Historical Traversals in Native Graph Databases [1]

Authors: Konstantinos Semertzidis and Evaggelia Pitoura
Dept. of Computer Science and Engineering, University of Ioannina, Greece
fksemer, pitourag@cs.uoi.gr

Introduction

1. Focus is on - Traversals of volatile graphs (sequence of graph snapshots at different time instances) using a native graph database.
2. Two main aspects of paper:

   a) Introduction of models for storing such snapshots in graph database.

   b) Some Algorithms for shortest path queries and historical reachability.

3. At last, evaluation and comparison of various models and models using both synthetic and real datasets.

Objective

1. Efficient storing and querying big graph datasets like social, citation, hyperlink, biological networks, etc. using a native graph database.
2. Approaches: Single Edge (Only 1 edge with a value as list of timestamps) and Multi-Edge (Multiple edges showing edges at different timestamps)
3. Database Used: Neo4j [2]
4. Interval Based Approach (Single Edge) proves more efficient. Better preprocessing time and storage.

Related Work/Literature Survey

1. Very few papers use storage as native graph database rather they store database in RAM or disk.
2. [4] uses hierarchical time index to support snapshots with different granularities. (months and days) on same dataset. Focus on retrieving specific snapshots.
3. [5] introduced time logs to capture any event (like add/remove of edge/node).
4. [6] works on graphs with static structures but the changes in edge and node properties is frequent.
5. [1] This paper on **structural updates** and **reachability and path queries**.

Native vs Non-Native Graph Database [3]

1. In a native graph database, a node record's main purpose is to simply point to lists of relationships, labels and properties, making it lightweight.
2. Native graph databases are optimized for storing graphs
3. Eg. Neo4j

1. Non-native graph storage uses a relational database, a columnar database or some other general-purpose data store rather than being specifically engineered for the uniqueness of graph data.
2. Non-native graph databases are not optimized for storing graphs

Required Basics - 1. Historical Graphs

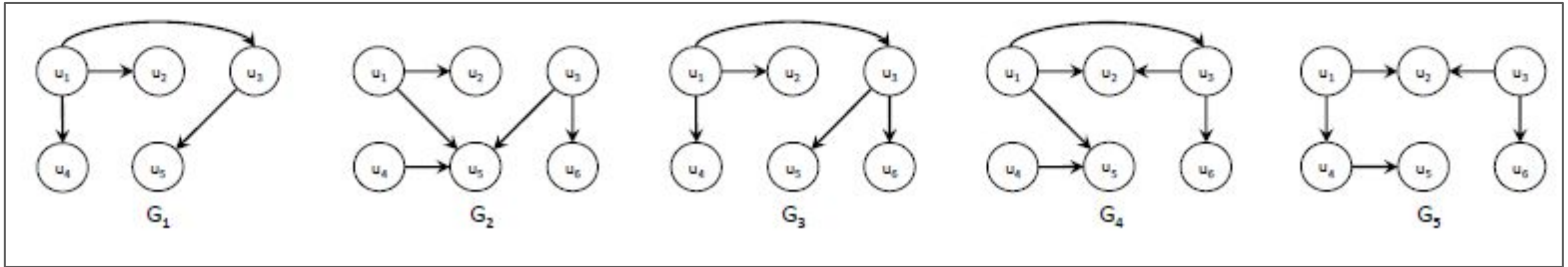


Figure 1: Example of a historical graph [1]. Nodes and edge labels are not shown for simplicity.

A historical graph $\mathcal{G}_{[t_i,t_j]}$ *in time interval* $[t_i, t_j]$ *is a sequence* $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ *of graph snapshots.*

Example of lifespan : These are sets of time intervals for which a particular edge exists.
ls((u1,u3)) = {[1,1], [3,4]}
Example of time join : Given 2 sets of time intervals, join includes only those intervals which are present in both. {[1,3], [5,10], [12,13]} JOIN {[2,7], [11,15]} = {[2,3], [5,7], [12,13] }

Required Basics - 2. Historical Traversal Query

A traversal query Q_H on a historical graph, $\mathcal{G}_{[t_i,t_j]}$, called a historical traversal query, is a tuple $(Q, \mathcal{I}, L)$ where Q is a traversal query $Q = u \xrightarrow{\alpha} v$, $\mathcal{I}$ is a set of time intervals and L is a positive integer. For a path p, let $D(p) = ls(p) \otimes \mathcal{I} \otimes [t_i, t_j]$. Q_H retains the paths p from u to v in $\mathcal{G}_{[t_i,t_j]}$ that satisfy α and for which in addition $D(p)$ contains at least L time instances.

