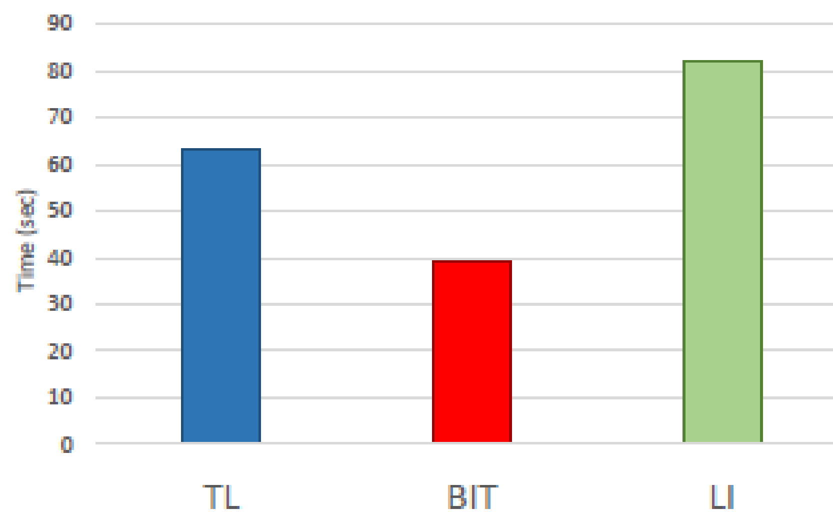
Dataset	# Nodes	# Edges
DBLP	1,026,946	4,122,070

In [1959, 2014]



Size of VG
Because most co-operations are transient

Construction time



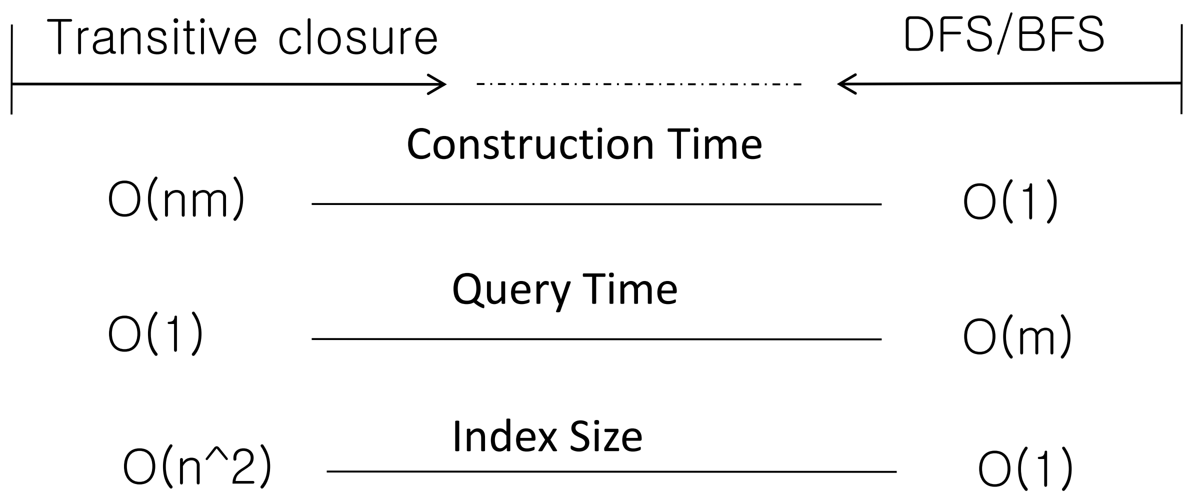
Reachability and Shortest Path Queries

Two extreme approaches

1. Online traversal of the graph
2. Pre-computation of the transitive closure (reachability) or full distance table

- In between: maintain indexes

Trade off



Navigational Queries

- Focus on
  - Shortest path (mainly on reachability queries)

Outline

- Online traversal
- Indexing

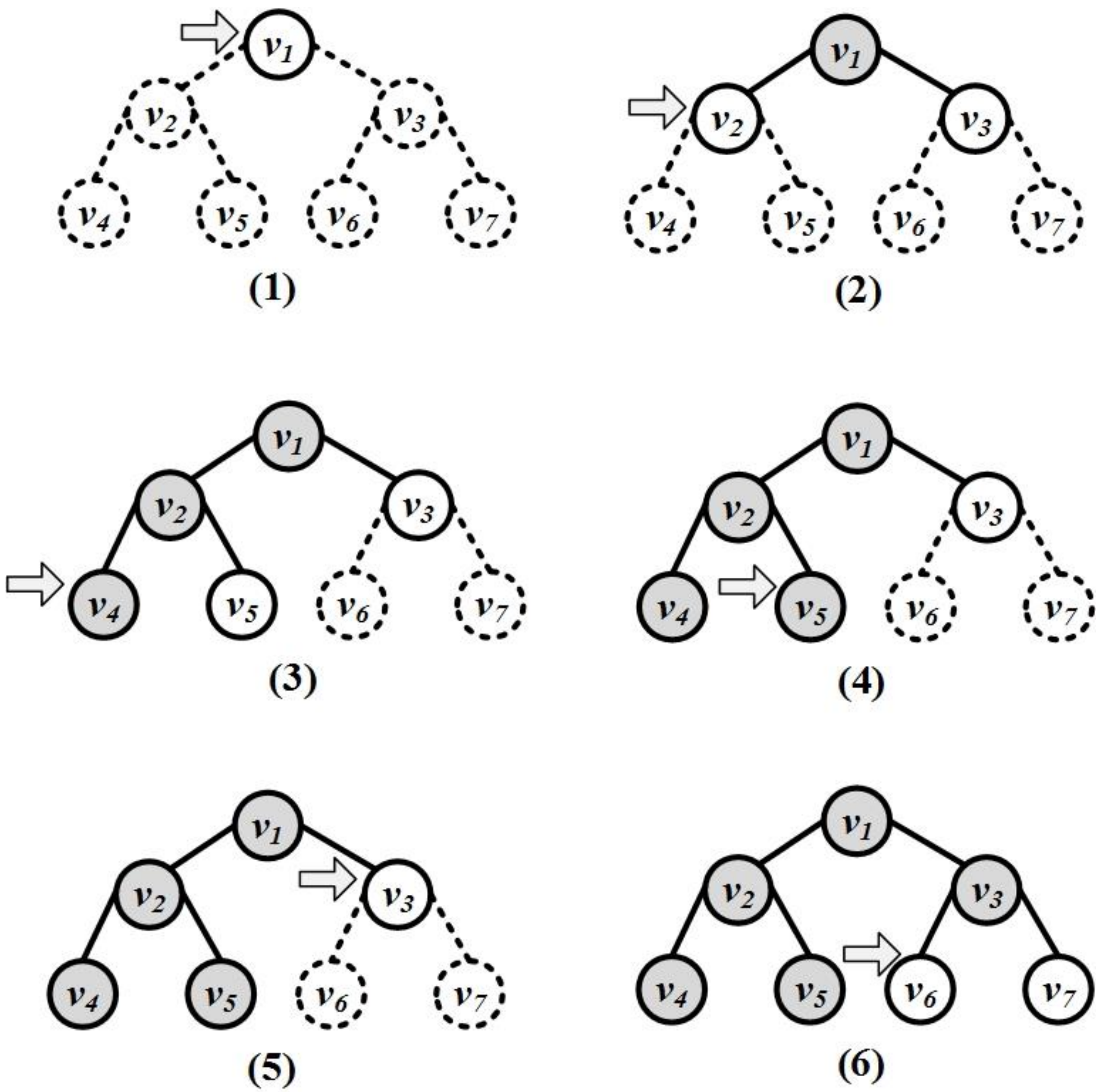
Graph Traversals (basics)

- A traversal is a procedure for visiting (going through) all the nodes in a graph
- Two basic traversals
  - DFS
  - BFS

Depth First Search Traversal (basics)

- Depth-First Search (DFS) starts from a node *i*, selects one of its neighbors *j* and performs Depth-First Search on *j* before visiting other neighbors of *i*.
  - The algorithm can be implemented using a *stack structure*

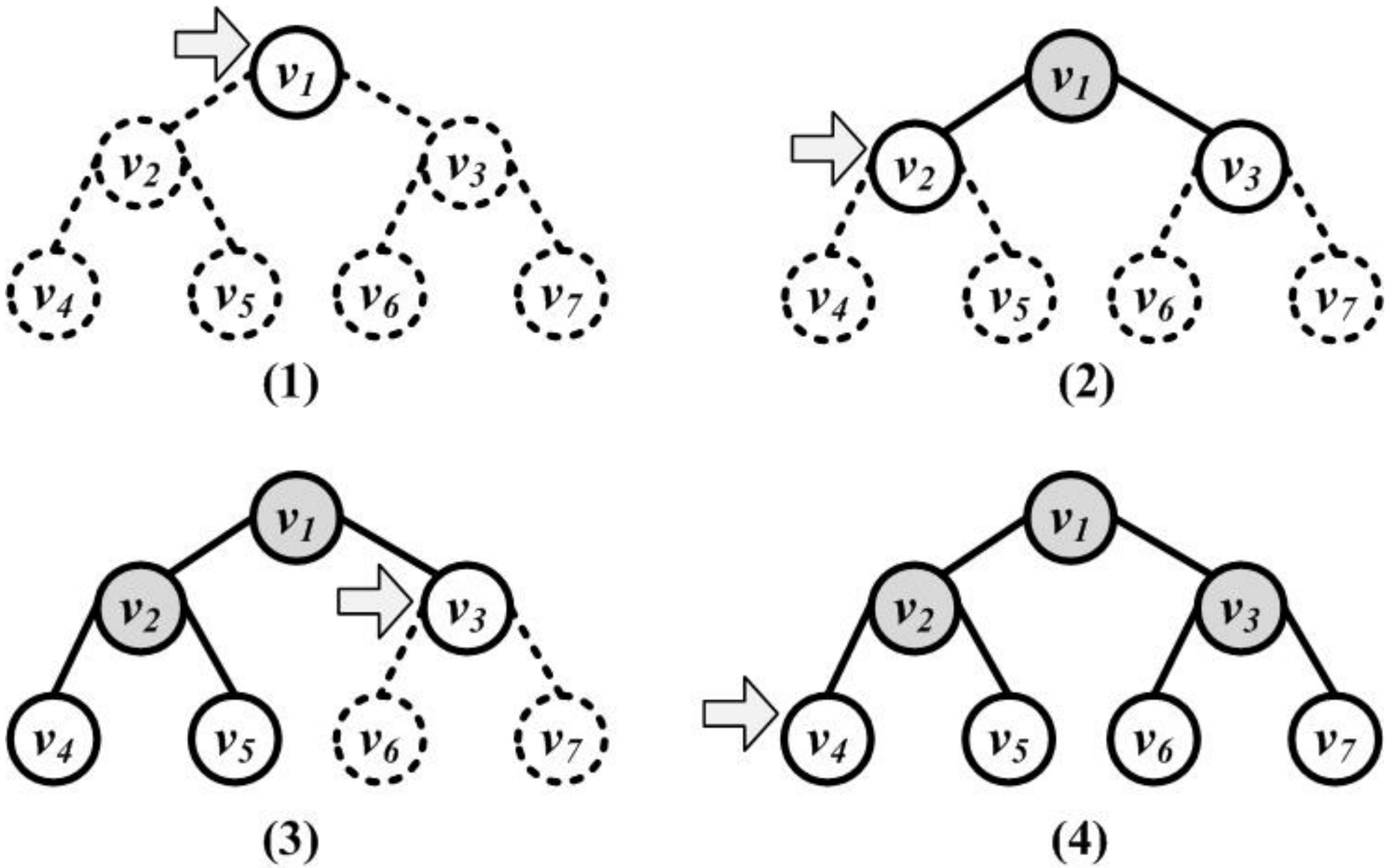
Example DFS (basics)



