Indexing Continuous Paths in Temporal Graphs

Bart Kuijpers[1], Ignacio Ribas[2], Valeria Soliani[1,2], and Alejandro Vaisman[2](✉)

[1] Hasselt University, Hasselt, Belgium
bart.kuijpers@uhasselt.be
[2] Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina
{iribas,vsoliani,avaisman}@itba.edu.ar

Abstract. Temporal property graph databases track the evolution over time of nodes, properties, and edges in graphs. Computing temporal paths in these graphs is hard. In this paper we focus on indexing Continuous Paths, defined as paths that exist continuously during a certain time interval. We propose an index structure called TGIndex where index nodes are defined as nodes in the graph database. Two different indexing strategies are studied. We show how the index is used for querying and also present different search strategies, that are compared and analyzed using a large synthetic graph.

Keywords: Temporal graphs · Path indexing · Temporal graph index

1 Introduction

Property graphs [1], whose nodes and edges are annotated with properties, are used in most graph databases in the marketplace. In practice, these graphs are typically *static*, i.e., they do not change over time. However, in most real-world applications, edges, nodes, and properties can be added, deleted, and updated as needed. This is addressed in [2], where a model (TGraph) for temporal graph databases is proposed. First-class citizens in this model are temporal paths of three types: Continuous, Pairwise Continuous and Consecutive paths. Continuous Paths (CPs) are paths valid during a certain time interval. In Pairwise Continuous Paths, every pair of adjacent edges has an overlapping time interval. In Consecutive Paths, for every pair of adjacent edges, the time validity of one edge ends before the validity time interval of its consecutive one starts. The model comes equipped with a high-level SQL-like query language called T-GQL, which includes functions to compute the three kinds of paths above. In this work we propose to index CPs to improve their computation.

Indexing paths in temporal and non-temporal graphs has been studied to a limited extend. Pokorny et al. [7] index graph patterns in Neo4j, using a structure stored in the same database as the graph (an approach we follow in this work). Huo and Tsotras [5] study the problem of efficiently computing shortest-paths on evolving social graphs. The authors use an extension of Dijkstra's algorithm

to achieve this for a time-point or a time-interval. To improve performance of queries in temporal databases, Elmasri et al. [4] proposed a basic indexing technique for temporal data that can be combined with conventional attribute indexing schemes to process temporal selections and temporal join operations. In [6] an index structure for temporal attributes is proposed.

In this work we present two index structures for Continuous Paths, one that indexes all the paths and another one that indexes all paths of length two. In the latter case, computing the paths of length higher than two requires additional processing. We show how queries are evaluated using these indices. We also consider reducing the search space by limiting the time window to consider the one in which queries will most likely fit. As far as we are aware of, this is the first proposal for indexing temporal paths on graphs.

Section 2 briefly presents the concepts and definitions in which the present paper is based. Section 3 presents two index structures and in Sect. 4, we show how they are used to process T-GQL queries and updates over the temporal graph. In Sect. 5, we report experimental results. We conclude in Sect. 6.

2 Background

To make the paper self contained, we briefly present the TGraph model.

Definition 1 (Temporal property graph (TGraph) (cf. [2])). *A temporal property graph is a structure $G(N_o, N_a, N_v, E)$ where G is the name of the graph, E is a set of edges, and N_o, N_a, and N_v are disjoint sets of nodes, denoted object nodes, attribute nodes, and value nodes, respectively. Every object and attribute node and every edge are associated with a tuple (name, interval). The name represents the content of the node or the type of the edge, and the interval represents the period(s) when a node or edge is (was) valid. Analogously, value nodes are associated with a (name, interval) pair. For any node n, the elements in its associated pair are referred to as n.name, n.interval, and n.value. As usual in temporal databases, a special value Now tells that the node is valid at the current time. All nodes also have an identifier denoted id.* □

Nodes and edges in G satisfy a collection of temporal constraints. For the sake of space, we refer the reader to [2]. Figure 1 shows a social network represented using the model in Definition 1. There are three kinds of object nodes: Person, City, and Brand. There are also three types of edges: LivedIn, Friend, and Fan. The first one is labeled with the periods when someone lived somewhere, the second one with the periods when two people were friends. An edge of type Fan tells that Peggy Sue-Jones is a Samsung fan since 2005. The temporal attribute node Name represents the name associated with a Person node. Finally, for clarity, if a node is valid throughout the complete history, the temporal labels are omitted.