Figure 2: Definition [1]. u and v are nodes. Alpha = constraint on the query.

Example Queries

1. Whether two nodes are reachable in at-least k time instances?
2. What is Earlier shortest path between two nodes? (ESP)
3. What is global shortest path among all time instances? (Stable SP)
4. Shortest among the path that exists in at least k snapshots (KSP).

Required Basics - 3. Storage of Historical Graphs

Fig 3.
ME = Multi Edge Representation

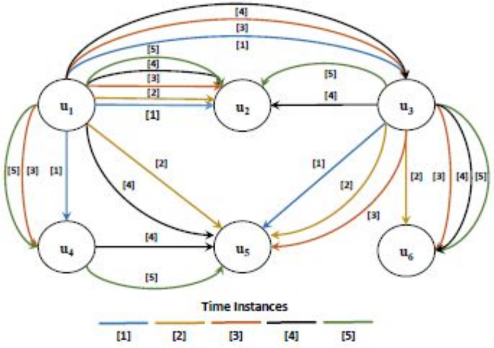
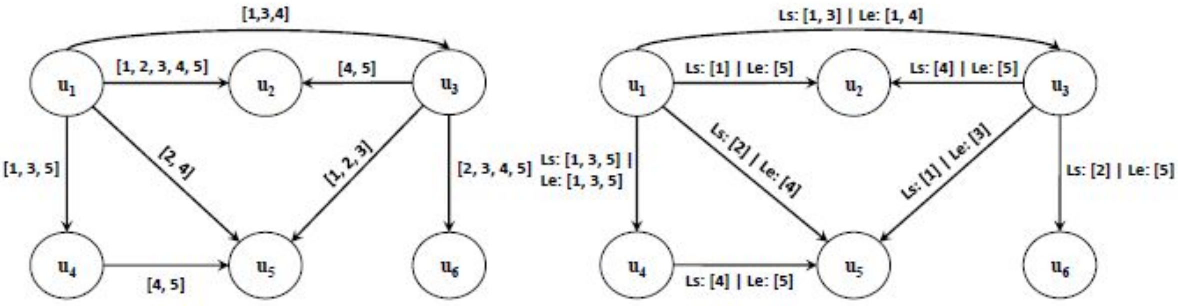


Fig 4.
SE = Single Edge Representations has 2 types

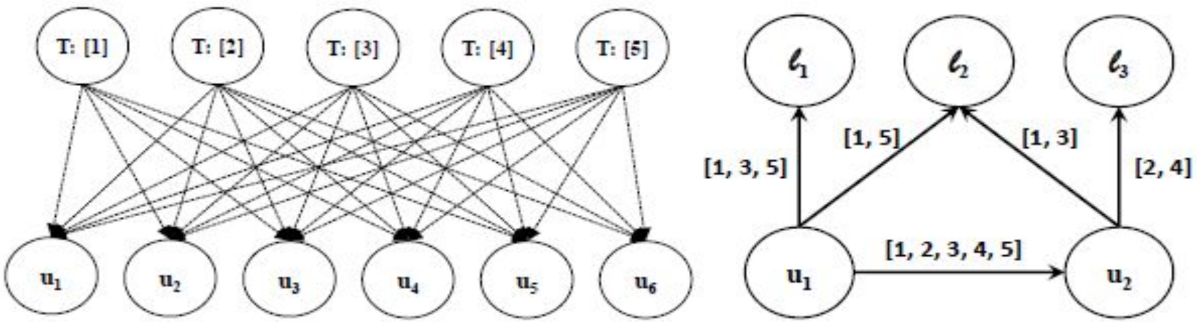
1. Single Edge with Time Points
2. Single Edge with Time Intervals



