Backlogs and Interval Timestamps: Building Blocks for Supporting Temporal Queries in Graph Databases.

Gabriel Campero Durand
Otto-von-Guericke-Universität Magdeburg
23 PUBLICATIONS 33 CITATIONS
SEE PROFILE

Marcus Pinnecke
Otto-von-Guericke-Universität Magdeburg
22 PUBLICATIONS 58 CITATIONS
SEE PROFILE

David Broneske
Otto-von-Guericke-Universität Magdeburg
39 PUBLICATIONS 79 CITATIONS
SEE PROFILE

Gunter Saake
Otto-von-Guericke-Universität Magdeburg
637 PUBLICATIONS 6,779 CITATIONS
SEE PROFILE

Backlogs and Interval Timestamps: Building Blocks for Supporting Temporal Queries in Graph Databases

Work in progress paper

Gabriel Campero Durand
University of Magdeburg
Germany
gabrielcampero@acm.org

Marcus Pinnecke, David Broneske,
Gunter Saake
University of Magdeburg
Germany
firstname.lastname@ovgu.de

ABSTRACT

The analysis of networks, either at a single point in time or through their evolution, is an increasingly important task in modern data management. Graph databases are uniquely suited to improve static network analysis. However, there's still no consensus on how to best model data evolution with these databases. In our work we propose an elementary concept to support temporal analysis with property graph databases, using a single-graph model limited to structural changes. We manage the temporal aspects of items with interval timestamps and backlogs. To include backlogs in the model we examine two alternatives: (1) global indexes, and (2) using the graph as an index by resorting to timestamp denormalization. We evaluate density calculation and time slice retrieval over successive days from a SNAP dataset, on an Apache Titan prototype of our model, observing from 2x to 100x response time gains by comparing differential vs. snapshot methods; and no conclusive difference between the backlog alternatives.

CCS Concepts

•Information systems → Graph-based database models; Temporal data;

Keywords

Graph-based database models, Temporal analysis of information networks, Differential graph algorithms

1. INTRODUCTION

There is a common saying that time changes everything. While it might be a trivial observation, it seems to apply particularly well to information systems: the semantics and value of any data item might not be the same if it was recorded today rather than several years ago.

The timestamp of when a data item is created, updated or deleted, adds meaningful context to this item; a context that in turn enables better auditing, security and historical understanding. Clearly, storing the time dimension of data can make a difference.

Capitalizing on the declining cost of storage, present-day information systems can go beyond the simple timestamping of the latest item transactions, by employing databases to additionally track all historic versions of business data items. This amounts to dismissing in-place updates and adopting an accumulate-only approach, where any change in an item is appended as a new item. As a result, the whole history of data evolution can be mined for valuable business insights.

The database community has produced a plethora of concepts and tools for supporting the temporal dimension in information systems under the relational model [8, 21]. However, pure relational models make navigation of real world networks a cumbersome endeavor, thus they do not support network analysis as well as graph databases. For this task relational systems might gain from specialized indexes [18].

Although *temporal analysis* has proven to be important for understanding networks [1], spawning taxonomies for its fine-grained sub-tasks [2, 11, 23], the support for temporal features in graph databases is in a patently less developed state as for their relational counterparts.

The common workflow for ad hoc temporal network analysis revolves around large sequences of separate snapshots. Processing is performed in an embarrassingly parallel way, usually without combining the snapshots. A few specialized systems have been proposed for such workflow [10, 16].

With the increasing adoption of graph databases another workflow can be considered, based on the continuous collection of snapshots in an accumulate-only graph. Scarce studies exist tackling this workflow or, more broadly, how general graph databases could aid temporal analysis.

In this short paper we present our ongoing studies on the support of general graph databases for temporal network analysis. We begin by briefly presenting the long-established concepts of interval timestamps and backlogs (Section 2). Afterwards we continue with our contributions as follows:

1. We outline a concept for including interval timestamps and backlogs into a property graph model (Section 3).
2. We give empirical evidence illustrating the benefits of our approach with a prototype based on Apache Titan, Cassandra, and Gremlin [20] (Section 4).

Subsequently we discuss our findings (Section 4.2). We conclude this paper by presenting related work (Section 5) and proposing future directions for our research (Section 6).

