Investigation of database models for evolving graphs

Alexandros Spitalas
Aristotle University of Thessaloniki, Greece

Anastasios Gounaris
Aristotle University of Thessaloniki, Greece

Kostas Tsichlas
University of Patras, Greece

Andreas Kosmatopoulos
Aristotle University of Thessaloniki, Greece

Abstract

We deal with the efficient implementation of storage models for time-varying graphs. To this end, we present an improved approach for the HiNode vertex-centric model based on MongoDB. This approach, apart from its inherent space optimality, exhibits significant improvements in global query execution times, which is the most challenging query type for entity-centric approaches. Not only significant speedups are achieved but more expensive queries can be executed as well, when compared to an implementation based on Cassandra due to the capability to exploit indices to a larger extent and benefit from in-database query processing.

2012 ACM Subject Classification Information systems → Database design and models

Keywords and phrases Temporal Graphs, Indexing

Digital Object Identifier 10.4230/LIPIcs.TIME.2021.2

Acknowledgements The open access publication of this article was supported by the Alpen-Adria-Universität Klagenfurt, Austria.

1 Introduction

During a conference in 2009 some of the attendees (405 out of 1200 attendees in total) were carrying a RFID tag that could detect close contacts for two days [16]. They aggregated all daily contact information into two networks (snapshots) for the two consecutive days of the conference. They finally generated snapshots of longer timescales by repeating these two networks and adding some stochastic noise. Their goal was to study a SEIR epidemiological model on the contact network. Such an approach, where a series of aggregated snapshots of the same graph is analyzed, has two main advantages. Firstly, ease of modeling: aggregating the dynamic contact information at a coarser time granularity, e.g., per day, renders the modelling simpler albeit at the expense of losing some information, such as in which exact time period within a day two people met. Secondly, ease of management: storing and managing such time-evolving graphs for long periods is not an easy task, thus having fewer snapshots facilitates their processing and analysis. In this paper, we focus on the second aspect, related to the efficient data management of time-varying graphs. Improving the data management efficiency also mitigates the problem of information loss given that the more efficient the management of a series of snapshots, the higher the frequency at which snapshots can be generated.

Introducing the time dimension in the analysis of networks has been of increasing interest the last years in various scientific fields. This is the reason why such networks have been called dynamic networks, adaptive networks, time-varying networks, evolving networks and

28th International Symposium on Temporal Representation and Reasoning (TIME 2021).
Editors: Carlo Combi, Johann Eder, and Mark Reynolds; Article No. 2; pp. 2:1–2:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

temporal networks that essentially refer to the same idea. In [1], a unified framework called TVG (Time-Varying Graph) was proposed and all different formalisms have been shown to be expressed easily in such a framework. Various time-related operations are discussed with respect to their implementation. However, there is no discussion as to how these networks are stored. In general, it seems that there is a lack of a comprehensive framework to actually handle these time evolving graphs, meaning that one should start from the storage model and go all the way to the actual implementation of the algorithm for a specific problem.

TVGs constitute a *graph data structure* with entities corresponding to *vertices* and the relationships between them corresponding to *edges*; both the vertex and edge elements may be annotated by *attributes*, such as name and weight respectively. The distinctive feature of TVGs is their dynamic nature with vertices and edges constantly being inserted, removed or altered as time progresses and entities interact with each other. By studying the evolution of these dynamic graphs we can obtain useful information and metrics regarding the nature of the originating network itself. As a result, one of the greatest challenges that arises in the presence of such *evolving graphs* is maintaining the state of the graph at different time instances (referred to as *snapshots*) in a spatially and temporally efficient way.

1.1 Background and Related Work

There have been two main approaches with regard to a TVG system's design [10, 3, 4], the *time-centric* approach and the *entity-centric* approach (see [2] for a related discussion with a comparison between them). In the former case, the system is indexed according to the time instances (i.e. changes are organized by the time instance they occur in), while in the latter case the system is indexed according to the entities, their relationships and their respective history throughout the snapshots (i.e. changes are organized based on the vertex or edge they refer to). Most of the previous research work conducted so far aims at storing the changes themselves (known as *deltas*) that occur between different snapshots. A system that maintains sets of deltas is thus able to reconstruct any particular snapshot by sequentially applying all the deltas up to the desired time instance. This framework can be used in both approaches but lends itself more naturally to a time-centric approach.

Another viewpoint concerning a system's design is based on the type of queries that the system should be able to evaluate. *Local queries* are based on a particular vertex or a limited selection of adjacent vertices (e.g., the 2-hop neighborhood of a vertex) while *global queries* consider the majority or the entirety of a graph's vertices (e.g., global clustering coefficient). Furthermore, both local and global queries should be able to be executed on either a single snapshot or on a range of snapshots (e.g., average shortest path length between two vertices in the ten first snapshots).