Breadth First Search Traversal (BFS)

- Breadth-First-Search (BFS) starts from a node, visits all its immediate neighbors first, and then moves to the second level by traversing their neighbors.
  - The algorithm can be implemented using a *queue structure*

Example of BFS (basics)



Breadth First Search Traversal (BFS)

- We can find all shortest paths from a node *w* using BFS
  - Starting from *w*, visit all neighbors of *w* at distance 1, at distance 2, etc
- We visit each node once
  - we do not have to revisit a node again, since we already have its shortest distance from the root of BFS

Breadth First Search Traversal (BFS)

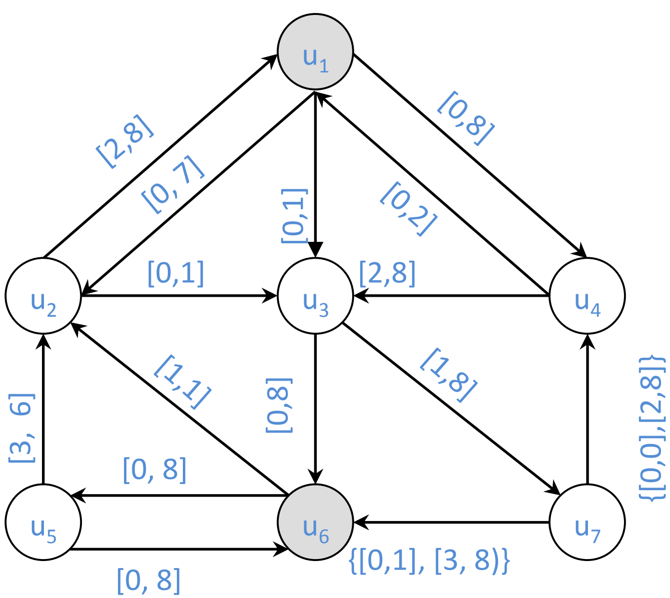
- Shortest paths on weighted graphs are harder to construct
  - There are several well known algorithms for finding single-source, or all-pairs shortest paths
  - For example: Dijkstra's Algorithm

Historical Reachability: Online BFS Traversal
[EDBT2015]

Traverse the graph once for the whole query interval I_Q

- Follow only path P whose lifespan intersects I_Q
- At each node, *maintain the lifespan of paths computed so far* (PC)
  - Pruning: never traverse a node twice for the same interval

Stop traversing when the whole query interval is covered

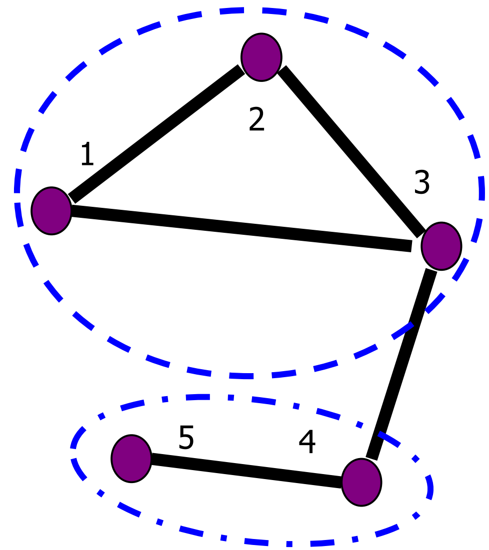


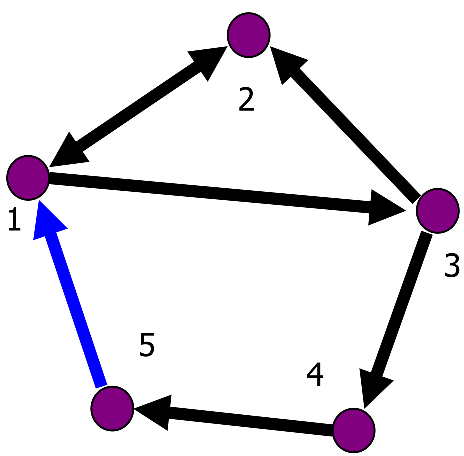
$u_1 \xrightarrow{[0,3]} u_6$

$\left.\begin{array}{l} PC_1 = [0,1] \\ PC_2 = [2,3] \end{array}\right\}$ ✓

Connected components (basics)

- *Connected* graph: a graph where there every pair of nodes is connected
- *Disconnected* graph: a graph that is not connected
- *Connected Components*: subsets of vertices that are connected





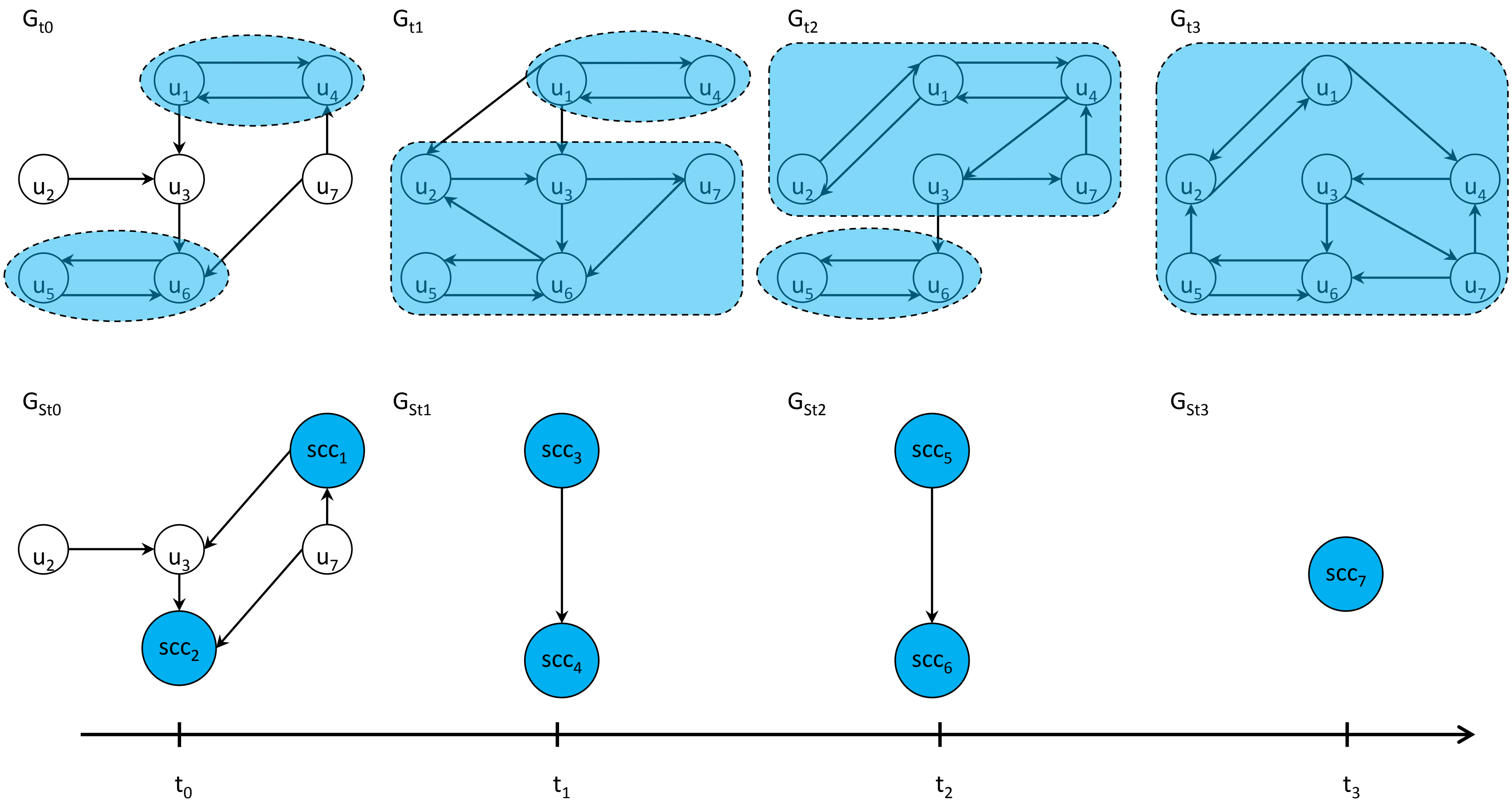
- *Strongly connected graph:* there exists a path from every i to every j
- *Weakly connected graph:* If edges are made to be undirected the graph is connected