2. FUNDAMENTALS

Historical data has been managed for decades in relational database systems. For a broader background, we refer readers to reference work entries on *Temporal Data Models*[15] and other work [8, 21, 7]. In this section, we describe two major temporal data representation models that have been used for this purpose, *interval timestamps* and *backlogs.*

2.1 Running example

As a running example we consider a group of devices and management services which monitor these devices and track their interconnections, similar to the as-733 SNAP dataset. Devices and the connections between them, can be added or removed (which corresponds to events). The connections between devices can be represented by a `connects-to` relationship. If a device is removed, its connections to other devices are removed as well.

2.2 Interval timestamps

The time-dependent existence of entities can be expressed using *interval timestamps.*

An interval timestamp contains two bits of information: the *start* date (including) and the *end* date (excluding). Using an interval timestamp as metadata, temporal objects can be annotated to express their validity and existence inside a certain range of time between the start date and the end date. For instance, consider two temporal objects A and B with timestamps $t(A) = [d_{\text{start}}, d_{\text{end}})$ and $t(B) = [d'_{\text{start}}, d'_{\text{end}})$. The object A exists before the object B if $(d_{\text{start}} < d'_{\text{start}})$ holds. Both objects A and B exist together in a certain time span if also $(d_{\text{end}} > d'_{\text{start}})$ holds.

By adding interval timestamps, temporal queries can be presented in a straightforward and efficient way, such as *asking for all items that exist at a certain point in time*, an operation known as *time slice retrieval.*

Physically, interval timestamps add two columns to historical data, which correspond to the start and end date. For a better understanding, we give an example of interval timestamping in Figure 1 by using traditional tabular representation. The entries record a sequence of events, happening on 3 successive days (d_1, d_2, d_3): On the first day, devices 1 and 2 are added to the database, in addition to a *connects-to* relation from device 1 to 2, with a as the id of this relation. On the second day, device 3 is added, next to a *connects-to* relation from device 2 to 3, with b being the id of such relation. At the third day, both the device 1 and its *connects-to* relation (tuple a) are deleted/archived.

Note that a distinction can be made between tuple interval timestamps (affecting the whole tuple) and attribute interval timestamps (affecting only a certain tuple attribute). For the purpose of our discussion we will limit ourselves to tuple interval timestamps. Furthermore, we consider a model where items only exist within a given interval, if an item is deleted and re-inserted, it would be considered a different item.

2.3 Backlogs

An alternative to timestamped intervals is a *backlog.* A backlog is basically a list of events ordered by time.

Figure 1: Example for tuple interval timestamps: device relation (left) and connects-to-relation (right)

Start	End	id	Extras
d1	d3	1	
d1		2	
d2		3	

Start	End	id	From id	To id	Extras
d1	d3	a	1	2	
d2		b	2	3	

Figure 2: Backlog example, based on [7].

Timepoint	Event	Device Id	Connects-to Id	From Id	To Id	Extras
d1	creation	1				
d1	creation	2				
d1	creation		a	1	2	
d2	creation	3				
d2	creation		b	2	3	
d3	deletion	1				
d3	deletion		a	1	2	

Backlogs are a useful concept to recap the evolution of a system under observation. Each entry (d_i, E_i) in a backlog is a record containing the actual timestamp d_i of the recorded event E_i.

For instance, we can monitor a system and catch its stream of events $(d_1, E_1), (d_2, E_2), ..., (d_n, E_n)$ in a protocol, namely the event backlog.

For a better understanding of the differences between interval timestamps and backlogs, we illustrate in Figure 2 the same succession of events presented previously with interval timestamps (see Section 2.2), but in a traditional backlog representation, with $E = \{creation, deletion\}$.