There have been two main research directions over the previous years with regards to evolving graph storage processing. Systems for non-evolving graphs, such as Trinity [14], Pregel [9], and others can be leveraged to support historical queries through explicitly storing each snapshot, but apparently, such solutions are inefficient. A comprehensive survey for evolving graph data management can be found in [5] with the most notable proposals being those in [3, 8, 15, 11]. In general, these techniques rely on storage of snapshots and deltas (logging), which exhibits a trade-off between space and time. Having a large number of snapshots results in deltas of small size but the space cost is substantial since we need to maintain many copies of the graph. On the other hand, having a handful of snapshots means that deltas will be quite large and queries at specific time instances may require a long time to execute. Three of these proposals operate in a parallel or distributed setting, i.e., DeltaGraph [3], TGI [4] and G^* [8]. Notably, the G^* parallel system takes advantage of the commonalities that exist between snapshots by only storing each version of a vertex once and avoids storing redundant information that is not modified between different snapshots. Furthermore, G^* achieves substantial data locality since each G^* server is assigned its own set of vertices and corresponding entities. On the other hand, G^* uses some form of logging to store connection information between different entities.

We take an entity-centric approach, the storage model of which has appeared in [7]. This new storage model is more space efficient and in most of the cases more time efficient. This model departs from the logging framework (snapshots + deltas) by storing the history at the level of the nodes instead of storing snapshots. It has been attached in the G^* prototype [15] for handling TVGs. Unfortunately, this prototype is incomplete and with severe restrictions that render its use rather impractical (see [6] for a related discussion). There also exist solutions for specific problems that cannot be generalized to arbitrary operations, including historical reachability queries [13], mining the most frequently changing component [17], continuous pattern detection [17] or shortest path distance queries [12]. To tackle these limitations, an early attempt to depart from G^* and use NoSQL has appeared in [6]. In this work, we further improve upon [6] by replacing Cassandra with MongoDB with a view to exploiting additional indexing options and techniques to perform query processing.

1.2 Our contribution

Based on the aforementioned discussion, someone can expect that the time-centric approach is more suited towards evaluating global queries over a few snapshots. At the same time, in order to efficiently handle local queries, an entity-centric approach seems to be the natural choice. While there has been plenty of work revolving around the usage of deltas and (variants of) the time-centric approach, entity-centric systems are at their infancy and have not been thoroughly studied. This paper describes our work on devising efficient storage solutions for the entity-centric model; our work capitalizes on our previous work in [7, 6]. In particular, in Section 2 we describe the vertex-centric storage model given by the authors in [7], and we provide details for two completed implementation approaches of the vertex-centric schema in Cassandra as described in [6], which are shown to outperform solutions based on tailored graph management systems, such as Neo4j. We describe our new approach using MongoDB, which better exploits in-database query processing mechanisms in Section 3. To be more precise, our main motivation behind using MongoDB is to exploit the wider range of indexing options and the capabilities provided to reduce the client involvement when processing queries. Our proposal, which is freely available, is thoroughly evaluated in Section 4 and the results show significant improvement especially for global queries, whereas we manage to run more expensive queries on the same infrastructure. Our focus on global queries is justified by the fact that local queries can be easily and efficiently handled by our purely entity-centric approach. Finally, we conclude in Section 5.

2 The HiNode storage model and its initial implementation

Let $G = (V, E)$ be a graph consisting of a set of *vertices* V and a set of *edges* E. The state of the graph G at a particular time instance t, that is, the active vertices and edges of G at a time instance t, is termed as *snapshot* and is denoted by G_t. We regard time as strictly increasing quantities of indivisible time intervals that follow a linear order. Under this notion of time $\mathcal{G} =< G_1, G_2, \ldots >$ corresponds to a constantly *evolving graph sequence* of snapshots that are to be stored and maintained appropriately.

In [7], the first purely entity-centric, and more specifically, vertex-centric model for maintaining graph historical data, termed as *HiNode* is introduced. Its strongest point is that it builds upon a theoretical storage model that is asymptotically space-optimal, time efficient and supports a general notion of time that needs not be contrained to linear as previously described. The core idea behind HiNode's solution is that a vertex history throughout all snapshots is combined into a set of collections called *diachronic node*. HiNode supports adding or removing vertices and attributes as fundamental operations upon which more complex operations and queries (e.g., graph traversal, shortest path evaluation etc.) are constructed. In HiNode, each change is stored $O(1)$ times, resulting in an asymptotically optimal total space cost. Furthermore, due to the local handling of history, HiNode performs well on local queries and the authors further demonstrate that HiNode on top of G^* is competitive regarding global queries as well when compared to G^* [7].