TimeReach Index

- Many real-world graphs consist of *large strongly connected components (SCC)*
  - Nodes in the same SCC are reachable
  - It suffices to maintain node-SCC participation and inter-SCC reachability information in each snapshot

- For each snapshot G_i
  - Identify SCCs
  - Construct condensed graph $G_{Sti}(V_{Sti}, E_{Sti})$
  - Store node-SCC participation (node-SCC list)

TimeReach Index: Construction



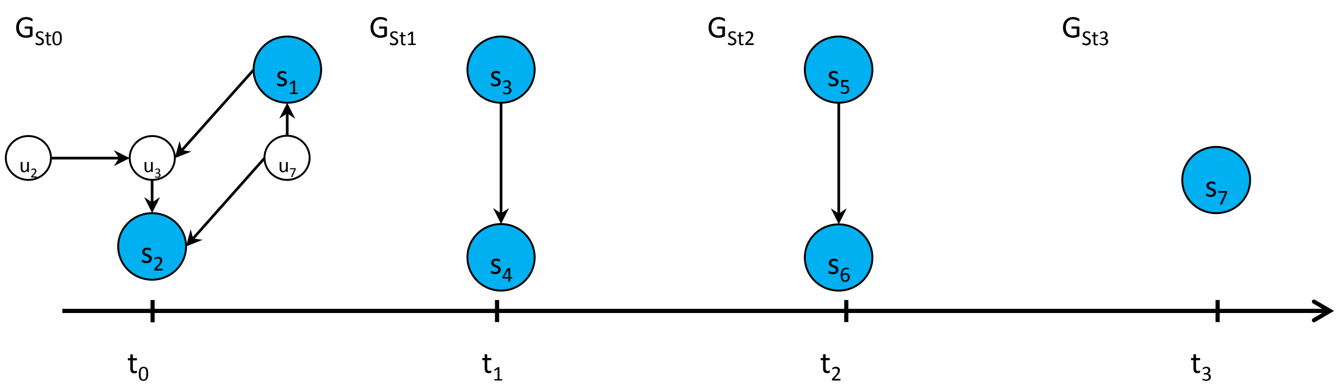
TimeReach Index: Construction

- Query for u,v and interval I_Q
  - For each t in I_Q check if u and v belong to the same SCC
  - Otherwise traverse the corresponding condensed graph(s)

$u_1 \xrightarrow{[0,3]} u_6$

Node-SCC list

	t_0 ✓	t_1 ✓	t_2 ✓	t_3 ✓
u_1	s_1	s_3	s_5	s_7
u_2	u_2	s_4	s_5	s_7
u_3	u_3	s_4	s_5	s_7
u_4	s_1	s_3	s_5	s_7
u_5	s_2	s_4	s_6	s_7
u_6	s_2	s_4	s_5	s_7
u_7	u_7	s_4	s_6	s_7



TimeReach Index: Construction

- Efficiency
  - Fast incremental construction (using Tarjan's algorithm [1])
    - Identify and condense each snapshot
  - Significantly smaller storage than Transitive Closure
  - Faster query processing than Online Traversal
    - From the list or traversal of small condensed snapshots

- Can we do better?

TimeReach Index: Compression

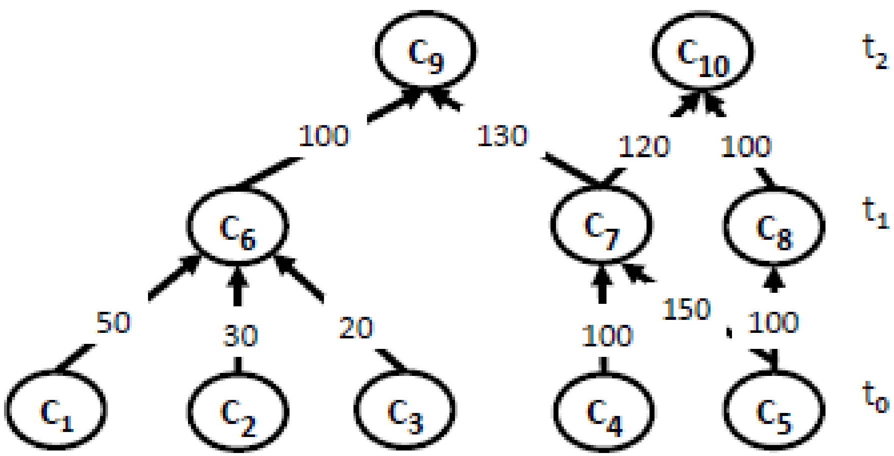
- Speed-up traversals
  - Construct condensed version graph
  - Interval based traversal of the condensed graph
- Compress the node-SCC list
  - Replace the list with per node SCC-postings → (SCC-id, time-interval) pairs
  - Minimize the total number of postings
- How to minimize the number of postings?
  - A new posting is created when a node is associated with a different SCC-id → RE-ASSIGN IDs

Nodes	Posting List
1-50	$(C_1,t_0),(C_6,t_1),(C_9,t_2)$
51-80	$(C_2,t_0),(C_6,t_1),(C_9,t_2)$
81-100	$(C_3,t_0),(C_6,t_1),(C_9,t_2)$
101-200	$(C_4,t_0),(C_7,t_1),(C_9,t_2)$
201-230	$(C_5,t_0),(C_7,t_1),(C_9,t_2)$
231-350	$(C_5,t_0),(C_7,t_1),(C_{10},t_2)$
351-450	$(C_5,t_0),(C_8,t_1),(C_{10},t_2)$

TimeReach Index: Compression

Basic idea for reassigning IDs (mapping conponents)

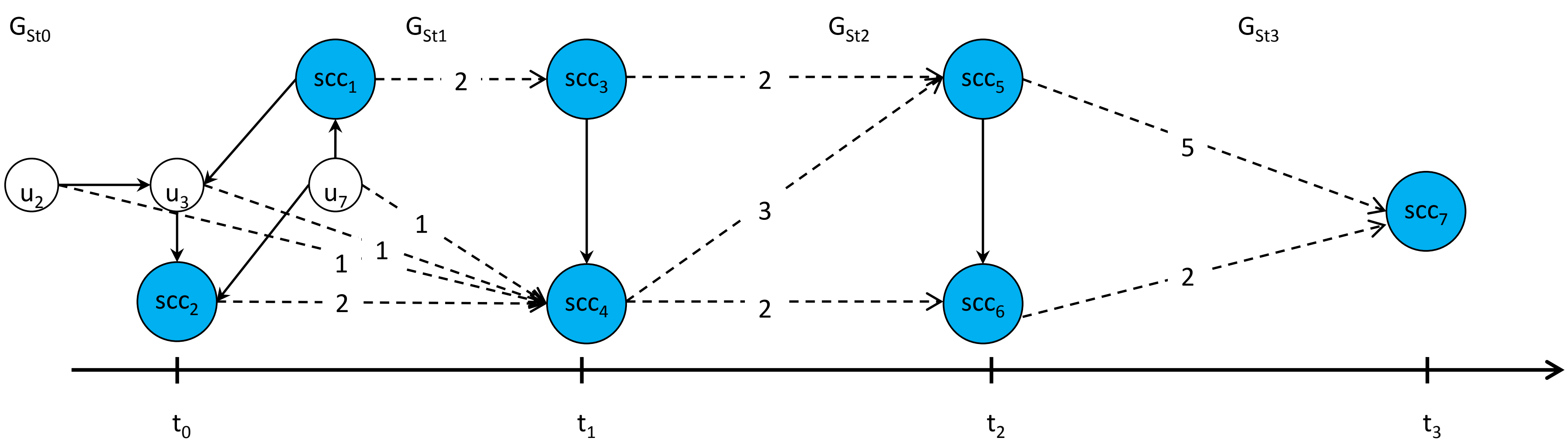
- Model SCC evolution using *a weighted graph*
- Each node corresponds to a SCC that existed at some time t
- An edge connects two nodes if the corresponding SCCs have at least a common node
- The weight of edge (U,V) equal to the number of nodes *in both* U, V



TimeReach Index: Compression

We model SCC evolution using a weighted graph $G_C(V_C, E_C, W_C)$

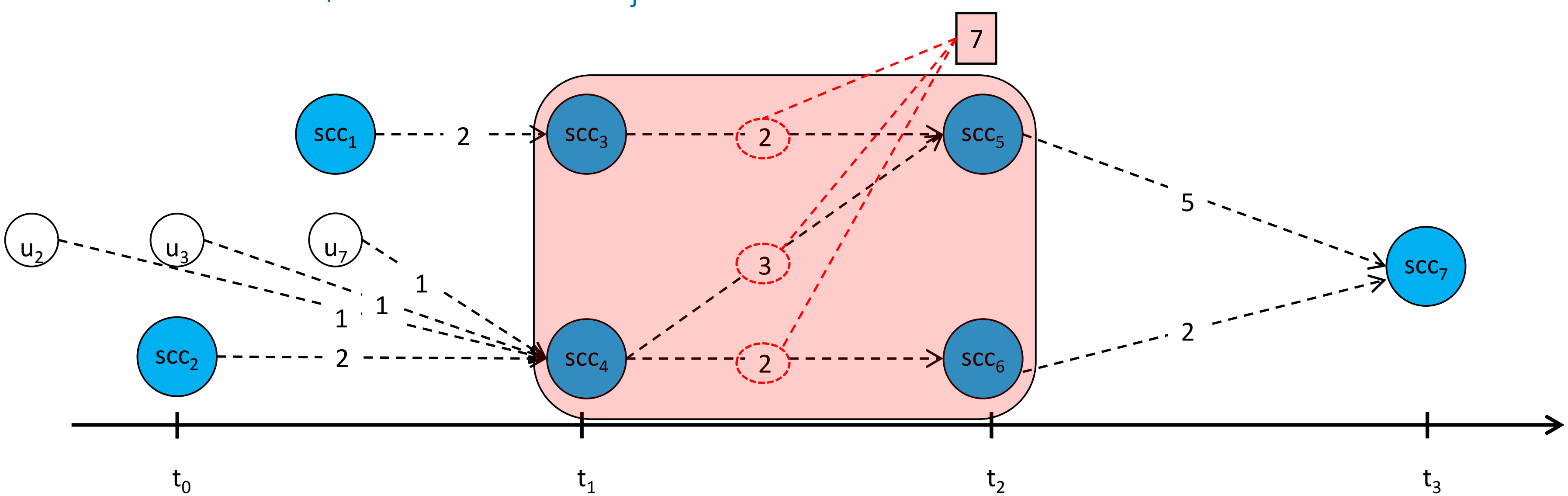
- Each node corresponds to a SCC that existed at some time t
- An edge connects two nodes if the corresponding SCCs have at least a common node
- W assigns to edge (U,V) weight equal to the nodes *in both* U, V