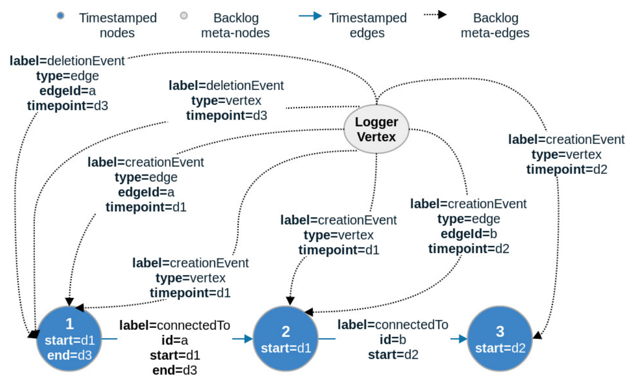
In direct comparison to interval timestamps, the entries in a backlog representation are immutable and instead of intervals each entry is only valid at one specific timepoint rather than during a time span. To reconstruct the state of an entity at a given moment, this representation requires to replay the sequence of events affecting the entity (either from scratch or over a previously validated state of this entity).

3. TIME IN PROPERTY GRAPHS

After presenting the necessary background about backlogs and interval timestamps in the previous section, we now provide our proposal to incorporate both concepts into a property graph database system(Section 3.1). Following this, we introduce an example of how both time representations could be used for the commonplace task of density calculation. To illustrate how to exploit the effective availability of backlogs, we include an incremental version of density calculation. Our examples are given using a state-of-the-art graph query language, GREMLIN (Section 3.2).

3.1 Including time in a property graph

In Figure 3, we display possible ways to embed temporal information in a property graph model, continuing with our running example from Section 2.1.



Timestamped nodes
Backlog meta-nodes
Timestamped edges
Backlog meta-edges
label=deletionEvent type=edge edgeId=a timepoint=d3
label=deletionEvent type=vertex timepoint=d3
Logger Vertex
label=creationEvent type=vertex timepoint=d2
label=creationEvent type=edge edgeId=a timepoint=d1
label=creationEvent type=vertex timepoint=d1
label=creationEvent type=edge edgeId=b timepoint=d2
label=creationEvent type=vertex timepoint=d1
1 start=d1 end=d3
label=connectedTo id=a start=d1 end=d3
2 start=d1
label=connectedTo id=b start=d2
3 start=d2

Figure 3: Example of using interval timestamps and embedded backlogs, enabling the use of the graph as an index: Resulting graph at day 3 (d3)

While the interval timestamps can be easily recorded as properties of the items themselves, native support for backlogs could be accomplished in 2 different ways:

1. **Global Indexing**. In this alternative, global indexes covering creation and deletion times of items can be used at run-time to efficiently reconstruct ordered sets of backlogs for given periods (Figure 3, with only the timestamped nodes and edges).

2. **Graph as an index**. Another alternative is to use the graph as an index in itself (GRAIN), by using a special class of edges –backlog edges, connected to logger meta-vertexes (Figure 3, considering all entities). The collection of backlog edges (going out from the logger vertexes) serves as a backlog table (Figure 2).

Assuming that indexes can exist solely over a given entity type (vertices or edges), the alternative that capitalizes on global indexes might imply the need for 4 different indexes to capture the different events (creation or deletion). On the other hand, if we adopt GRAIN, then when any item is created or deleted, a new backlog edge must be created and connected from a logger vertex to the given item (or, in the case of edges, to the vertex at the head of the item). These backlog edges record a timepoint (the denormalized timestamps), two possible edge labels to distinguish the events (i.e., either create or archive events) and a type property to distinguish the type of logged element.

Through its outgoing connections, the logger vertexes can act as single entry point to query about structural changes in the graph, accelerating the access to elements associated with specific change events. In this strategy, queries might further benefit from the existence of local indexes in the logger vertexes, covering the relevant properties from the backlog edges and granting efficient access to the backlogs.

Regardless of the approach selected for providing backlogs, a fundamental benefit that can be expected from having both temporal representations in the same graph model, is the possibility of combining *differential* (supported by the backlogs) with *snapshot* (supported by the interval timestamps) *processing* in a single operation, thus providing adjustable means to optimize temporal analysis.

3.2 Density calculation using backlogs and interval timestamps

To illustrate the different processing approach that backlogs could enable, we consider density calculation[1]. This task requires to count the number of vertices and edges valid at a given timepoint. Such counting can be naively implemented with incremental processing by calculating the actual values on the first day and incrementally updating them with backlog records, for the following days.