(a) SETP representation

(b) SETI representation

Required Basics - 4. Indexing and Time varying labels



(a) Time index (b) Time-varying labels

1. As shown above,fig 5 (a) describes indexing where T is special node which corresponds to a specific time instance. Each connection of T to a node shows this node is present at this time instance. Like u6 is absent at T1.
2. As shown above,fig 5 (b) shows way to store labels where l1, l2, l3 are varying labels being represented in SETP fashion. Example of storing time varying labels of 2 nodes u1 and u2

Types of Queries

Reachability Queries:

1. **Disjunctive -** Two nodes are reachable in at least 1 time instance. (OR)
2. **Conjunctive -** Two nodes are reachable in all time instance. (AND)
3. **At-Least K Time Instance -** Two nodes are reachable in at least K time instance.

Algorithm Used for processing Historical Traversal Queries

1. Multi-edge Representations

TRAVERSALBFS (Built-in): **Query** : Get path between two nodes **u** and **v** during time interval **I** :

1. **Approach 1:** Call TRAVERSALBFS starting from **u** once for each timestamp **t** in **I** and then **combine** these results. Ex ESP only (Computationally expensive)
2. **Approach 2: (edge-at-a-time approach)** Call TRAVERSALBFS starting from **u** once for each timestamp **t** in **I,** traverse only the edges of type **t** until we reach **v.** Ex. SSP and KSP

Algorithm Used for processing Historical Traversal Queries

2. Single-edge Representations

Note:
For Pruning (Not traversing same edge again)

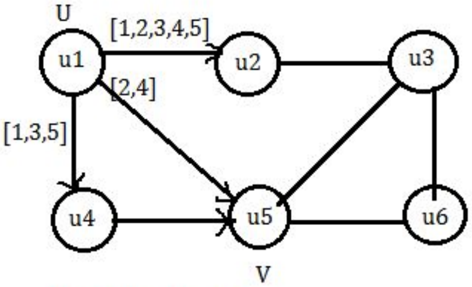
Lines 19-23 work

Algorithm 1 (SETP-SETI) Conjunctive-BFS(u, v, I)

Require: nodes u, v, interval I
Ensure: True if v is reachable from u in all time instances in I and false otherwise

```
1:  create a queue N, create a queue INT
2:  enqueue u onto N, enqueue I onto INT
3:  while N ≠ ∅ do
4:      n ← N.dequeue()
5:      i ← INT.dequeue()
6:      for each e ∈ n.getEdges() do
7:          I_e ← TIME_JOIN(e, i)
8:          if I_e = ∅ then
9:              continue
10:         end if
11:         w ← r.getOtherNode(n)
12:         if w = v then
13:             R ← R ∪ I_e
14:             if R ⊒ I then
15:                 return true
16:             end if
17:             continue
18:         end if
19:         if IN(w) ⋣ I_e then
20:             IN(w) ← IN(w) ∪ I_e
21:             enqueue w onto N
22:             enqueue I_e onto INT
23:         end if
24:     end for
25: end while
26: return false
```

Example Run



1. Start from U = u1
2. Let I = [3,4] (Given)
3. n = u1
4. e = (u1->u5)
5. Ie = ls(u1->u5) INTERSECTION I = [3,4]
6. w = u5 = V
7. R = ls(u1->u5) UNION Ie = [2,4]
8. So I = [3,4] is a subset of [2,4]
9 Returns True

Experimental Setup

1. **Database Used:** Neo4j graph database.
2. Algorithms implemented using Neo4j Java API.
3. They used Quad-core Intel Core i7-3820 3.6 GHz processor, with 64GB memory. Only 1 core is used in all experiments.
4. 2 real and 1 synthetic dataset. Dataset and Graph database characteristics are shown below. Time is load time of dataset in database. Each dataset is stored in 3 different graph database (GDBs) instances

Dataset	# Nodes	# Edges	# Snapshots
DBLP	1,167,854	5,364,298	58
FB	61,967	905,565	871
Synthetic	1,000,000	1,999,325	100