Definition 2 (Continuous Path). *Given a temporal property graph G, a* continuous path (CP) *with interval T from node n_1 to node n_k, traversing a relationship r, is a sequence $(n_1, \ldots, n_k, r, T)$ of k nodes and an interval T such that there is a sequence of consecutive edges of the form $e_1(n_1, n_2, r, T_1)$, $e_2(n_2, n_3, r, T_2)$, $\ldots$, $e_k(n_{k-1}, n_k, r, T_k)$, $T = \bigcap_{i=1,k} T_i$.* □

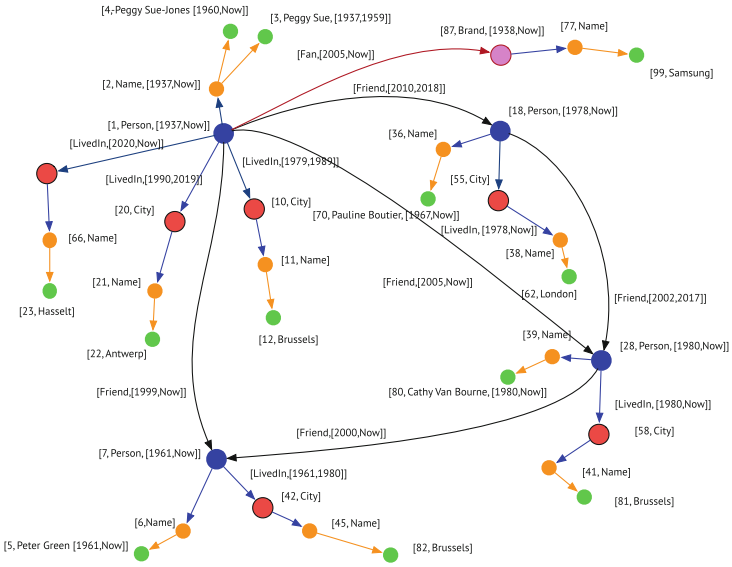


Fig. 1. A temporal property graph.

Continuous paths capture queries like: "Compute the friends of the friends of each person, and the period such that the relationship occurred through all the path." In Fig. 1, for example, Pauline (person node 18) was a friend of Cathy (person node 28) between 2002 and 2017. Also, Peggy Sue (person node 1) was a friend of Pauline between 2010 and 2018. Thus, the path ($PeggySue \xrightarrow{\text{Friend}} Pauline \xrightarrow{\text{Friend}} Cathy, [2010, 2017]$) will be in the answer.

3 An Index for Continuous Paths: TGIndex

Given a TGraph, the *Temporal Out Degree (tod)* of an Object Node $n \in N_o$, with relationship r is the number of intervals associated with r coming out from n. The *tod* of a node is always equal or greater than its out degree. Given an Object node n and a length L, the upper bound on the number of CPs is $O({tod_{max}}^L)$. If N is the number of Object nodes then the TGraph may have a maximum number of CP bounded by $O(N * {tod_{max}}^L)$. Thus, computing the CPs in a temporal graph is an expensive operation. To improve performance when computing CPs, we consider indexing CPs. The idea proposed in this paper builds on [7] together with typical methods for time indexing [3]. We may reduce the search space if we knew in advance that queries ask for paths within a time window $[t_1, t_2]$, with $t_1 \geq t_0$ (the minimum timestamp in the graph) and $t_2 \leq Now$. This way we could index just those CPs. The CPs that exist outside the indexed time interval could be retrieved combining the Index and the TGraph. We describe next two indexing approaches denoted **TGIndexL** and **TGIndex2**.