2.1 Data Structure Overview

A vertex $v \in G_i$ is characterized by a set of attributes (e.g., color), a set of incoming edges from the other vertices of G_i and a set of outgoing edges to the other vertices of G_i. We construct an external interval tree $\mathcal{I}$ that maintains a set of intervals $\{\mathcal{T}^v_{t_s,t_e}\}$ where an interval $\mathcal{T}^v_{t_s,t_e}$ signifies the "lifetime" of a vertex v, i.e. from time instance t_s to time instance t_e. We mark a vertex to be "active" (alive) up until the latest time instance, by setting the t_e value to be $+\infty$. Finally, each interval $\mathcal{T}^v_{t_s,t_e}$ is augmented with a pointer (handle) to an additional data structure for each vertex v, called *diachronic node*.

A diachronic node $\mathcal{D}_v$ of a vertex v maintains a collection of data structures corresponding to the full vertex history in the sequence $\mathcal{G}$, i.e. when that vertex was inserted, all corresponding changes to its edges or attributes and finally its deletion time (if applicable). More formally, a diachronic node $\mathcal{D}_v$ maintains an external interval tree $\mathcal{I}_v$ which stores information regarding all of v's characteristics (attributes and edges) throughout the entire $\mathcal{G}$ sequence. An interval in $\mathcal{I}_v$ is stored as a quadruple $(f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)$, where f is the identifier of the attribute that has values ℓ_1, ℓ_2, $\ldots$ during the time interval $[t_s, t_e]$. Note that an edge belonging to v (i.e. one endpoint of the edge is v), can be represented as an attribute of v by using one value ℓ_i to denote the other end of the edge, another value ℓ_j to mark the edge as incoming or outgoing and a last value ℓ_h that is used as a handle to the diachronic node corresponding to the vertex in the other end of the edge. The remaining ℓ values can be used to store the attributes of the edge themselves (e.g., weight). Additionally, the diachronic node maintains a B-Tree for each attribute and for each individual edge of the vertex. Full details are in [7].

2.2 Initial implementation in Cassandra

The first HiNode implementation, hereafter termed as *HiNode-G^**[1], was based on extensions to the G^* [8, 15] parallel graph database. This design choice incurred severe limitations regarding the efficiency and scalability of the *HiNode-G^** prototype (see [6] for a detailed discussion). In a follow-up work [6], we proposed to leverage NoSQL as the underlying database technology providing preliminary results about two different implementation approaches in Cassandra. These approaches consist of the *Single Table (ST)* and *Multiple Table (MT)* implementations. In the former case, the entire history of a vertex is stored in a single table with each vertex

[1] Source code available at `https://github.com/hinodeauthors/hinode`

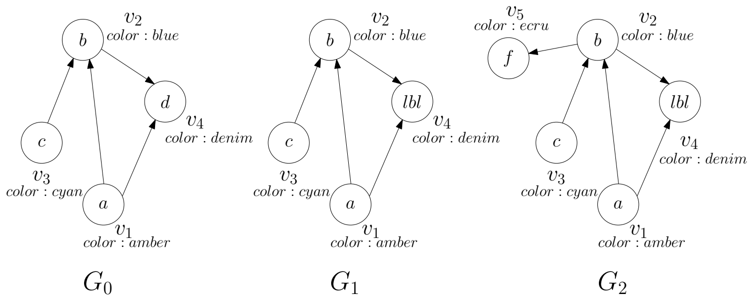

Figure 1 Each vertex possesses two attributes: a name and a color. Additionally, vertices are connected by labelled edges. Three consecutive snapshots are depicted. Snapshot G_1 is obtained by changing the name of v_4 in G_0 from d to lbl. Snapshot G_2 is obtained from G_1 by inserting v_5 and an edge from v_2 to v_5.