TimeReach Index: Compression

$G_C(V_C,E_C,W_C)$ is an |T|-partite graph

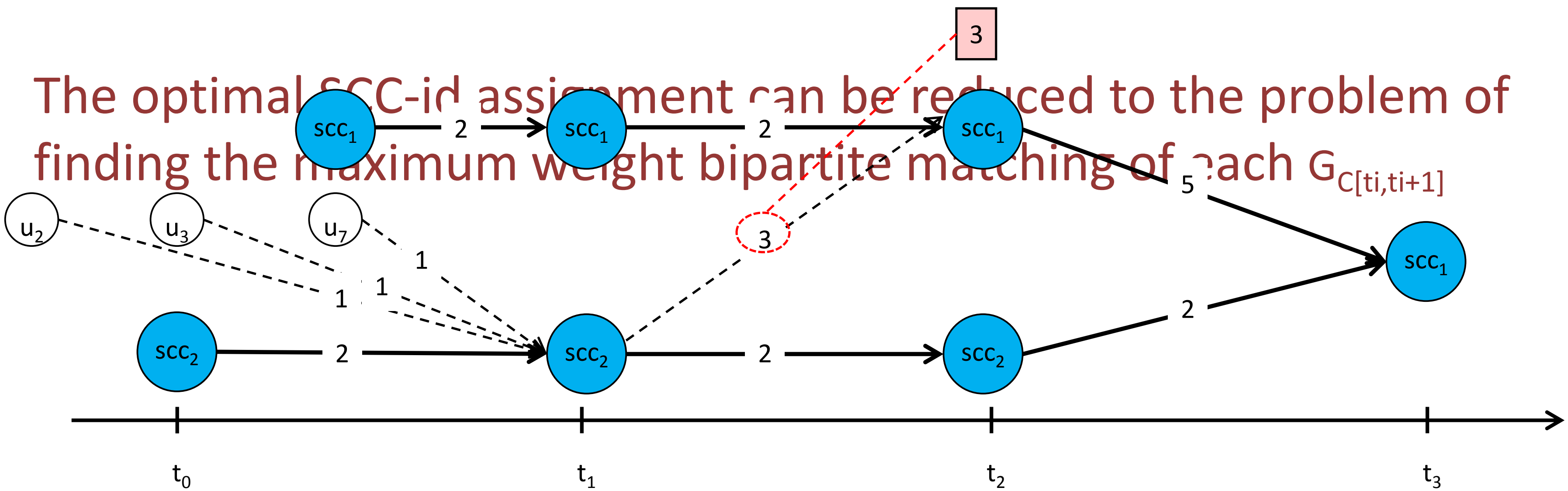
- Each subgraph $G_{C[ti,\ ti+1]}$ corresponding to two consecutive time instants is a bipartite graph
- The number of new postings for time t → the sum of weights from nodes U_i at level $t-1$ to V_j at level t with different ids



TimeReach Index: Compression

The optimal SCC-id assignment can be reduced to the problem of finding the maximum weight bipartite matching of each $G_{C[ti,ti+1]}$

The optimal SCC-id assignment can be reduced to the problem of finding the maximum weight bipartite matching of each $G_{C[ti,ti+1]}$



TimeReach Index: Compression

Incremental algorithm

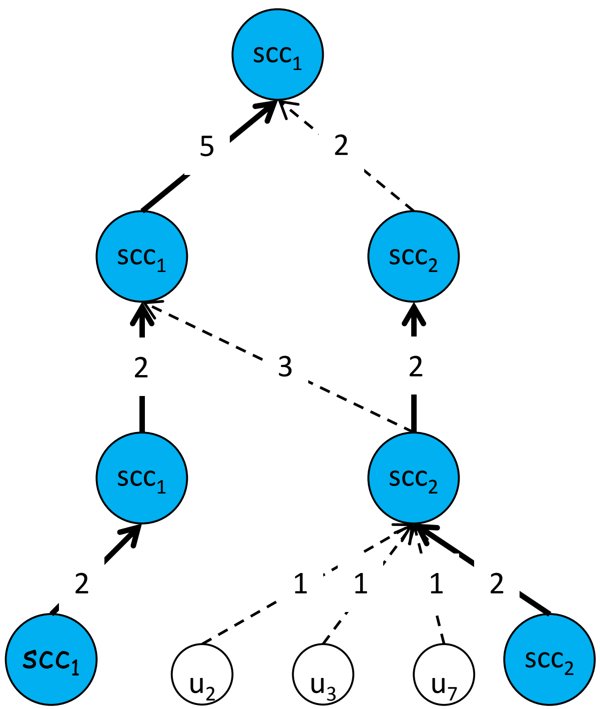
- Compute SCCs in current snapshot G_t
- Construct bipartite graph $G_{C[t-1,t]}$
- Compute maximum weight bipartite matching of $G_{C[t-1,t]}$
- Use the computed maximum weight bipartite matching to assign ids to SCCs
- Update the SCC postings created at time $t-1$
  - Create new entry only for nodes that change SCC-id

TimeReach Index: Processing

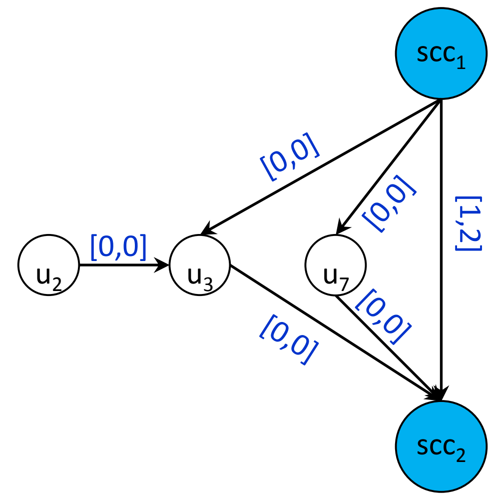
Two steps

- Retrieve the SCC postings of u and v: if they belong to the same SCC during I_Q we are done
- Otherwise
  - Split the query based on the postings
  - Answer subqueries from the postings or by interval based traversal of the condensed version graph
  - Combine the results

TimeReach Index: Processing



u_1	$(s_1,[0,inf)$
u_2	$(u_2,[0,0]),(s_2,[1,1]), (s_1,[2,inf))$
u_3	$(u_3,[0,0]),(s_2,[1,1]), (s_1,[2,inf))$
u_4	$(s_1,[0,inf))$
u_5	$(s_2,[0,2]),(s_1,[3, inf))$
u_6	$(s_2,[0,2]),(s_1,[3, inf))$
u_7	$(u_7,[0,0]),(s_2,[1,1]), (s_1,[2,inf))$



Conjunctive query $Q_{[0,3]}u_1 \rightarrow u_6$

Split query

Lookup postings

$Q_{[0,2]}S_1 \rightarrow S_2$: traversal of VG $\rightarrow$ true

$Q_{[3,3]}S_1 \rightarrow S_1 \rightarrow$ true

Navigational Queries

Outline

- Online traversal
- Indexing
  - reachability
    - label the nodes, look at the labels to decide reachability
    - we will look into one 2hop reachability index
  - distance

Reachability Index (static)

Compact form of the transitive closure

	u_1	u_2	u_3		u_n
u_1	1	0	1		0
u_2	0	1	0		0
u_3	0	1	0		0
u_n	1	0	1		1

For each pair of nodes whether they are reachable or not

2-Hop Labeling (static)

Labels – set of nodes

For each node u, maintain two sets of labels (nodes):

Lout(u): a set of nodes reachable from u and

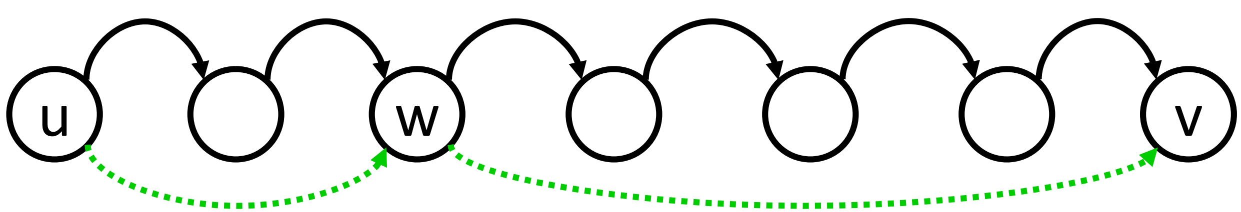
w in Lout(u): there is a path u → w

Lin(u): a set of nodes from which u is reachable

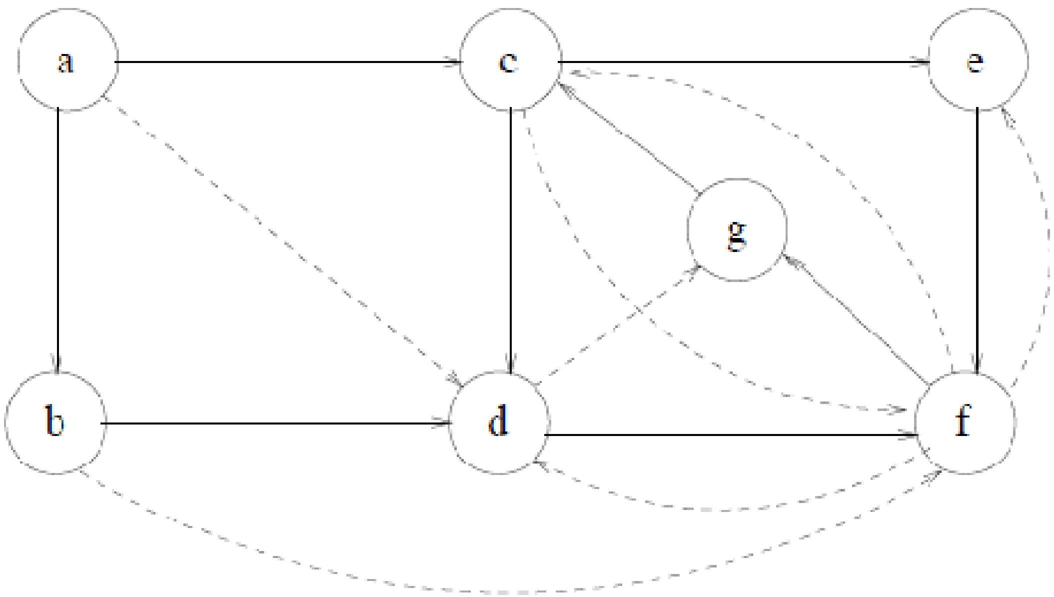
w in Lout(u) – there is a path w → u

To test whether a v is reachable from u (there is a path u → v), check **Lout(u) ∩ Lin(v)≠∅** (path u → w → v)