Next are some examples of different GREMLIN traversals that could be used for this task. We start with the basic case, using simple traversals to count the number of elements active at a given date, based on interval timestamps (with the predicate: $deletedAt > date \wedge createdAt \leq date$). The examples cover the vertices, similar approaches could be followed for the edges:

```
titanGraph.V().
             hasLabel("device").
             has("deletedAt", P.gt(date)).
             has("createdAt", P.lte(date)).
             count().next();
```

Next we include traversals for incremental density calculation, finding out the number of elements changed on a given date using equality predicates.

The following example uses global indexes over *createdAt* and *deletedAt*. Note the need for accessing two indexes for vertices and (not shown) two additional indexes for edges:

```
titanGraph.V().
             hasLabel("device").
             has("createdAt", date).
             count().next();
titanGraph.V().
             hasLabel("device").
             has("deletedAt", date).
             count().next();
```

Finally, we illustrate the GRAIN strategy. In the example we count all backlog edges labeled as *created* on a given date, and group them by the type of entity created.The same approach can be used for deletions.

```
Map<Object,Long> creationCount =
(Map<Object, Long>) titanGraph.V(logId).
               outE("created").
               has("createdAt", date).
               groupCount("created").by("createdType").
               cap("created").next();
numCreatedVertices= creationCount.get(Vertex.class);
numCreatedEdges= creationCount.get(Edge.class);
```

4. EVALUATION

To provide an early assessment of possible improvements from backlog-mediated differential processing, we implemented a prototype using Apache TITAN 1.0[2], a property graph database using the TINKERPOP 3[3] framework.

[1]For the purposes of our study, we define the density of a graph as $|E|/(|V| * |V| - 1))|$, with E being the number of edges and V the number of vertices of the graph.
[2]http://thinkaurelius.github.io/titan/
[3]http://tinkerpop.incubator.apache.org/

We formulated our queries in GREMLIN [20]. As storage backend for TITAN we used CASSANDRA 2.1.11.

In TITAN the properties of a vertex and all the information of its connected edges, are stored within each vertex storage space. Among other options, performance tuning is made possible by defining global indexes (over combinations of vertex/edge properties), or local vertex-level indexes (that map from edge properties and values to matching edges stored in the same vertex row).

So as to adequately represent TITAN's capabilities in our prototype, we defined global indexes covering the exact combination of properties that we queried against. Vertex-level indexes were also created in the logger vertex, to speed up access to its edges along the lines of the properties we queried against.

Given that currently the basic TITAN indexes (known as composite indexes) only address exact matches, interval queries (which tend to appear in temporal analyses) are notably lacking support in our implementation. This amounts to a fundamental limitation in our prototype. In future work we will use TITAN's full-text indexes (referred to as mixed indexes) or exogenous interval-specific indexes, to provide support for this type of queries.

The experiments were conducted on a commodity multicore machine running Ubuntu 14.04 and Java SDK 8u111-linux-x64, with an Intel® Core^{TM} i7-2760QM CPU @ 2.40 GHz processor (8 cores in total) and 7.7 GiB of memory.

4.1 Data and workload characteristics

For the dataset we selected the SNAP as-733 dataset [14], a collection of 733 daily instances from November 8 1997 to January 2 2000, of peer relationship networks (defined by having traffic flows between them) among autonomous systems. Vertices on this dataset only possess one property –their nodeId. Unlike other types of networks, vertices and edges appear and disappear on a daily basis, following a super-linear growth over time in the edge to vertex ratio; with a densification exponent of a=1.18 [13].

We decided on testing two basic functions, stemming from the original research on this dataset [13]: density calculation and a whole-graph time slice retrieval over successive days. For density calculation we used the same strategies described in the GREMLIN examples on Section 3.2. The incremental version of time slice retrieval followed a similar approach, starting by reconstructing the complete graph as of the first day, and for the successive days limiting itself to updating the daily subset of items marked as changed in the backlogs.

4.2 Performance results

In our studies the basic approach proved to be less efficient than its incremental counterparts, a point made markedly clear as the number of successive days increased.Figure 4 and 5 display the observed average response times for density calculation and time slice retrieval requests, based on instrumented application-level timers. In both tasks, the basic approach was observed to scale almost linearly with the progression of days. The scalability profile of the incremental strategies could not be deduced from our observations. To understand this, one should remember that the performance of differential methods does not depend on the number of days but on the amount of elements changing across the days, a factor that averages low in the as-733 dataset[4].