id	tart	nd	name	color	incoming_edges	outgoing_edges
1	0	∞	[{value: 'a', start: '0', end: '∞'}]	[{value: 'amber', start: '0', end: '∞'}]	null	'2': [{label: 'elbl1', start: '0', end: '∞'}], '4': [{label: 'elbl2', start: '0', end: '∞'}]
2	0	∞	[{value: 'b', start: '0', end: '∞'}]	[{value: 'blue', start: '0', end: '∞'}]	'1': [{label: 'elbl1', start: '0', end: '∞'}], '3': [{label: 'elbl5', start: '0', end: '∞'}]	'4': [{label: 'elbl3', start: '0', end: '∞'}], '5': [{label: 'elbl4', start: '2', end: '∞'}]
3	0	∞	[{value: 'c', start: '0', end: '∞'}]	[{value: 'cyan', start: '0', end: '∞'}]	null	'2': [{label: 'elbl5', start: '0', end: '∞'}]
4	0	∞	[{value: 'd', start: '0', end: '1'}, {value: 'lbl', start: '1', end: '∞'}]	[{value: 'denim', start: '0', end: '∞'}]	'1': [{label: 'elbl2', start: '0', end: '∞'}], '2': [{label: 'elbl3', start: '0', end: '∞'}]	null
5	2	2	[{value: 'f', start: '2', end: '∞'}]	[{value: 'ecru', start: '2', end: '∞'}]	'2': [{label: 'elbl4', start: '2', end: '∞'}]	null

Table 1 ST (Single Table) representation for the graph sequence shown in Fig. 1. The fields "start" and "end" correspond to the time range in which the corresponding value is valid.

corresponding to a single table row, while in the latter case the data of each vertex is stored in multiple tables with each table corresponding to a single vertex attribute. Tables 1 and 2 show the single table and multi table implementations for the toy example shown in Figure 1.[2]

In order to adequately support global type of queries (i.e. queries that involve a significant part of a snapshot's vertices), the two models offer two querying modes for the retrieval of all vertices relevant to a specified query. Let $[t_s, t_e]$ be a specified time range for which a query is about to be executed. In the first mode (termed *retrieve_all* (*RA*)), and regardless of the given time range, we retrieve all vertices from each model and then perform a client-side filtering operation, where we discard any vertices that do not belong in $[t_s, t_e]$. In the second mode (termed *retrieve_relevant* (*RR*)), in each model, we first determine the vertices that are "alive" at $[t_s, t_e]$ and then, we retrieve them.

While in ST, the implementation of *RR* is straightforward, MT requires additional work since retrieving a particular (set of) attribute(s) during a certain time interval $[t_s, t_e]$ would translate to a range query and the retrieval of all data with a "timestamp" value between t_s

[2] Source code available at `https://github.com/akosmato/HinodeNoSQL`

(a) vertex

vid	start	end
1	0	∞
2	0	∞
3	0	∞
4	0	∞
5	2	∞

(b) vertex_name

vid	start	name
1	0	a
2	0	b
3	0	c
4	0	d
4	1	lbl
5	2	f

(c) vertex_color

vid	start	name
1	0	amber
2	0	blue
3	0	cyan
4	0	denim
5	2	ecru

(d) edge_incoming

targetid	start	end	sourceid
2	0	∞	1
2	0	∞	3
4	0	∞	1
4	0	∞	2
5	2	∞	2

(e) edge_outgoing

sourceid	start	end	targetid
1	0	∞	2
1	0	∞	4
2	0	∞	4
2	2	∞	5
3	0	∞	2

(f) edge_label_incoming

targetid	start	sourceid	label
2	0	1	elbl1
2	0	3	elbl5
4	0	1	elbl2
4	0	2	elbl3
5	2	2	elbl4

(g) edge_label_outgoing

sourceid	start	targetid	label
1	0	2	elbl1
1	0	4	elbl2
2	0	4	elbl3
2	2	5	elbl4
3	0	2	elbl5

Table 2 The MT (Multiple Table) representation of the graph sequence shown in Fig. 1. The fields "start" and "end" correspond to the time range in which the corresponding value is valid. "vid" corresponds to the id of the vertex while "sourceid" and "targetid" correspond to the source and the target of a directed edge respectively.

and t_e (i.e. we are not interested in any updates that occur outside $[t_s, t_e]$). Since Cassandra does not natively permit double-bounded range queries for the sake of efficiency, we fetch the relevant data with a timestamp larger than t_s and then filter all data with a timestamp larger than t_e at the client side. In [6] there is extensive experimental evaluation. The conclusion is that the choice of a particular vertex-centric implementation is not straightforward and exhibits different trade-offs depending on the query at hand.

3 A MongoDB implementation

Our main motivation behind using MongoDB is to exploit the wider range of indexing options and the capabilities provided to reduce the client involvement when processing queries. Additionally, in Cassandra, data are saved as strings and, as such, they are being serialized when returned to the client, while in MongoDB, we have the ability to store the elements of the nodes with a combination of lists and documents. Overall, we are able to perform more complex in-database queries and decrease the client involvement in query processing. Finally, in the new implementation, instead of getting the documents from the database in a single large batch, we have the option to employ a `foreach` approach (when this is expected to be more efficient) and as a result, to mitigate intermediate client-side storage requirements.[3]

3.1 Schema alternatives

Both the ST and MT models shown in Tables 1 and 2 have been transformed to comply with MongoDB's JSON format in a straightforward manner. In addition, we developed an alternative schema for both models, where the elements of the primary key are inserted as characteristics in the document; as primary key, we insert the standard key assigned by MongoDB automatically. The reason for this schema is to further simplify the client-side tasks (i.e., the processing refers to the document content exclusively) with no difference in the capability of answering specific types of queries.