2-Hop cover is set of hops (x, y) so that every connected pair is covered by 2 hops [SODA2002]



2-Hop Labeling (static)



v	$L_{in}(v)$	$L_{out}(v)$
a		b, c, d
b		f, d
c	f	d, f
d	f, c	g, f
e	c, f	f
f	d, c	g, c, e
g	d, f	c

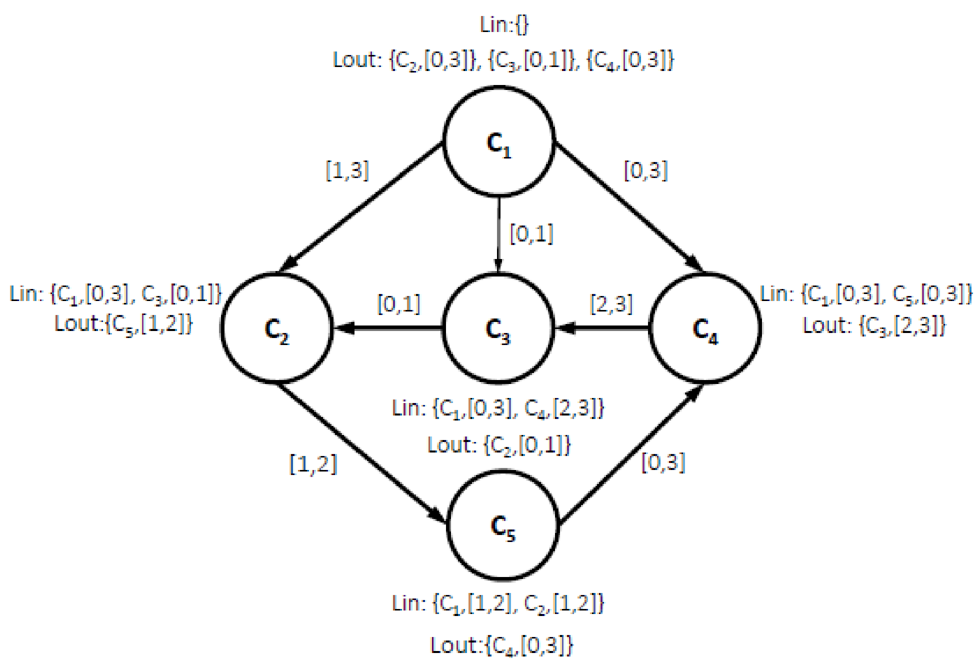
a → *f?*
c → *b?*

Figure from SODA02 (dashed edges not graph edges)

Indexing (historical)

Simple solution

- Compute 2hop cover for each instance
- Augment labels with lifespans



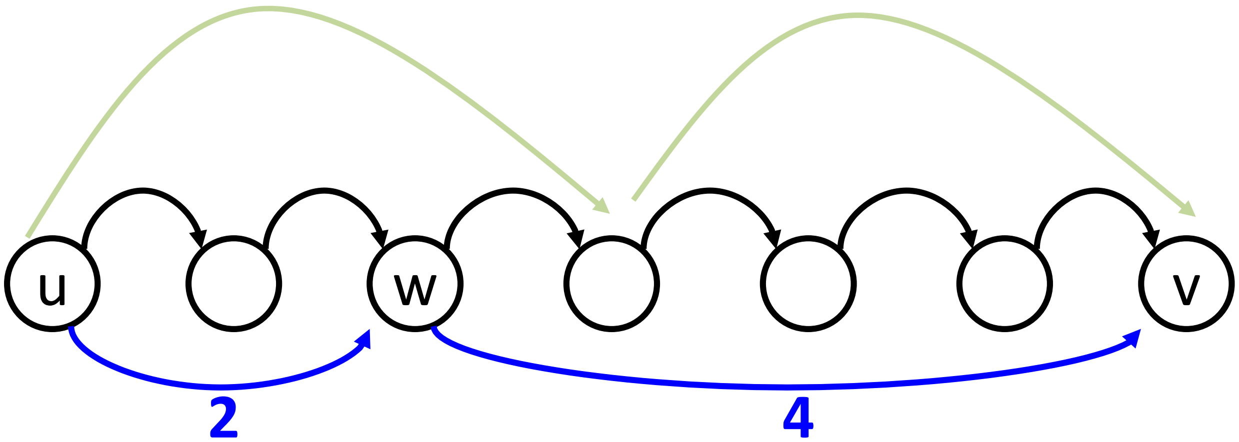
Distance Index (static)

Full distance matrix

	u_1	u_2	u_3		u_n
u_1	0	-	5		-
u_2	-	0	0		-
u_3	-	2	0		-
u_n	4	-	2		0

Can we just augment the 2HOPs with distance information?

Distance Index (static)



For each pair of nodes v and w, at least one node in their shortest path must be included in Lout(u) and Lin(v) - landmarks

We compute the distances (sum) for all landmarks and maintain the smallest one

Vary few papers on shortest paths

Incrementally update 2hops

T. Akiba, Y. Iwata, Y. Yoshida, *Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling*, WWW 2014

T. Hayashi, T. Akiba, K. Kawarabayashi: *Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks.* CIKM 2016: 1533-1542

Dijkstra online traversal

W. Huo, V. Tsotras, *Efficient temporal shortest path queries on evolving social graphs*, SSDBM 2014

FVF

C. Ren, E. Lo, B. Kao, X. Zhu, R. Cheng, DW Cheung *Efficient Processing of Shortest Path Queries in Evolving Graph Sequences*, Information Systems, Available online 7 June 2017

Navigation (summary)

- Many interesting problems
  - Labels for historical graphs
  - Durability
  - Evolution
  - Labeled or property paths
    - Constraints on the labels/properties
    - Time-varying properties

Graph Pattern Queries

Pattern Matching

Labeled graphs

Input: *Graph G*(V, E, L), L: V → Σ*

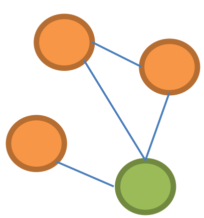
Pattern P(V_P, E_P, L_P)

Output: *Subgraphs* $m = (V_m, E_m, L_m)$ of G, such that, there exists a *bijective function* f : $V_p \rightarrow V_m$:

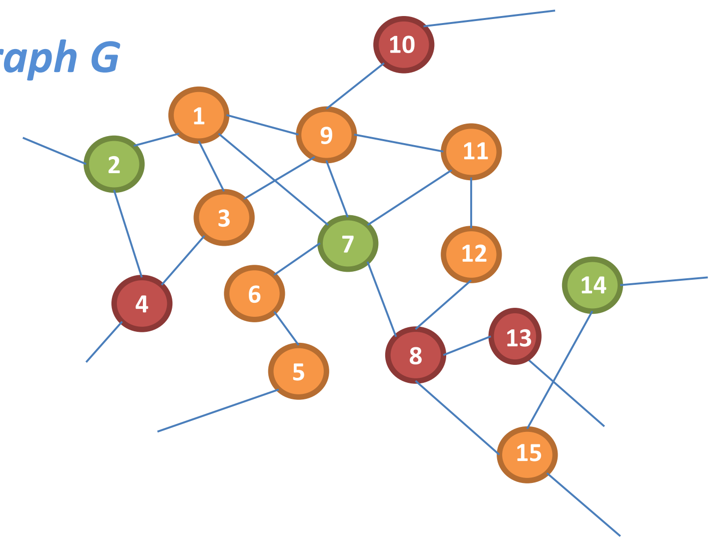
- for all u in V_P, $L_p(u) \in L_m(f(u))$ and
- for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$

Graph m is called a *match* of P in G

Pattern P



Graph G



Pattern Matching

Labeled graph

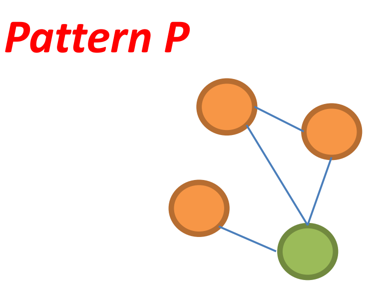
Input: ***Graph G***(V, E, L), $L: V \rightarrow \Sigma^*$