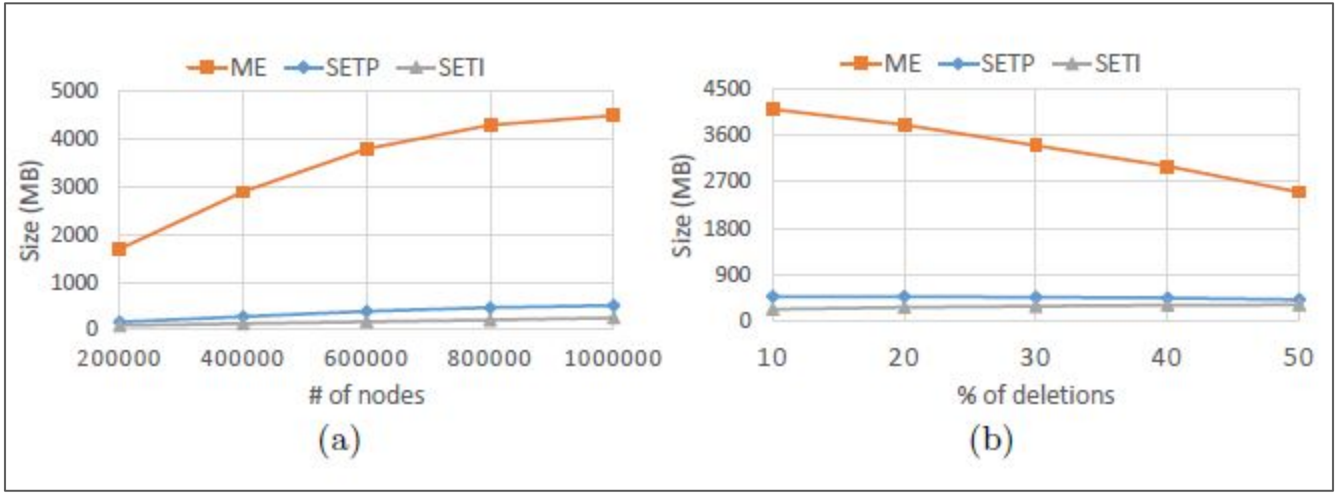
(a) Dataset characteristics

Dataset	GDB	Size (MB)	Index Size (MB)	Time (sec)
DBLP	ME	353	131.37	39
	SETP	528.84		22
	SETI	546.55		23
FB	ME	6,000	830	631
	SETP	400		65
	SETI	31.98		33
Synthetic	ME	4,500	1,700	1,620
	SETP	513		145
	SETI	253		86

(b) Graph database size and creation time

Results for Size

Fig 5



- **Size** (a) for varying number of nodes and (b) percentage of deletions
- Synthetic Dataset is used
- SETI is more space-efficient

Results - Reachability Queries Time

-200 Historical Traversal Queries.
-Source and Target nodes are chosen randomly (uniformly) with restriction that both source and target nodes are present in the query interval.
- Average Query Time is shown for DBLP and FB.

INFERENCES -

1. Disjunctive Queries are faster than conjunctive.
2. And Conjunctive are faster than at-least k.
3. ME for DBLP is a success since fewer edges in this data.
4. ME remains competitive.
5. SETP performs better than SETIi only when the lifespan includes very few time instances (as in DBLP)