[4]To give some perspective, the number of elements valid on a given day was in the lower thousands (~3-6k vertices, ~10-26k edges), whereas the number of elements changed was generally less than 200.

Figure 4: Avg. response time of density calculation

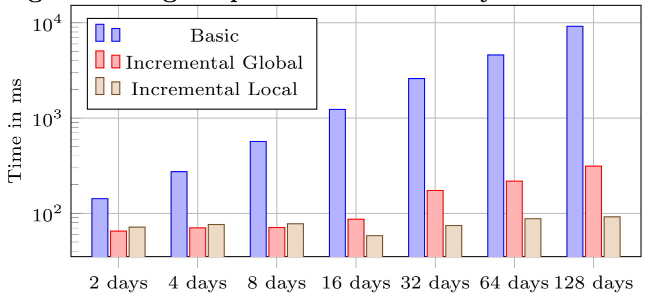
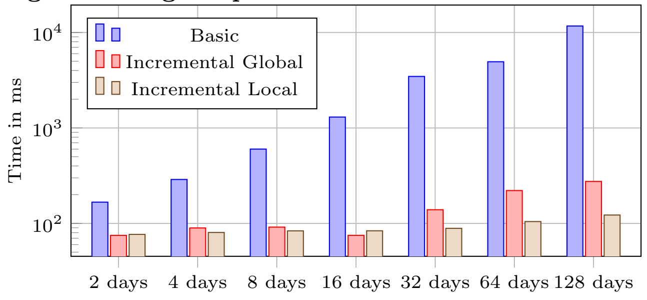


Figure 5: Avg. response time of time slice retrieval



In the case of density calculation, an average speedup of 2.1x was observed at 2 days between the incremental and the basic cases; at 128 days it increased to 95x. For time slice retrieval the average speedup went from 1.9x at 2 days, to 100x at 128 days.

TITAN's lack of index support for the interval queries in the basic method[5], paired with good index usage from the incremental methods (exclusively asking for exact queries, e.g. createdAt==today), contributed to widen the performance gap between the basic and incremental cases.

A closer look at the daily traversals, enabled by GREMLIN's *profile step*, revealed that the basic methods spent more than 95% of their time in a global *TitanGraphStep*, a step tasked with finding vertices or edges valid at a given timepoint. Such a step could be deemed the equivalent of a full-table scan in relational databases. The global index solution required 4 daily traversals, each of them consuming 95% of their time in the *TitanGraphStep*, however the total time for these traversals was considerably less than in the basic case given that the underlying access was mediated by a global index, and the number of elements to retrieve was smaller (thousands vs. tens). The remaining time for the traversals in the basic and global incremental cases corresponded to a *count step* (for densities) and an *id step* (for time slices).

The GRAIN solution required only 2 traversals per day (see code snippets in section 4), each of them taking approximately 2% of their time for accessing the logger vertex, 4.7% on selecting the backlog edges marking the changes for a given timepoint (in a *TitanVertex step*), and the rest of the time in *groupCount* or *branch steps*, required to group the properties of edges and vertices.

[5]i.e. $deletedAt > today \wedge createdAt \leq today$.

4.3 Memory consumption

Consistent with the observed traversal profiles, portraying the GRAIN solution as more compute-bound than others, heap usage was observed (via JConsole) to be the lowest for that approach, peaking at 173 Mb throughout a single run over increasing number of successive days. This can also be explained because of the smaller number of indexes used and cached (only 2 vertex-local indexes) and reduced reloading of physical entities, thanks to the use of denormalized properties (i.e., ids). In contrast, the basic method evidenced the highest memory footprint, peaking at 460 Mb, followed by the global index method, which peaked at 203 Mb. The high footprint of the basic method was expected, as this approach sifts without indexes over the whole data. The difference between global and local functions could be attributed to the memory costs of loading several global indexes, without any gain from denormalization.