[3] Source code available at `https://github.com/alexspitalas/HiNode-MongoDB/`

In the ST model, a document is a representation of a diachronic node and consists of the primary key as a triple (`vid, start` and `end` of the node), the incoming and outgoing edges and the vertex attributes. The features forming the key are stored as atomic string values, while the vertex attributes are stored as a list of sub-documents, where each document is a triple. The incoming and outgoing edge metadata are stored as a sub-document containing a list of triples (where each triple is a MongoDB sub-document). The former document is essentially a hashmap structure with the key corresponding to the vertex id, while the nested sub-document stores the attributes and the period for each edge. The following 3 indices are built on: (i) `vid`; (ii) `start` and `end`; the complete key. The first index allows quick retrieval of a specific vertex, while the second and third indices facilitate stabbing queries. Finally, as already mentioned, we have experimented with an alternative ST model created (termed as NoKey), where the key is the default `_id` provided by MongoDB, and `vid, start` and `end` are inserted as characteristics of the document, while the indices are the same.

In the MT model, we split the diachronic node into 3 sets of collections, one about the vertex ((a)-(c) in Table 2), one about incoming edges ((d),(f)) and one about outgoing edges ((e)-(g)). Each set consists of one collection about the time period and the existence of the vertex or edge and one collection for every attribute. The standard indices are on `vid` and `vid,start` in the first set of collections ((a)-(c) in Table 2). For the edge collection sets, the multikey indices are on `sourceID` (or `targetID`) and the `start` timestamp. In summary, the main difference with the Cassandra-based implementation in [6] in terms of modelling is the increased flexibility regarding indices and the fact that sub-documents are stored without being serialized as strings.

3.2 Query processing

For local queries, the server (database) side is straightforward, while most of the work is performed on the client side. The local queries we investigate in this paper are retrieving the history of a vertex and one hop queries. In the former case, we retrieve the history of a specified vertex for some time period. In the latter query, all neighboring vertex ids of the query vertex at a specified time period are returned. Both tasks are supported by the two implementation models in a straightforward manner.

Due to the vertex-centric approach, we investigate global queries since local queries can be supported very efficiently. Global query processing comprises two phases. The first is concerned with the retrieval of the data, while in the second one, the processing of the retrieved data takes place. These phases can be intertwined. In our implementation and experiments, the two phases are separated so that the client's side is the same for all ST-based and all MT-based techniques. Regarding the retrieval of the data, three variants have been developed, *retrieve_relevant* (*RR*), *retrieve_all* (*RA*) and *in-database* (*ID*).

In *RR*, the main objective is to find the relevant documents by retrieving only their necessary characteristics. In *RA*, we retrieve all the characteristics of the document, checking at the same time whether the document is needed for the query. Compared to *RR*, we perform only one read at the database, but we retrieve more data than necessary if the document is not needed for the query; as a result, *RR* is expected to perform better when the amount of data stored per node is much higher than the data needed to establish the necessity of the node. The necessity check, along with the rest of the query execution, is performed on the client. In the new MongoDB implementation, contrary to the initial Cassandra one, we adopt a more incremental (iterative) approach instead of returning all data in a single batch; this has increased the scope of global queries that can be executed without throwing an out-of-memory error.