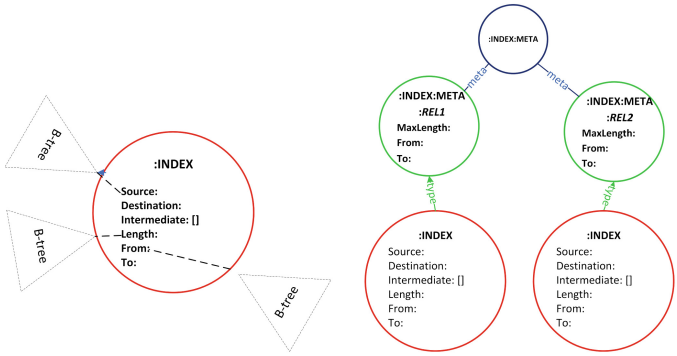


Fig. 2. Left: Ix node; Right: TGIndex nodes Meta and Ix.

The TGIndexL Approach. We denote $cPath_r^*$ the set of all the CPs in G with relationship r. Let $cPath_r^*(L, t_1, t_2)$ be a subset of $cPath_r^*$ such that the CPs have a maximum length L and are valid between t_1 and t_2. We call **TGIndexL** the index $I_{rL}[t_1, t_2]$ containing those cPaths. Each $I_{rL}[t_1, t_2]$ has N index nodes, each one corresponding to a $cPath_{rL}^i$, denoted $Ix(source, destination, length, from, to, intermediate)$ where: (a) *source* contains the *id* of the starting Object node of $cPath_{rL}^i$; (b) *destination* contains the *id* of the ending Object node of $cPath_{rL}^i$; (c) *length* is the number of hops involved in $cPath_{rL}^i$, where $length \leq L$; (d) *from* is the starting time of $cPath_{rL}^i$; (e) *to* is the ending time of $cPath_r^i$; (f) *intermediate* is an ordered list of the of the id's of the intermediate Object nodes $cPath_{rL}^i$. Each Ix node also has two outgoing edges labeled *start* and *end* that connect it with the starting and ending Object nodes of its corresponding $cPath_{rL}^i$, respectively. The Ix nodes are ordered by their starting time *from*, their *length* and by the *source* attribute. Following [4], Neo4j B-Tree indices on those properties are created for this. Figure 2 shows a sketch of the proposed index node. We can see that the TGIndex has two types of nodes: the Ix nodes (leaves), and the nodes containing metadata (indicated by the META label). When creating a $I_{rL}[t_1, t_2]$ we must indicate the time window $[t_1, t_2]$ to calculate only $cPath_r$ existing between t_1 and t_2. Figure 3 shows a TGIndex stored in the same TGraph database (GDB). For clarity, the TGraph shown is an abstraction of the actual graph, since each node of the graph is actually composed by the Object, Attribute and Value nodes.

The TGIndex2 Approach. For very large graphs, our experiments showed that it may take a long time to build TGIndexL, and the required storage space may be large. Therefore, we propose a second approach, denoted **TGIndex2**, that indexes only the CPs of length 2 and then answers the queries by rebuilding the paths from there. This way, the storage space is reduced and the query time, although higher than when indexing every possible CP, is lower than when we compute the paths over the original TGraph. The correctness of the procedure is based on two CPs properties: (a) Every CP of even length k can be decomposed

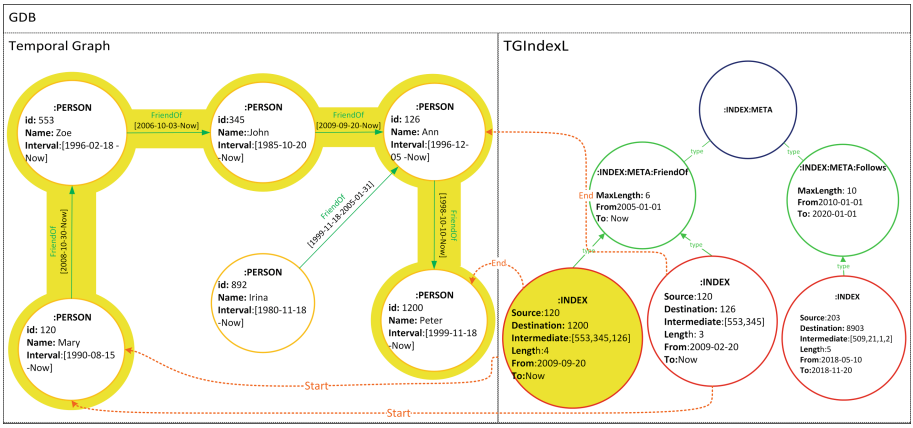


Fig. 3. Indexing paths in a temporal graph: GDB and TGIndexL.