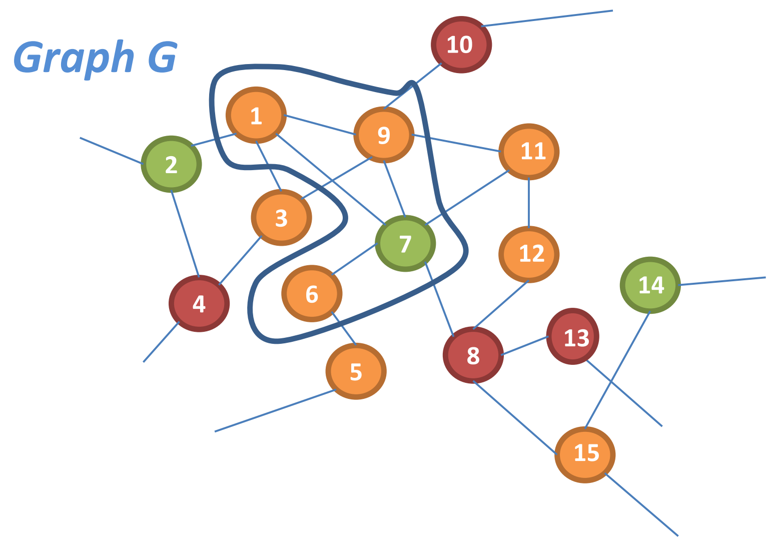
Pattern P(V_P, E_P, L_P)

Output: ***Subgraphs*** $m = (V_m, E_m, L_m)$ of G, such that, there exists a *bijective function* $f : V_p \rightarrow V_m$:

- for all u in V_P, $L_p(u) \in L_m(f(u))$ and
- for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$

Graph m is called a ***match*** of P in G





Pattern Matching

Labeled graph

Input: *Graph G*(V, E, L), $L: V \rightarrow \Sigma^*$

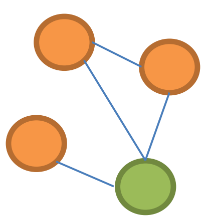
Pattern P(V_P, E_P, L_P)

Output: *Subgraphs* $m = (V_m, E_m, L_m)$ of G, such that, there exists a *bijective function* $f : V_p \rightarrow V_m$:

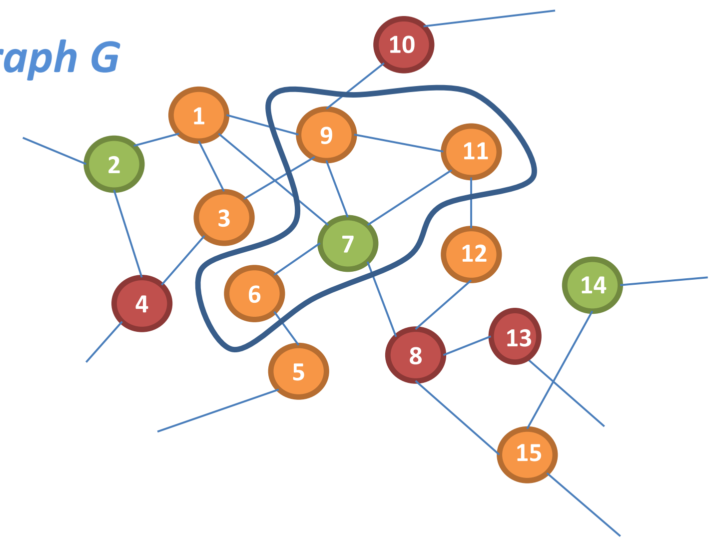
- for all u in V_P, $L_p(u) \in L_m(f(u))$ and
- for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$

Graph m is called a *match* of P in G

Pattern P



Graph G



Related Work

- *(sub) graph isomorphism*, NP complete

Large body of work:

- Most work *many small graphs:* identify the ones with (at least) one match (aka *graph containment*, *graph retrieval*) – *we consider a single large graph*
- Various algorithms:
  - Most *graph indexes* (based on features such as paths, trees, neighbors, sub-graphs, etc)
  - Often, a two phase approach
    - *filter-and-verify*: in the first phase use graph index to generate *candidate matches* and then in the second phase *verify* them using some form of graph *isomorphism search*
    - *decompose-and-(multi-way join):* in the first phase *decompose* into subgraphs and use the index to find matches and then *join* the results

Durable Graph Patterns: definitions [ICDE16]

Given a sequence of graph snapshots ***G*****,** a pattern ***P*****,** and a set of time intervals ***I***, find the most durable matches: the matches that exist for the ***largest time period*** of time during ***I***

Two interpretation for the duration of a set of time intervals ***I***

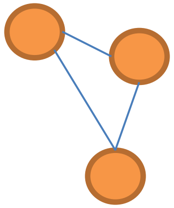
- *collective duration*: the number of time instants in ***I***
- *continuous duration*: the duration of the longest time interval in ***I***

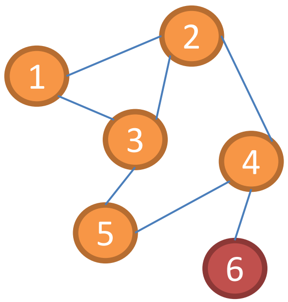
Example ***I*** **= {[1, 3],[5, 10], [12, 13]}** – Collective: **11**, Continuous: **6**

(Durable Graph Pattern Matching): Two types:

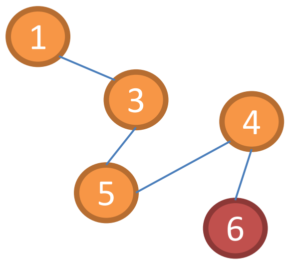
- *collective-time durable graph pattern query*
- *continuous-time durable graph pattern query*

Example

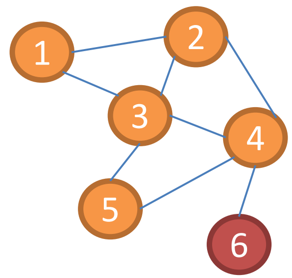




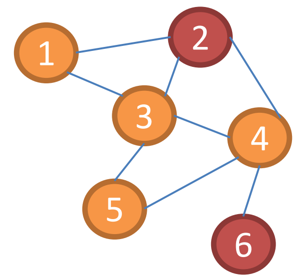
G_1



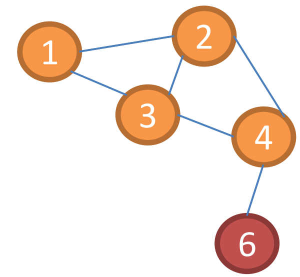
G_2



G_3

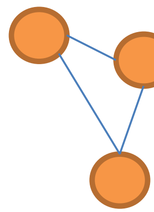


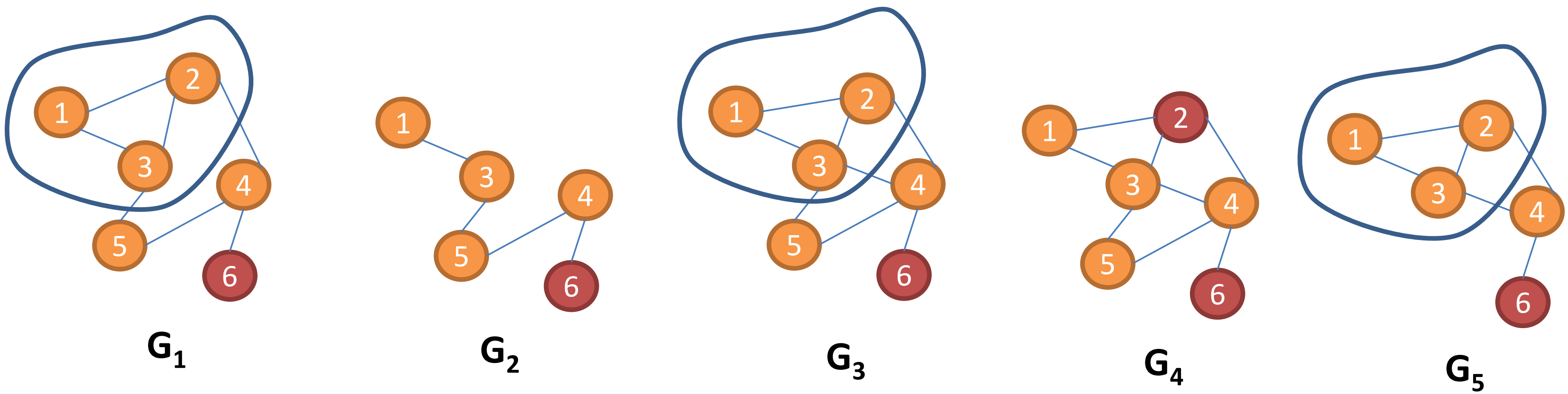
G_4



G_5

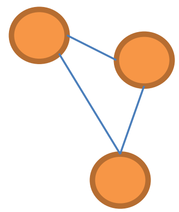
Example

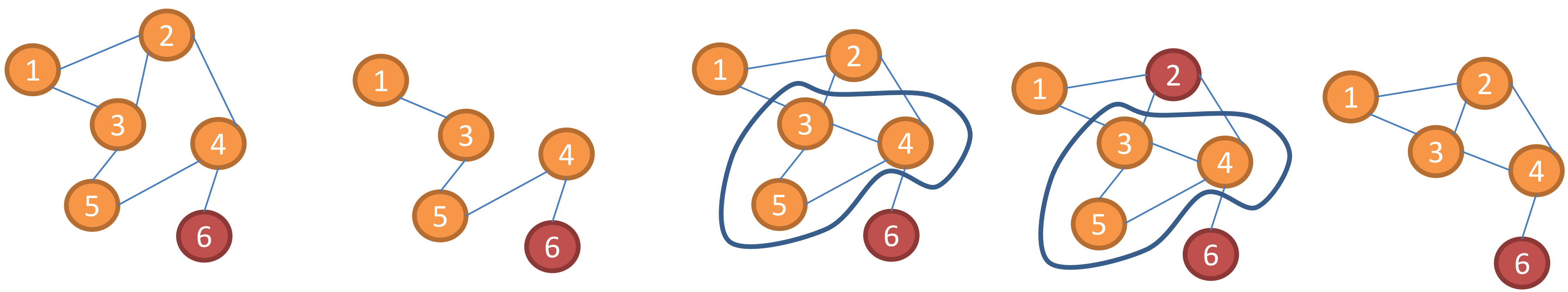




Collective: 3
Continuous: 1

Example





G_1 G_2 G_3 G_4 G_5

Collective: 2
Continuous: 2

Durable Graph Patterns: applications

- In collaboration or social networks*:* most persistent research collaborations, friendships, interactions
- In a protein network, the protein complex that is durable through the evolution
- In a large biological network, the durable chain of nucleotides of virus RNA for predicting which genes are prone to mutations.
- In marketing, identify for a product, an idea or a person, the durable patterns of supporters among specific demographic groups labeled by their age, location or other characteristics.