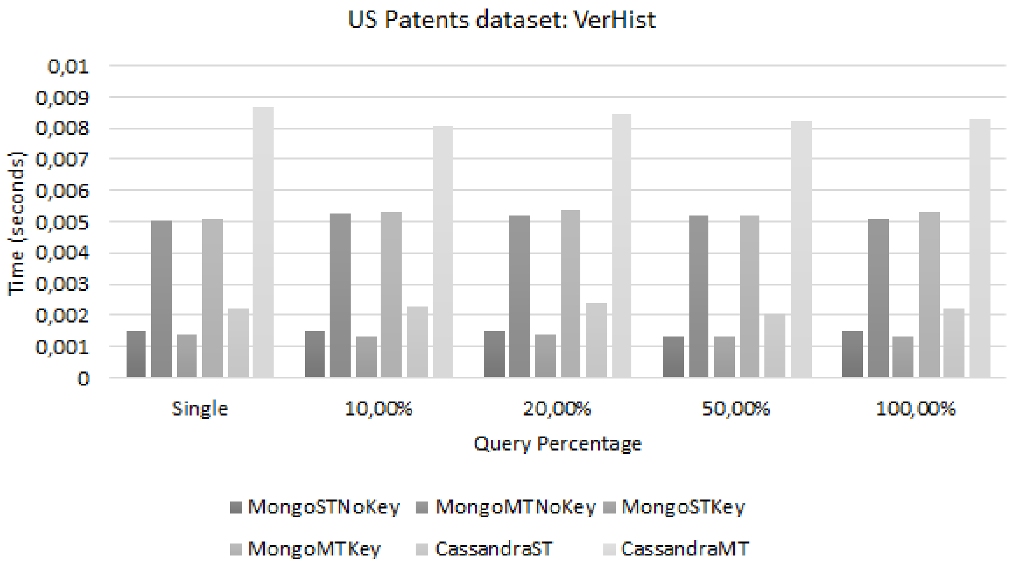

Figure 2 Results for the vertex history query on the US Patents dataset

However, the most notable difference between the two implementations is that MongoDB naturally lends itself to *in-database* query processing, so that the client gets only the data needed to compute the final results. This is achieved by submitting more complex queries that are supported by the MongoDB driver. To this end, we use the in-database MongoDB mechanisms to perform the necessity checks mentioned in the *RR* technique. Similarly to *RR*, the data needed for the final answer computations are returned incrementally to the client. As such, this approach has even lower space requirements on the client-side and at the same time, it allows for both the server and client working in parallel. It should be mentioned, that in some local queries (like onehop query), it may make sense to adopt an in-database query processing rationale, but this is beyond the scope of this paper.

4 Experiments

In the experiments, we use the same 4 queries as in [6] (Vertex History, One Hop, Average Vertex Degree, Vertex Degree Distribution) in 3 different datasets (hep-Th with 27.77K vertices, 352.8K edges and 156 snapshots; hep-Ph with 34.5K vertices, 421.6K edges and 132 snapshots; and US Patents with 3,774.8K vertices, 16.5M edges and 444 snapshots). We experiment with all MT and ST Cassandra and MongoDB combinations. For MongoDB, we test both key and nokey flavors and all modes of global query processing (*RA*, *RR* and *ID*). Each query is executed referring to a range of snapshots from 1 to all. We use a client application written in Java, and all the experiments were executed on a single node system with i5-3210M, 16GB RAM, and a 500GB SSD, while the client and the databases are collocated on the same machine.

4.1 Local queries

Regarding local queries, we investigated the vertex history query and one hop query query at the 3 datasets. We repeated each query 500 times and we report the average values. For each