Finally, while through all of our studies incremental strategies led conclusively to better performance than the basic implementation (Figure 4 and 5); it was not statistically clear over several runs that one approach to providing backlogs would be better than the other in most scenarios. Within the limited scope of our tests, the GRAIN strategy consistently outperformed the global alternative only on very large number of days (64 and more), possibly because of the former's lower memory footprint and reduced access to physical entities by using denormalized values. Further tests are needed to establish the distinguishing performance characteristics of the alternatives, their tradeoffs and opportunities.

5. RELATED WORK

Industry studies have found that temporal database technologies are able to significantly reduce the cost of developing business products that model time[6]. This is reflected in changes to the SQL standard, increasing support for temporal operations in mainstream database systems. SQL:2011 incorporates temporal features through system versioning and validity intervals associated with tuples [12]. Relational database system vendors are nowadays shipping temporal support to end users (e.g., IBM DB2, Microsoft ImmortalDB,and SAP HANA [9]).

To our knowledge, only a handful of proposals exist for augmenting a general graph database with temporal functionality.

TGraph [5] builds upon Neo4j and targets specific types of graphs for which changes in vertex/edge properties are frequent and structural changes happen seldom. The authors develop a specialized storage system to manage the dynamic properties, but do not address the structural changes that are the focus of our work. Cattuto et al. [3] also base their research on Neo4j, addressing time-varying social networks. They present a data model that -like our study- uses the graph as an index in itself, with the concept of user-level *frame nodes* which are stringed together to form a *timeline*, each *frame* pointing to the elements valid within it, in such a way that any element can be part of several *frames*. Contrasting to our approach, the *frame node* constitutes an index over the validity of items whereas our logger vertex acts as an index over change events on items.

[6]https://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/

Semertzidis and Pitoura [23] describe a model following the *frame node* concept, with Sparksee as a backend. They offer algorithms for historic reachability queries, comparing two graph representations: one where each edge can record all the times in which it was active by having several disjoint intervals of validity (single-edge case), and another where each edge can only have one interval of validity, forcing copies of the edge to be created for each new validity interval (multi-edge case, as our study). The authors find that the second case is better supported by the native graph storage, leading to reduced latencies.

The improvement of specific temporal queries, while using interval timestamped elements, has also been researched. Indexing and algorithmic changes have been shown to provide meaningful contributions [6, 22, 24].

The proposals listed above do not include differential processing, hence our approach could be complementary to them.

Studies that do not use general graph databases often address a workflow defined by the loading of separate snapshots. Moffitt and Stoyanovich [17] introduce a distributed processing framework for time graphs running on GraphX. They examine 3 physical representations for time graphs, and the kind of locality favoured by each: SnapshotGraph, OneGraph and HybridGraph. The first one keeps separate snapshots (favoring structural locality), the second one compacts all changes in a single graph with interval timestamps (favoring temporal and structural locality), the third combines both, trading compactness for better structural locality. While the OneGraph case is close to our solution, the authors use a single-edge strategy for tracking changes, and do not discuss backlogs or differential processing.

Chronos [4] and ImmortalGraph [16] are storage and execution engines designed to compute over a series of snapshots. The authors present strategies to improve locality when laying out multiple snapshots of a graph in memory. The processing is further tuned to the data layout with a locality-aware batch scheduling method, and additional incremental computation powered by pre-computing the intersection or union of snapshots.

Graph deltas, or the set of changes between two snapshots, have been at the core of studies seeking to exploit them to reduce the number of physical snapshots, improving processing and storage. Among them, Khurana and Deshpande [10] document a solution for time slice retrieval based on the concept of DeltaGraph: a hierarchical distributed index of deltas that, once overlaid to a snapshot or an intermediate materialized view, can be used to efficiently reconstruct other snapshots of the graph. The authors also show that different parameters and materialization choices help control the query response times, suggesting possible improvements to their solution. Koloniari et al. [11] present a model that only keeps a physical copy of the current graph and stores the backlogs externally as append-only log files. Ren et al. [19] advocate for a different alternative to the problem of reducing the number of snapshots, by eschewing differential approaches in favour of forwarding queries to pre-computed representative models of snapshot clusters, subsequently verifying and fixing the results.

By aggregating all changes in one graph and natively supporting backlogs, our study amounts to a different perspective for powering differential computing in graph databases, with possible applications to large scale scenarios defined by loading separate snapshots, as the ones described above.