Baseline 2P algorithm

- Find the matches at *each snapshot*
- Return the matches with the *most appearances* (for efficiently identifying which matches are the same, represent subgraphs as strings and do string matching)

- expensive, since we have to *retrieve all matches at each graph snapshot*, even those matches that appear only in just one snapshot
- for *frequent patterns* and *long intervals*, the number of retrieved matches grows very fast (more than 24h for 1M nodes, 4M edges)

Durable Graph Pattern

Filter-and-Verify algorithm based on:

1. Version Graph representation of the snapshot sequence
2. Graph Time Indexes
3. ϑ-duration threshold

Durable Pattern Match (outline)

Input: Version graph ***VG***, pattern ***P***, set of intervals ***I***
Output: Most durable matches ***M***

```
ϑ ← 1; M ← {}
for each node p in  the pattern P do
     C(p) ← FILTERCANDIDATES( … )
     if C(p) = {} ; then return {}
C ← REFINECANDIDATES(…)
DURABLEGRAPHSEARCH(VG, ϑ, …)
return M
```

FILTERCANDIDATES:

- locate *candidate* matching nodes for each node in the pattern using *time indexes*.

REFINECANDIDATES:

- *refine* candidate sets using the *VG* and *time indexes*.

DURABLEGRAPHSEARCH:

- Search *VG* to verify for matches with duration *at least ϑ* *(dual graph simulation)* performing also "time-joins"
- Each time a match is found, ϑ is increased

Indexes

Time-label or **TiLa** index (basic index)

- Given a label ***l*** and a time instant ***t***: constant time retrieval of all nodes having label ***l*** at ***t***

First level: Array of size *T* where each position *i* refers to a time instant *i* and links to a set of labels *L*.
Second level: Each label *l* in this set links to the set of nodes that are labeled with *l* at *i*.

Time-path-label or **TiPLa** index (parameter λ)

- As TiLa but for labeled paths:
  Given a label path ***p*** and a time instant ***t***: constant time retrieval of all starting nodes of path ***p*** at ***t***

TiPLa enumerates all paths up to a maximum length *($\lambda = 2$)*

Indexes

Time-neighborhood-label or **TiNLa(r)** index

- For each node ***u*** information about the labels of its neighbors at distance ***r***, i.e., nodes ***r*** hops away from **u**

For each node u, a bit array of size L, where each position is a bit array of size T, where

$$Position(i) = \begin{cases} 1, & if\ node\ at\ distance\ r\ with\ label\ l\ at\ time\ i \\ 0, & otherwise \end{cases}$$

Counter time-neighborhood-label or **cTiNLa(r)** index

- Maintains the number of neighbors with the specific label

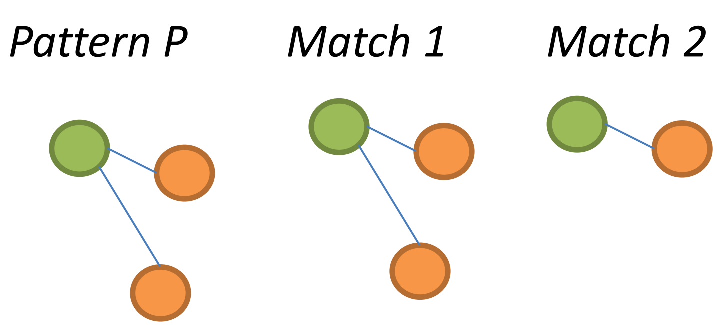
$$Position(i) = number\ of\ nodes\ at\ distance\ \ r\ with\ label\ l\ at\ time\ i$$

Candidate Nodes

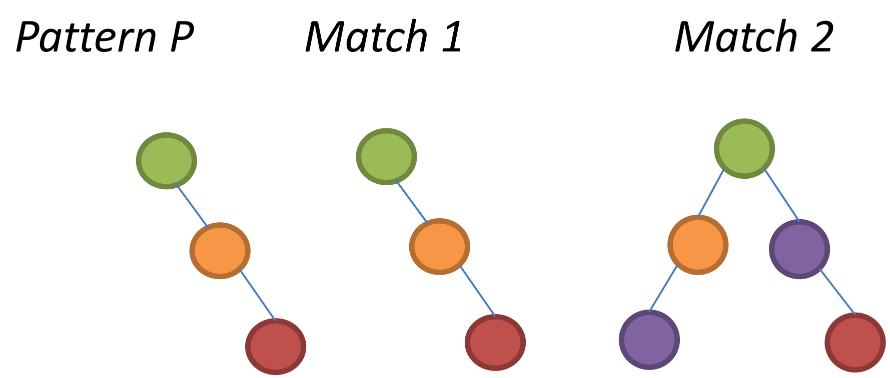
The indexes are used in FILTERCANDIDATES and DURABLEGRAPHSEARCH

$selectivity(\text{TiPLa}) \succcurlyeq selectivity(\text{TiNLa}) \succcurlyeq selectivity(\text{TiLa})$

($\succcurlyeq$: better)



$selectivity(\text{cTiNLa(1)}) \succcurlyeq$
$selectivity(\text{TiPLa})\ (\lambda = 1)$



$selectivity(\text{TiPLa})\ (\lambda = 2) \succcurlyeq$
$selectivity(\text{cTiNLa(1)} + \text{cTiNLa(2)})$

The ϑ-threshold

Simple threshold: Search with all matches with duration at least $\vartheta = 1$

- In the first runs, the algorithm considers edges that have a *short duration compared to the actual duration* of a potential match (poor pruning)

Use the indexes to estimate the duration of the match

- For a node ***p*** in the pattern ***P***,
  - *Rank*$^{\theta}$*(**p**)* = list of candidates matches with duration *at least* θ
  - d(***p***) = maximum duration for which ***p*** has *at least one match* (i.e., *Rank*$^{\theta}$*(**p**)* is not empty)
- Define $\vartheta_{max} = \min_{p \,\epsilon\, P}\{d(p)\}$
  - This is the *maximum possible duration* of a match

The ϑ-threshold

Search for matches with duration ϑ_{max}
If no match, search with a smaller ϑ

Next ϑ

- *Binary search*
- *MinMax search:* estimate the next possible maximum ϑ using the indexes as before

Evaluation (comparison with baseline)

			Collective (sec)		Continuous (sec)	
Dataset	Label value	Q. Size	Baseline	CTINLA(1)	Baseline	CTINLA(1)
DBLP	BEGINNER	2	>5,400	22	>5,400	17.63
DBLP	BEGINNER	3	>5,400	32.18	>5,400	25.96
DBLP	BEGINNER	4	>5,400	42.70	>5,400	34.74
DBLP	PROF	2	22	0.06	20.69	0.05
DBLP	PROF	3	6.78	0.08	6.82	0.08
DBLP	PROF	4	12	0.31	91.33	0.18
YT_{10}	MOST	2	>5,400	7.89	>5,400	8.23
YT_{10}	MOST	3	>5,400	11.87	>5,400	16
YT_{10}	MOST	4	>5,400	28.9	>5,400	18.31
YT_{10}	LEAST	2	91.80	0.96	91.81	1.03
YT_{10}	LEAST	3	110.63	110.63	110.63	1.82
YT_{10}	LEAST	4	157.68	2.12	157.68	2.33

Evaluation

- Overall, MINMAX outperforms BINARY
  - BINARY ordering reduces the threshold at each step *in half* often producing values far below the actual duration thus creating *large candidate sets in each step*

- SIMPLE works only when *candidate size is small* and durable matches have *short durations*

Example results with conference labels

- Assign labels based on conferences - looks for author cliques with the same conference

collective

Cliques	Size 2		Size 3		Size 4		Size 5		Size 6	
Conferences	**Duration**	**Matches**	**Duration**	**Matches**	**Duration**	**Matches**	**Duration**	**Matches**	**Duration**	**Matches**
SIGMOD	11	1	5	24	5	24	3	1000	3	1000
ICDE	8	1	5	6	3	72	2	1000	2	1000
VLDB	10	1	6	6	3	1000	3	1000	3	1000
EDBT	4	4	3	6	2	288	2	240		
KDD	9	4	6	18	5	24	3	840	3	720
WWW	9	1	5	12	3	48	2	600		
CIKM	6	4	5	6	2	1000	2	1000	2	1000
SIGIR	8	6	6	12	5	360	5	720	5	720
FOCS	8	1	3	6	2	24				
STOC	8	2	9	6	2	120				
SODA	6	5	3	18	2	240	2	120		
ICALP	5	4	4	6	2	96				
OSDI	4	2	2	132	2	144	2	120		
SOSP	4	1	3	6	2	72				
USENIX	5	1	3	48	3	24	2	1000	2	1000
SIGCOMM	6	1	3	36	3	24	2	1000	2	1000
SIGMETRICS	6	4	4	12	3	24	2	240		
SIGOPS	3	6	2	42	2	24				
SIGGRAPH	8	2	5	18	4	168	4	120	3	1000

- "database" conferences – larger & most durable cliques SIGMOD, VLDB > ICDE > EDBT
- Large cliques SIGIR (durable) cliques KDD
- "theory" conference smaller cliques

Example authors' "cliques" (collective)