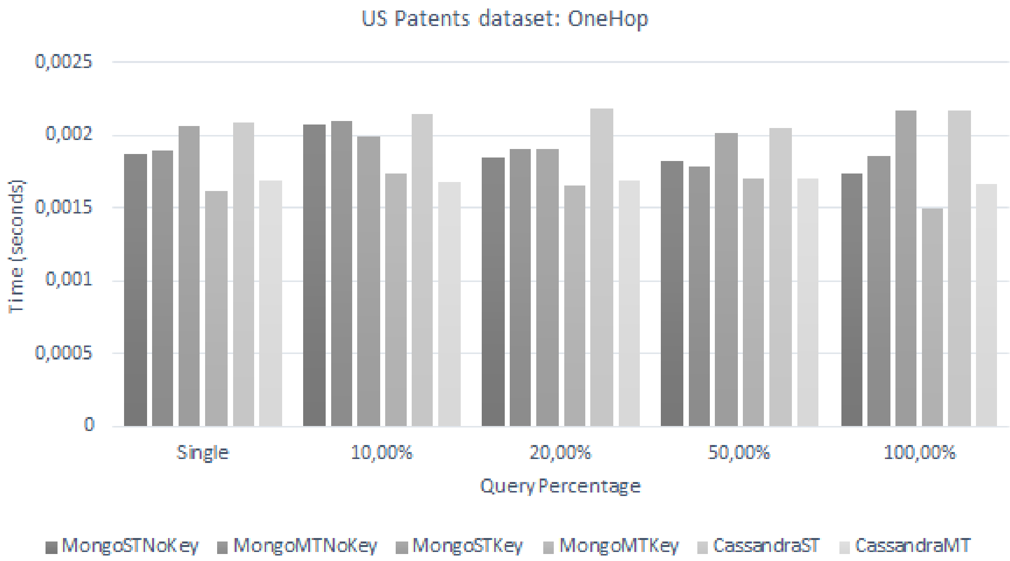

Figure 3 Results for the 1-hop query on the US Patents dataset

set of such runs we have a cold start. Fig. 2 presents the results for the US Patents dataset, but the results are similar for all datasets. In summary, for the vertex history query, ST outperforms MT by more than 70% (as also mentioned at [6]) while we observe an increase in performance at the best performing ST models executed in MongoDB by up to 42% (in average, it is 39.6% across all snapshot amounts) compared to the Cassandra ones. The best performing MT model in MongoDB is faster by up to 42% compared to its Cassandra counterpart, as well. Another observation is that the performance of the models does not depend on the amount of snapshots in the query. Finally, Key and NoKey settings (recall that in NoKey, the key is the default `_id` provided by MongoDB) have small differences (Key is faster by up to 8.5 %)

In the oneHop query, the observations are mixed but are still consistent across the three datasets (see Fig. 3 for the large dataset). Firstly, the differences between Cassandra and MongoDB are smaller with no dominant database system. More specifically, the differences between the two databases for any range of snapshots are up to 20 % when MongoDB performs better, and up to 6% when the dominant model is the Cassandra-based one, i.e., the differences between the two database systems are smaller than previously. Secondly, MT is better than ST in all cases by up to 26%, while the average performance improvement is 13.2% (due to the fact that only edge info needs to be retrieved). When considering only MT models, the differences between MongoDB and Cassandra do not exceed 10%. Finally, Key and NoKey have larger differences, up to 23 %. In all cases except one, for the MT model, the key version is the dominant one.

4.2 Global queries

For global queries, we demonstrate the results for both Hep-Th and US Patents datasets, since hep-Ph has similar results as hep-Th. The graphs include more query processing modes than previously, since we distinguish between *RA*, *RR* and *ID*.