in $k/2$ CPs of length $l = 2$; (b) Every CP of odd length k $cPath_r(n_1, \ldots, n_{k+1}, I)$ can be decomposed in a CP of (even) length $k-1$, $cPath'_r(n_1, \ldots, n_k, I')$ and the last edge e_k between n_k and n_{k+l}, $e_k\{n_k, n_{k+1}\}$. Here, the intersection between the intervals of the $cPath'_r$ and the edge interval equals the interval of the original $cPath$, $e_k.I \cap I' = I$. Every pair of consecutive edges can be seen as $k-1$ CPs of length 2 of the form $cPath^1_r(n_1, n_2, n_3, I_1)$, $cPath^2_r(n_2, n_3, n_4, I_2)$, $cPath^3_r(n_3, n_4, n_5, I_2)$, $\ldots$, $cPath^{k-1}_r(n_{k-1}, n_k, n_{k+1}, I_{k-1})$. Moreover, when k is even, to rebuild the original path we only need $cPath^1_r$, $cPath^3_r$, ..., which are the CPs whose destination matches the source of another one. If k were odd, we can split the CP into two: the one from n_1 to n_k (of length $k-1$), $cPath'_r(n_1, \ldots, n_k, I)$ and the last edge of the CP, which goes from n_k to n_{k+1}. Since $cPath'_r$ has even length, it can be decomposed into several CPs of length 2. The edge to complete the original path is obtained from the last CP of the decomposition $cPath^{k-1}_r$.

Figure 4 illustrates the above. There is an index node for every CP of length 2 valid in a certain time window. Retrieving a CP of length 2 from the index is straightforward: the index nodes themselves are the answer. To retrieve a path of even length k, $k \geq 4$ we obtain the $k/2$ index nodes Ix where $Ix_i.destination = Ix_{i+1}.source$. To improve the performance of queries asking for paths of length greater than 2, the indexed paths of length 2 are connected to each other by edges of type :concat, as Fig. 4 shows. An edge between two index nodes Ix_1 and Ix_2 exists if there is a CP of length 4 in G starting at $Ix_1.source$, ending at $Ix_2.destination$, with $\{Ix_1.destination, Ix_2.source\}$ being the intermediate nodes of the path, and a validity interval I. This interval, which can be obtained by the intersection of $[Ix_1.from, Ix_1.to]$ and $[Ix_2.from, Ix_2.to]$, is a property of the :concat edge. The structure of the index node Ix is the same as in TGIndexL except for the *length* property, which is no longer required.

To estimate the number of index nodes we must calculate the number of CPs of length 2. Let $M = tod_{max}$ and $N = |N_o|$ be the number of Objects nodes in

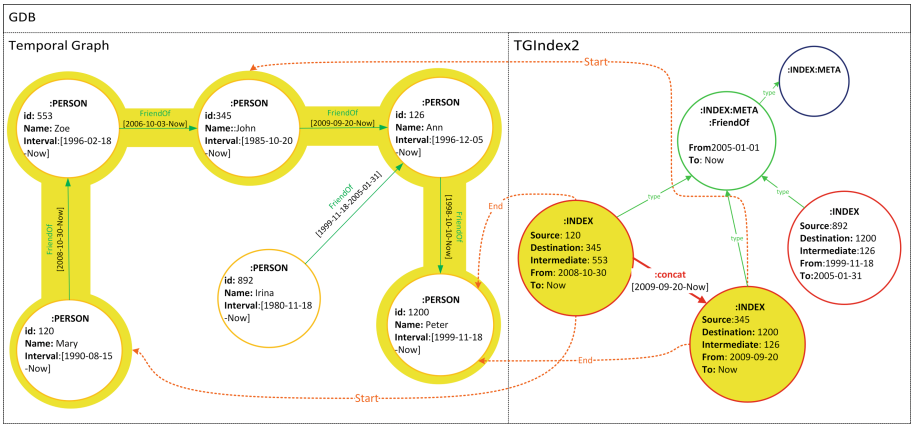


Fig. 4. Indexing paths in a temporal graph: GDB and TGIndex2

G, then the maximum number of paths of length 2 from a node x is $O(M^2)$. Then, there could be a maximum number of $O(N * M^2)$ of index nodes.