	Duration	Matches	Authors
SIGMOD	11	1	Beng Chin Ooi, Kian-Lee Tan
VLDB	10	1	Kian-Lee Tan, Beng Chin Ooi
WWW	9	1	Min Zhang, Yiqun Liu
KDD	9	4	Martin Ester, Hans-Peter Kriegel | Jiawei Han, Philip S. Yu | Jiawei Han, Xifeng Yan | Wei Fan, Philip S. Yu
STOC	8	2	Eyal Kushilevitz, Rafail Ostrovsky | Yossi Azar, Baruch Awerbuch
FOCS	8	1	Oded Goldreich, Shafi Goldwasser
ICDE	8	1	Divyakant Agrawal, Amr El Abbadi
SIGGRAPH	8	2	Takuji Narumi, Tomohiro Tanikaw | Andrew Jones, Paul E. Debevec
SIGCOMM	6	1	Albert G. Greenberg, David A. Maltz
SODA	6	5	Leonidas J. Guibas, John Hershberger | Constantinos Daskalakis, Ilias Diakonikolas | Alexandr Andoni, Piotr Indyk | Esther M. Arkin, Joseph S. B. Mitchell | Fedor V. Fomin, Daniel Lokshtanov
USENIX	5	1	Christopher Kruegel, Engin Kirda
SOSP	4	1	M. Frans Kaashoek, Eddie Kohler

“Combining” Conference

Combinations	Duration	Matches
WWW-SOSP		
WWW-CIKM	5	1
WWW-STOCS	3	3
WWW-SIGGRAPH	3	2
WWW-EDBT	6	3
CIKM-USENIX	2	8
CIKM-SIGIR	6	1
VLDB-KDD	8	5
VLDB-ICDE	11	1
ICDE-EDBT	5	2
OSDI-SOSP		
VLDB-EDBT	5	2
SIGMOD-KDD	7	2
SIGMOD-ICDE	7	3
SIGMOD-EDBT	4	2
KDD-SIGGRAPH	4	1
SIGMOD-VLDB	9	1
SODA-FOCS-STOC	3	3
OSDI-SOSP-USENIX		
SIGMOD-SIGCOMM	4	1
ICDE-EDBT-SIGMOD	3	3
VLDB-EDBT-SIGMOD	3	6
FOCS-STOC-SODA-ICALP		
SIGMOD-ICDE-VLDB-EDBT	2	224
SIGCOMM-SIGMETRICS-SIGOPS		

Example authors’ “cliques”

	Duration	Matches	Authors
VLDB-ICDE	11	1	Jeffrey Xu Yu, Xuemin Lin
VLDB-SIGMOD	9	1	Beng Chin Ooi, Kian-Lee Tan
VLDB-KDD	8	5	Jiawei Han, Xifeng Yan | Charu C. Aggarwal, Philip S. Yu | Charu C. Aggarwal, Philip S. Yu | Jiawei Han, Philip S. Yu | Jian Pei, Philip S. Yu
SIGMOD-KDD	7	2	Jiawei Han, Xifeng Yan | Jiawei Han, Philip S. Yu
SIGMOD-ICDE	7	3	Divesh Srivastava, Nick Koudas | Beng Chin Ooi, Kian-Lee Tan | Nicolas Bruno, Surajit Chaudhuri
CIKM-SIGIR	6	1	Craig Macdonald, Iadh Ounis
WWW-CIKM	5	2	Yiqun Liu, Min Zhang
ICDE-EDBT	5	2	Haixun Wang, Xuemin Lin | Xuemin Lin, Jeffrey Xu Yu
SIGMOD-SIGCOMM	4	1	Joseph M. Hellerstein, Scott Shenker
SODA-FOCS-STOC	3	3	Ilias Diakonikolas, Constantinos Daskalakis, Anindya De | Ilias Diakonikolas, Rocco A. Servedio, Anindya De | Constantinos Daskalakis, Rocco A. Servedio, Anindya De
WWW-STOC	3	3	Ravi Kumar, T. S. Jayram | S. Muthukrishnan, Vahab S. Mirrokni | Arpita Ghosh, Aaron Roth

Pattern Queries

- First approach on durable patterns
  - Many interesting problems, e.g.,
    - using structural/snapshot partitions
- Other interesting variations of patterns (approximate)
- Beyond durability, e.g., efficient indexing/caching for historical queries

Outline

Introduction, problem definition

Taxonomy of historical queries

Part 1 (general techniques)

- Representation, Storage, Processing

Part 2

- Specific Types of Queries

➡ Conclusions and Future Work

Conclusions

Storage is cheap, store everything is possible (black mirror, novels by Ken Liu, and more)

How to find information in past history and explore it is key

This applies to graphs, generic model of relationships

Current research: first steps

Future Work

Consider historical versions of other types of graph queries

- Keywords
- Skylines
- Etc

Future Work

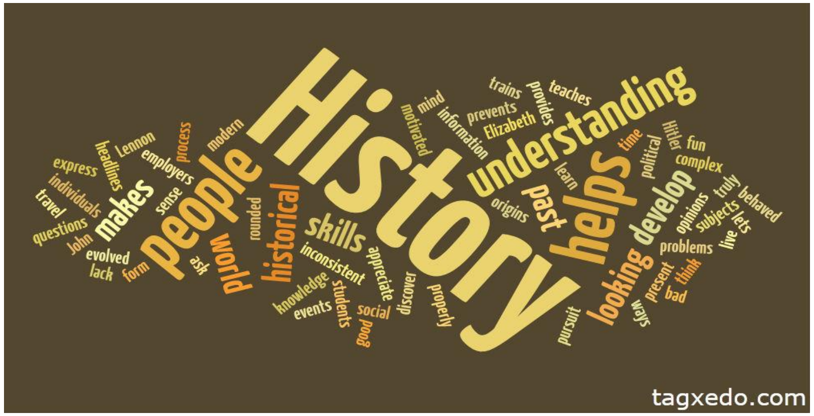
Extend existing systems with history such as: given a query execute it

- as historical query at specific time interval(s) in the past
  - we need also a specification of the semantics
- a most durable query

Future Work

Think of new ways of exploring history

Many more interesting problems in the intersection of query management and knowledge discovery



Thank you! Questions?

References I

[Eurosys14] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, E. Chen, *Chronos: A Graph Engine for Temporal Graph Analysis*, EuroSys 2014

[ACM TOS 2-15] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, W. Chen*: ImmortalGraph: A System for Storage and Analysis of Temporal Graphs*. TOS 11(3): 14:1-14:34 (2015)

[ICDE13] U. Khurana, A. Deshpande, *Efficient snapshot retrieval over historical graph data*, ICDE 2013

[EDBT16] U. Khurana, A. Deshpande: *Storing and Analyzing Historical Graph Data at Scale*. EDBT 2016

[WOS12] G. Koloniari, D. Souravlias, E. Pitoura, *On Graph Deltas for Historical Queries*, WOSS 2012, VLDB workshop

[GRADES13] G. Koloniari, E. Pitoura, *Partial view selection for Evolving Social Graphs*, GRADES 2013

[VLDB11] C. Ren, E. Lo, B. Kao, X. Zhu, R. Cheng: *On Querying Historical Evolving Graph Sequences*. VLDB 2011

[IS17] C Ren, E Lo, B Kao, X Zhu, R Cheng, DW Cheung. *Efficient Processing of Shortest Path Queries in Evolving Graph Sequences,* Information Systems, Available online 7 June 2017

T. Akiba, Y. Iwata, Y. Yoshida, *Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling*, WWW 2014

T. Hayashi, T. Akiba, K. Kawarabayashi: *Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks.* CIKM 2016: 1533-1542

References II

W. Huo, V. Tsotras, *Efficient temporal shortest path queries on evolving social graphs*, SSDBM 2014

[EDBT15] K. Semertzidis, K. Lillis, E. Pitoura: *TimeReach: Indexing for Historical Reachability Queries*, EDBT 2015

[ICDE16] K. Semertzidis, and E. Pitoura, *Durable Graph Pattern Queries on Historical Graphs,* ICDE2016

[ADBIS17] K. Semertzidis, and E. Pitoura, *Historical Traversals in Native Graph Databases*, ADBIS 2017

[PVLDB17] M. Then, T. Kersten, S. Guennemann, A. Kemper and T. Neumann *Automatic Algorithm Transformation for Efficient Multi Snapshot Analytics on Temporal Graphs*, PVLDB 2017

[ICDE15] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and Peter J. Haas: *Dynamic interaction graphs with probabilistic edge decay.* ICDE 2015

[SIMACSE17] Anand Iyer, and I. Stoica, *Time-Evolving Graph Processing on Commodity Clusters*, SIAM Conference on Computational Science and Engineering, 2017

[PVLDB14] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, *Path Problems in Temporal Graphs*. PVLDB 2014

[SODA2002] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick: *Reachability and distance queries via 2-hop labels.* SODA 2002

Additional Citations I

A. G. Labouseur, J. Birnbaum, P. Olsen Jr., Sean R. Spillane, J. Vijayan, W. Han, J. Hwang, *The G* Graph Database: Efficiently Managing Large Distributed Dynamic Graphs*, DAPD, (2014).

D. Caro, M. A. Rodríguez, N. R. Brisaboa *Data structures for temporal graphs based on compact sequence representations* Information Systems 51 (2015) 1–26

Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, Ion Stoica: *Time-evolving graph processing at scale*. GRADES 2016

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, Enhong Chen: *Kineograph: taking the pulse of a fast-changing and connected world*. EuroSys 2012: 85-98

Vera Zaychik Moffitt, Julia Stoyanovich: *Towards sequenced semantics for evolving graphs*. EDBT 2017: 446-449

Vera Zaychik Moffitt, Julia Stoyanovich: *Towards a Distributed Infrastructure for Evolving Graph Analytics.* WWW (Companion Volume) 2016: 843-848

Vera Zaychik Moffitt, Julia Stoyanovich: *Portal: A Query Language for Evolving Graphs.* CoRR abs/1602.00773 (2016)

Xiaoen Ju, Dan Williams, Hani Jamjoom, Kang G. Shin: *Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems*. USENIX Annual Technical Conference 2016: 523-536

Additional Citations II

Peter Macko, Virendra J. Marathe, Daniel W. Margo, Margo I. Seltzer: *LLAMA: Efficient graph analytics using Large Multiversioned Arrays.* ICDE 2015: 363-374

Konstantinos Semertzidis, Evaggelia Pitoura: Time Traveling in Graphs using a Graph Database. EDBT/ICDT Workshops 2016

Ciro Cattuto, Marco Quaggiotto, André Panisson, Alex Averbuch: Time-varying social networks in a graph database: a Neo4j use case. GRADES 2013