6. CONCLUSION

In this paper we document an elementary inquiry into supporting temporal queries with an existing property graph database. For simplicity, we limited our scope to a single-graph model under an accumulate-only approach, displaying structural changes rather than changes on graph properties. We identified interval timestamps and backlogs as promising building blocks to model graph evolution and aid computations over the graph. For embedding the backlogs we considered two fundamental alternatives: the use of global indexes to reconstruct ordered sets of backlogs on run-time, or the storage of the backlogs as graph elements themselves. The later solution was observed to enable further optimizations with the use of local indexes over the backlog edges.

We performed an empirical study, backed by an existing graph database, to compare the alternatives and assess the impact of differential processing as a query optimization mechanism. Early results show that backlog-mediated incremental processing can contribute significantly to improving temporal queries. These findings make the case that the efficient retrieval of backlogs should be a building block for temporal graph models, considering the ease of adoption and that it opens up opportunities for differential processing.

Further studies are needed to characterize the operations more likely to gain from backlogs. As future directions we aim to validate our observations with other datasets, queries and databases, improving the baseline case with special indexes for range queries. We will specially carry out more tests to characterize the pros and cons of the alternatives for backlog support. We will pacakage the temporal functionality into a thin layer that can be included over an existing graph database. We would also like to address property updates, hierarchies of logger vertices, query language integration, the role of backlogs for efficient snapshot reduction and the adjustable tuning of operations by combining differential with point-in-time processing.

7. ACKNOWLEDGMENTS

The authors would like to thank Thomas Spatzier and the IBM Bluemix Availability Monitoring team, for their generous mentoring in prototyping with graph technologies for real-world use cases. This work was partially funded by the DFG (grant no.: SA 465/50-1).

8. REFERENCES

[1] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *CSUR*, 47(1):10, 2014.

[2] J.-w. Ahn, C. Plaisant, and B. Shneiderman. A task taxonomy for network evolution analysis. *TVCG*, 20(3):365–376, 2014.

[3] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch. Time-varying social networks in a graph database: A neo4j use case. In *GRADES*, page 11. ACM, 2013.

[4] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *EuroSys*, page 1. ACM, 2014.

[5] H. Huang, J. Song, X. Lin, S. Ma, and J. Huai. Tgraph: A temporal graph data management system. In *CIKM*, pages 2469–2472. ACM, 2016.

[6] W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, page 38. ACM, 2014.

[7] C. S. Jensen and L. Mark. Queries on change in an extended relational model. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):192–200, 1992.

[8] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*, pages 1173–1184. ACM, 2013.

[9] M. Kaufmann, P. Vagenas, P. M. Fischer, D. Kossmann, and F. Färber. Comprehensive and interactive temporal query processing with SAP HANA. *VLDB*, 6(12):1210–1213, 2013.

[10] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008. IEEE, 2013.

[11] G. Koloniari, D. Souravlias, and E. Pitoura. On graph deltas for historical queries. In *WOSS*, 2012.

[12] K. Kulkarni and J.-E. Michels. Temporal features in sql: 2011. *SIGMOD Rec.*, 41(3):34–43, 2012.

[13] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*, pages 177–187. ACM, 2005.

[14] J. Leskovec and A. Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[15] L. Liu and M. T. Özsu. *Encyclopedia of database systems*, volume 6. Springer Berlin, Heidelberg, Germany, 2009.

[16] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *TOS*, 11(3):14, 2015.

[17] V. Z. Moffitt and J. Stoyanovich. Towards a distributed infrastructure for evolving graph analytics. In *WWW Companion*, pages 843–848, 2016.

[18] M. Pinnecke. Efficient single step traversals in main-memory graph-shaped data. Master's thesis, School of Computer Science, University of Magdeburg, 6 2016.

[19] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *VLDB*, 4(11):726–737, 2011.

[20] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10. ACM, 2015.

[21] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *CSUR*, 31(2):158–221, 1999.

[22] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *ICDE*, 2016.

[23] K. Semertzidis and E. Pitoura. Time traveling in graphs using a graph database. In *GraphQ Workshop*, 2016.

[24] K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, pages 121–132, 2015.