4 Using the Index

We explain now how a T-GQL query is executed using the two indexing approaches in the previous section. We start with TGIndexL and the T-GQL query:

```
SELECT paths
MATCH (p1:Person),(p2:Person),
paths = cPath((p1)-[:Friend*4]-> (p2),'2010','2020')
WHERE p1.Name = "Mary"
```

The query processor first checks if there is an index for the relationship Friend whose indexing interval includes the query interval. In the example, the query window time is completely included in the indexed window time. The query is translated into Cypher as:

```
MATCH (v1:Value)<--()<--(o1:Object)
WHERE v1.value = "Mary"
CALL graphindex.retrievePaths(o1,null,4,4,'2010','2020',
{EdgeLabel:'Friend', direction:'outgoing'}) YIELD path,interval
RETURN path, interval;
```

The function `graphindex.retrievePaths` is added to the Neo4j library, so it can be included in a Cypher expression.

Using the TGIndex2 approach for the same query, yields the translation:

```
MATCH (v1:Value)<--()<--(o1:Object)
WHERE v1.value = "Mary"
CALL graphindex.retrievePathsConcat( o1,null,4,4,'2010','2020',
{EdgeLabel:'Friend', direction:'outgoing'} ) YIELD path,interval
RETURN path, interval;
```

Two index nodes are needed to build a CP of length 4. The validity time of such path can be obtained directly from the interval in the relationship :concat (Fig. 4). For queries asking for CPs of length greater than 5, the validity time of the path can be obtained as the intersection of the intervals of the edges labelled :concat between the index nodes. If the query asks for a path of odd length, the procedure is similar but the last edge is obtained from the graph itself. The function `retrievePathsConcat` concatenates the connected index nodes and keeps only those paths whose intersection exist all the way. The time for retrieving CPs using this index depends not only on the number of indexed nodes but also on the execution times of the Neo4j pattern matching strategies. However, navigation through the :concat edges of the index can also be done in Breadth First Search (BFS) fashion. The BFS strategy starts from all the index nodes whose source matches the required id. We developed an algorithm for this second strategy to retrieve paths, denoted `retrievePathsBFSConcat`, not shown here for the sake of space.

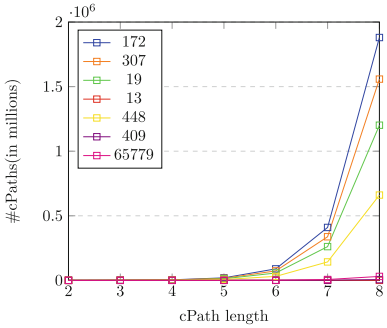
Strategies for Querying a TGraph using TGIndices. Consider a TGraph G and a TGIndex over a relationship r with interval $I_x = [t_1, t_2]$, in any of its implementations. Given the query $QcPath_r(source = x,\ \ dest = y,\ L_{min} = l_1,\ L_{max} = l_2, I_q)$, there are three possible scenarios: (a) $I_q \subseteq I_x$: I_q **during** I_x; (b) $I_q \cap I_x = \emptyset$: I_q **before** I_x or I_x **before** I_q; (c) $I_q \cap I_x \neq \emptyset$: I_q **overlaps** I_x or I_x **during** I_q. In the first case, as explained before, the result set R will be fully provided by the index. In the second, the index could not answer the query at all and the standard procedure `consecutive.continuous` described in [2] must be used. In the last case, the result can be obtained combining the use of the index and G itself. Depending on whether or not I_x is fully included in the query interval, it will be necessary to divide I_q into two or, at most, three separate time intervals. If $I_x \subset I_q$ there will be three sub-intervals: the interval before I_x, I_x itself, and the interval after I_x. Otherwise there will be two sub-intervals.