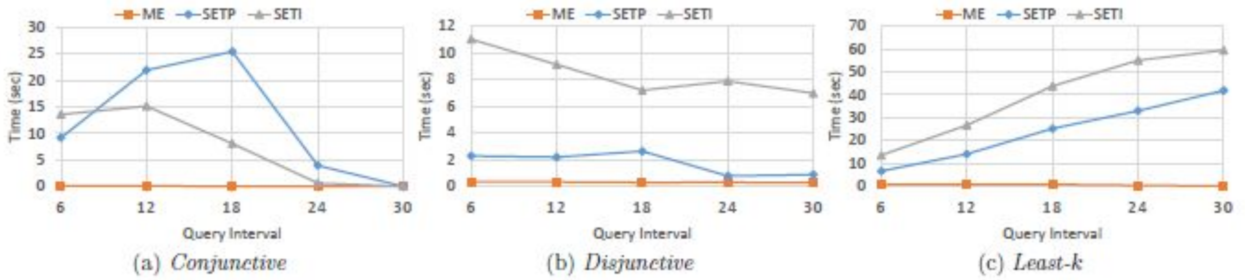


Fig. 6: Query time for historical reachability queries in DBLP

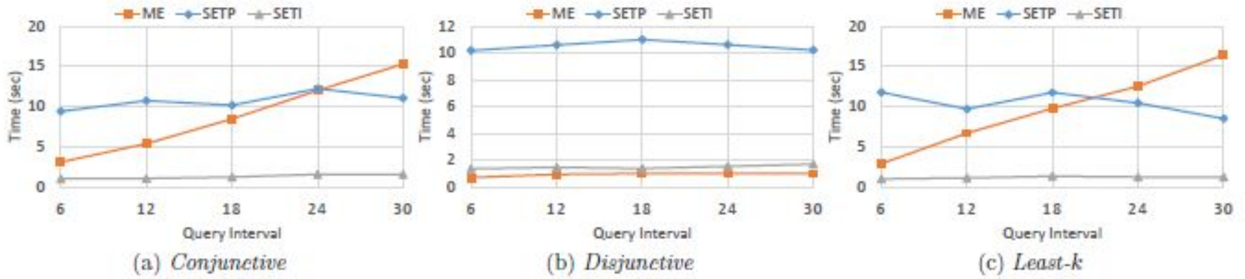


Fig. 7: Query time for historical reachability queries in FB

Results

INFERENCES -

1. Effect of lifespans on query performance are studied using synthetic dataset.
2. ME and SETI are better in conjunctive and disjunctive.
3. For atleast K, SETI is the best.
4. For other figure: SETI is fastest
5. SETP comes second in SSP and KSP.