5 Experimental Evaluation

We compare the performance of the *cPath* function built-in T-GQL against both TGindex2 strategies which we called IndexConcat (the one that uses Neo4j's pattern matching) and IndexBFSConcat (our BFS algorithm), using two kind of queries: (a) SO: Source-only CP queries, which obtain all the CPs of a certain length starting from a given node; (b) P2P: Point-to-Point CP queries, which, given a source and a destination node, obtain all the CPs between them. We discarded the TGIndexL strategy since the index sizes and building times were very high for our graph. We created a synthetic TGraph representing a social

Table 1. TGraph: *tod* & #CPs starting from selected Object Nodes.

	id	172	307	19	13	448	409	65779
	tod	1396	55	57	50	37	29	28
#cPaths by length	2	151	136	108	64	61	6	1
	3	1747	701	1209	703	722	10	23
	4	4212	3375	2683	1509	1436	19	59
	5	19464	15835	12419	0	6590	71	317
	6	89037	73353	57067	0	30775	291	1499
	7	409741	337693	261294	0	142791	1294	6560
	8	1880151	1558243	1201237	0	660440	5916	30502



Length	Class 1	Class 2
2	1	301
3	2	2263
4	6	3375
5	28	11376
6	156	54238
7	379	260262
8	1800	111111

Fig. 5. Left: CPs growth vs. length in SN TGraph; Right: Social network TGraph: Number of cPaths between two nodes classified as Class 1 and Class 2

network (SN) with 38,500 Object nodes of type Person and the same number of Attribute and Value nodes, resulting in 115,500 nodes in total. The Object nodes are connected through a relationship Friend. The graph has two connected components C1 and C2. In C1, for the relationship Friend, the minimum out degree of the Object nodes is 25; there is a Follower Object node whose out-degree is 1,394 and its *tod* is 1396. There is also an Influencer Object node whose in-degree is 15,000. In C2 nodes are less connected than in C1 but there is also a large number of CPs. For this graph, a TGIndex2 is created with a time window of 3 months obtaining 178,436 Index Nodes representing CPs of length 2. We selected representative Object nodes from C_1 to run the SO tests based on their *tod*, the degree and the number of indexed CPs of length 2 that start from them during the indexed period. We denote the nodes by their Object id. Table 1 shows those nodes, their *tod*, and the number of CPs from length 2 to length 8.

The selected time window impacts on the number and length of the CPs. Based on this, we classify the nodes as *Big* (172, 307 and 19), *Medium* (13 and 448) and *Low* (409 and 65779). We run the tests five times for each node and then average the resulting time. To run the P2P queries we chose eight pair of nodes from C_1 and C_2 and observed the number of CPs from length 2 to length

6. On the right-hand side of Fig. 5 we classify the paths as "Class 1" when the resulting number of CPs is low and "Class 2" when that number is high.

Figure 6 shows *the execution times of SO queries* (results are expressed in milliseconds and the scale is logarithmic). We can see that the BFS concat strategy is always better than the other two ones. When the number of CPs is high, both BFS strategies outperform IndexConcat. Also, the difference is higher for paths of even length than for the ones of odd length. On the right-down portion of the figure we show the averaged results of the three kinds of nodes for CPs of even length. We can see that both index strategies are better than the cPath strategy. Classifying the nodes according to the number of CPs retrieved suggests that the BFS strategy should be considered when the source nodes have a high *tod* and produce a high number of CPs of length 2, which in turn will produce a high growth in the number of CPs as the length increases.