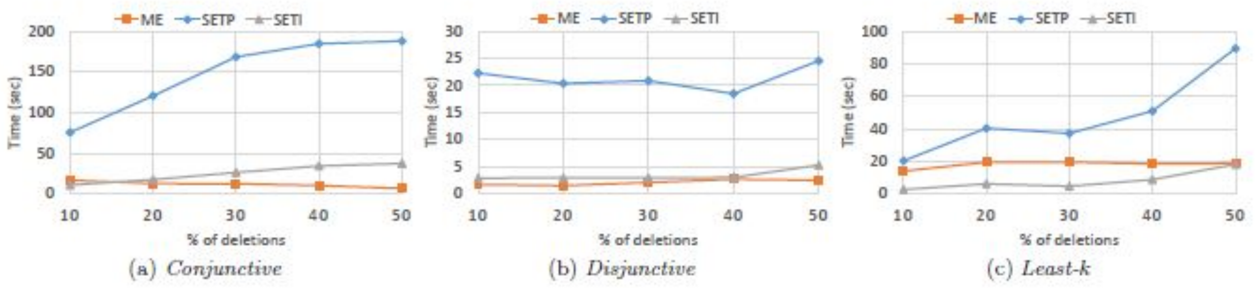


Fig. 8: Query time for historical reachability queries in the synthetic dataset

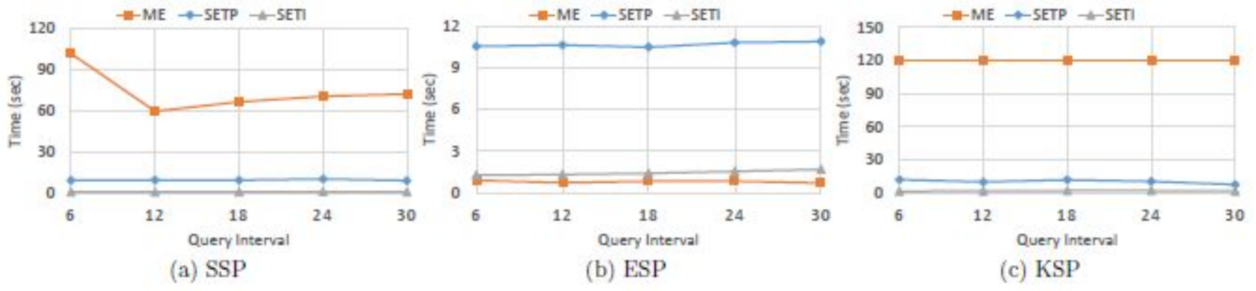
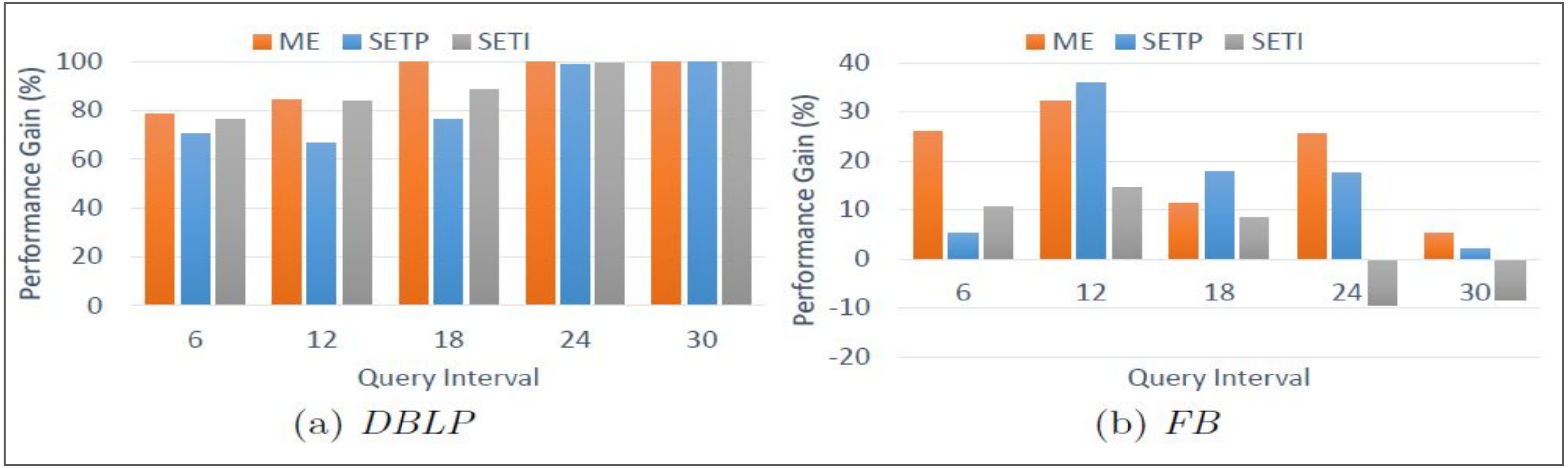


Fig. 9: Query time for historical shortest path queries in FB

Fig 10

Results- Performance Gain



INFERENCES -

1. Shown only for conjunctive queries and time indexing is omitted.
2. Performance increases in DBLP with increase in query interval since there are not many edges and thus indexing returns the negative answers very fast as asked in conjunctive queries.
3. For FB, indexing is helpful in ME and SETP only.

Conclusion and Future Work

1. By taking advantage of built-in traversal methods, ME works well for very short lived edges like in DBLP. However, for other cases, SETP and SETI proves more space and time efficient.
2. Future work proposed :
    - Extending historical queries to variable time and variable labels.
    - Support can be provided for historical graph queries inside native graph database.

What was learnt?

1. Difference between native and non native graph database.
2. Representations for historical graphs like Multiple Edge and Single Edge (SETP and SETI) for efficient storage in database.
3. Algorithm for faster query processing using single edge representations.
4. Exhaustive comparisons for ME, SETP and SETI representations which provide intuition behind algorithm development.
5. Better understanding of Graph databases. (GDBs)

References

1. Semertzidis, Konstantinos & Pitoura, Evaggelia. (2017). Historical Traversals in Native Graph Databases. 167-181. 10.1007/978-3-319-66917-5_12.
2. https://neo4j.com/
3. https://neo4j.com/blog/native-vs-non-native-graph-technology/
4. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: Time-varying social networks in a graph database: a neo4j use case. In: GRADES. p. 11 (2013)
5. Durand, G.C., Pinnecke, M., Broneske, D., Saake, G.: Backlogs and interval timestamps: Building blocks for supporting temporal queries in graph databases. In:EDBT/ICDT Workshops (2017)
6. Huang, H., Song, J., Lin, X., Ma, S., Huai, J.: Tgraph: A temporal graph data management system. In: CIKM. pp. 2469{2472 (2016)