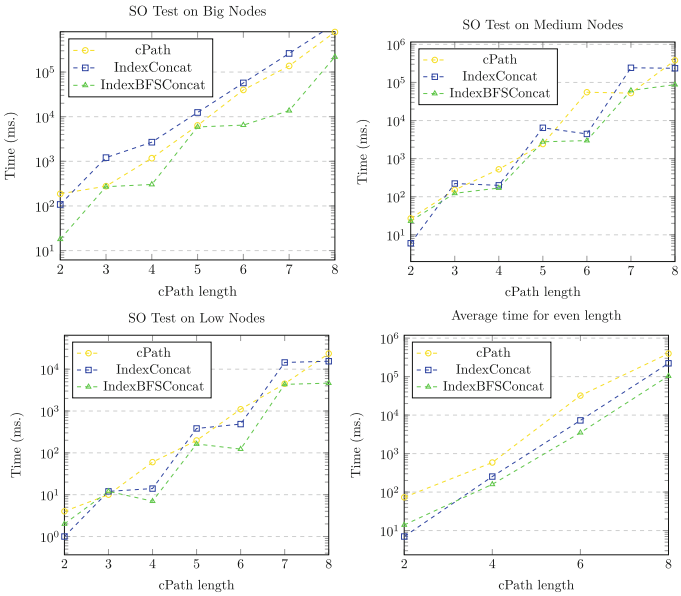


Fig. 6. Execution times vs. path length for SO tests.

Figure 7 compares the *execution times of P2P queries*. In this case, the *index-Concat* strategy clearly outperforms the others when the number of CPs is low (below 1000 approximately). This is because the Neo4j strategy over the index nodes is fast when the search space is small. The number of CPs retrieved when the query mentions a source and a destination node are less than when the source is not mentioned. The *BFSconcat* strategy on the other hand, presents a better performance as long as the number of CPs increases to a very high number of paths, since it prunes the results and avoids evaluating non-useful paths. For higher numbers, the three strategies perform similarly.

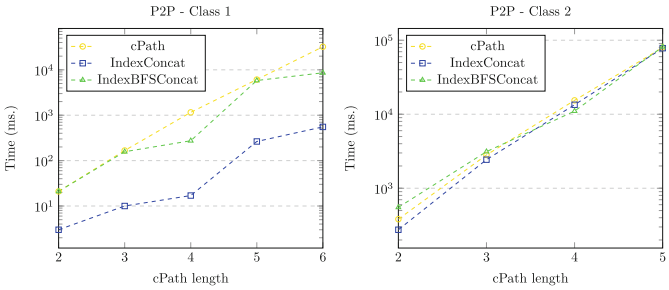


Fig. 7. P2P Test execution time vs. length. Left: Class 1; Right: Class 2.

6 Conclusion

We studied strategies for indexing CPs in temporal graph databases. These indices are defined within the database and point to the start of the CPs being indexed. Two kinds of indices were defined: one that indexes paths of any length (called TGIndexL) and another one that indexes paths of length two (called TGIndex2) and builds paths of any length from there. Since the latter proved to be more effective, two query evaluation strategies are defined for it. One of them takes advantage of the underlying graph database search strategies, and the other one using a BFS strategy. We carried out tests over synthetic temporal graphs with good results since both strategies outperformed the non-indexed evaluation algorithm. We believe that the ideas developed in this paper set the basis for future work that includes developing new evaluation and implementation strategies and performing further experiments.

Acknowledgements. Valeria Soliani and Alejandro Vaisman were partially supported by Project PICT 2017-1054, from the Argentinian Scientific Agency.

References

1. Angles, R.: The property graph database model. In: Proceedings of AMW 2018, CEUR Workshop Proceedings, Cali, Colombia, 21–25 May 2018, vol. 2100. CEUR-WS.org (2018)
2. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. VLDB J. **30**(5), 825–858 (2021). https://doi.org/10.1007/s00778-021-00675-4
3. Elmasri, R., Kim, Y., Wuu, G.T.J.: Efficient implementation techniques for the time index. In: Proceedings of ICDE 1991, 8–12 April 1991, Kobe, Japan, pp. 102–111. IEEE Computer Society (1991)
4. Elmasri, R., Wuu, G.T.J., Kim, Y.: The time index: an access structure for temporal data. In: Proceedings of VLDB 1990, Brisbane, Queensland, Australia, 13–16 August 1990, pp. 1–12. Morgan Kaufmann (1990)
5. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: SSDBM, Aalborg, Denmark, 30 June–2 July 2014, pp. 38:1–38:4 (2014)

6. Kvet, M., Matiasko, K.: Impact of index structures on temporal database performance. In: EMS 2016, Pisa, Italy, 28–30 November 2016, pp. 3–9. IEEE (2016)
7. Pokorný, J., Valenta, M., Troup, M.: Indexing patterns in graph databases. In: DATA 2018, Porto, Portugal, 26–28 July 2018, pp. 313–321. SciTePress (2018)