Average Vertex Degree Query. The results are shown in Figure 4. Our observations

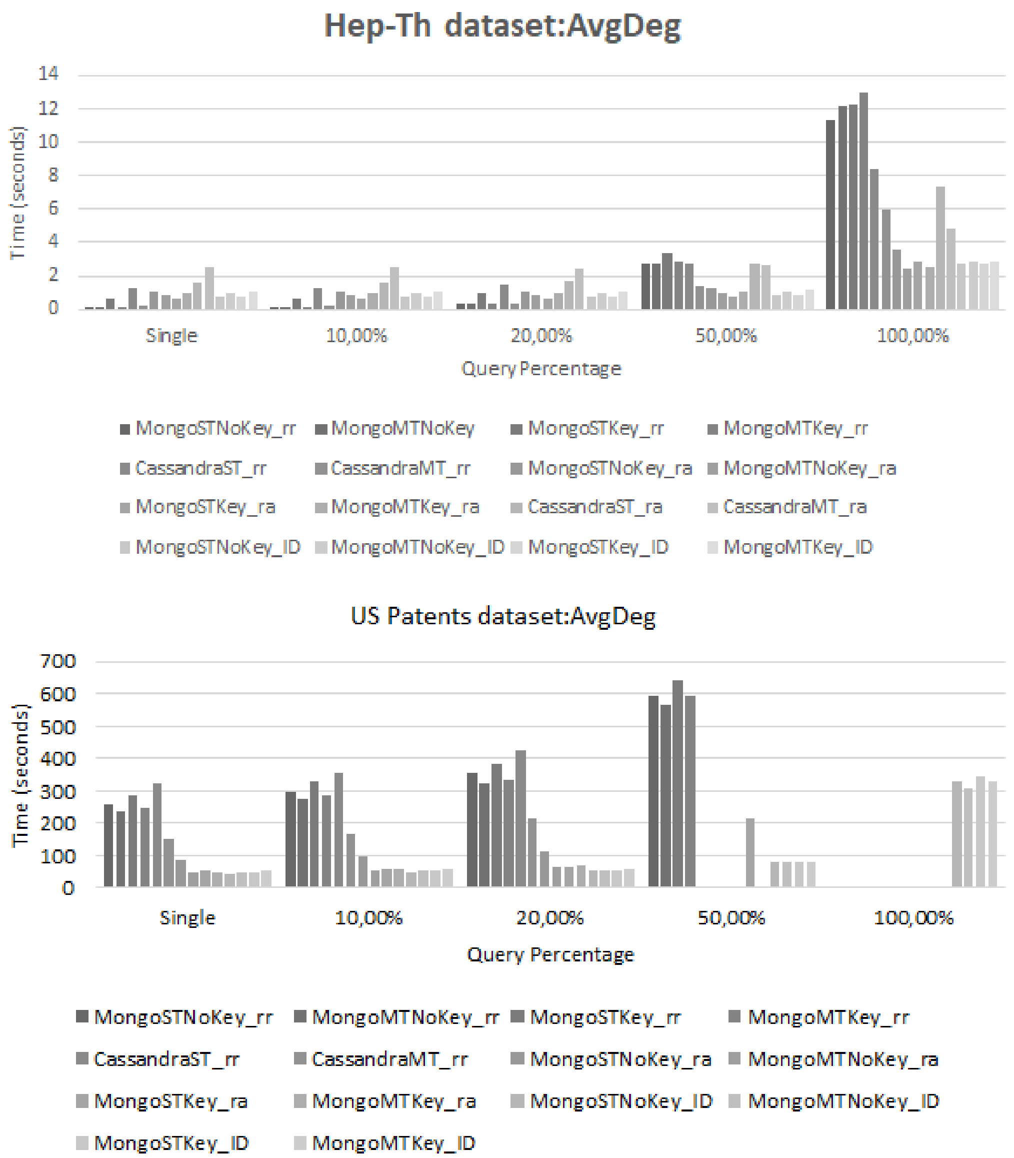

Figure 4 Results for the average degree query on the hep-Th (top) and US Patents (bottom) datasets

for the two datasets are summarized below.

1. For the big dataset (US Patents), the MongoDB in-database techniques are always the most efficient. Regarding the exact storage model, for snapshots in the range of [1-50%), ST models behave better, while for more snapshots, MT is superior. On average, *ID* MongoDB techniques improve upon the best performing Cassandra models (which manage to handle up to 20% snapshots) by up to 75.9% (speedup factor of 4×).
2. For the smaller datasets, when accessing 50% or 100% of the snapsots, the MongoDB *RA* approach is better than the *ID* approach, albeit by a small margin (on average, 11.5%). On the other hand, when accessing [1 to 20 %] of the graph snapshots, MongoDB *RR* approach is better than every other approach. When compared to *ID*, they improve the

performance on average by 84%, mostly due to the small intermediate results.

3. MongoDB solutions are superior to those of Cassandra, with the margin in performance increasing with more snapshots. When accessing all snapshots, the MongoDB speedup is 1.96× for the smaller dataset; for the larger dataset, MongoDB managed to return results when employing the *ID* technique, while Cassandra did not manage to run.
4. The *RR* and *RA* query processing methods perform differently in Cassandra and in MongoDB. In Cassandra's ST model, *RA* outperformed *RR* in 50% and 100% of snapshots by less than 12%. In the MT model, it outperformed only when accessing 100% of the graph (by 18.5%). On the other hand, in MongoDB, both for ST and MT, *RA* outperformed *RR* when the queries were applied on over 50% of the snapshots; the differences for ST were on average 73%, while, for MT, were 71.5%.
5. Overall, while MongoDB is always the main option, the best performing model differs. In smaller datasets, when the query accesses less than half of the snapshots, ST combined with *RR* is more efficient; for more snapshots MT combined with *RA* dominates. In the large dataset, *ID* is always the main option, but ST is more efficient when accessing less than 50% of the snapshots, while MT performs better otherwise.

Vertex Degree Distribution Query. The results are shown in Figure 5. Our main observations are the following:

1. MongoDB ST model combined with the *ID* technique manages to execute all queries over all snapshot ranges; no other combination of choices achieves this, e.g., Cassandra-based solutions can run queries only up to 20% snapshots.
2. For the big dataset, the MongoDB *RA* approach is the most efficient. When accessing up to 20% snapshots, MT combined with *RA* behave better and improve the performance by 77.3% on average. On the other hand, the only implementation that was able to execute on all snapshots was MongoDB MT combined with *ID*, which improved upon the best performing Cassandra models by 74% on average.
3. For the smaller datasets, MongoDB ST combined with *ID* is the best when accessing all snapshots, by up to 51% compared to the Cassandra implementation. When accessing 50% of the snapshots, MongoDB ST combined with *RA* is better than the best-performing Cassandra implementation by 48%. Finally, when the query access up to 20% of the snapshots, MongoDB MT combined with *RR* improves upon their Cassandra counterparts by 58.6% on average.
4. In Cassandra, *RA* does not increase the performance with minor exceptions. On the other hand, in MongoDB, *RA* always increases the performance by up to 80%.
5. The NoKey option performs more efficiently than Key in most cases but by a small margin, i.e., it does not decrease the best performing times by more than 10%.

Finally, to assess the impact of indices, we experimented with a MongoDB model without using indexes on any non-key characteristic. The experiment was performed on the Hep-Th dataset with global queries using only the ST model. The results show a serious performance degradation. More specifically, when we use indexes, the execution time is reduced by up to 98% (improvement by a factor of 41×). This speedup is observed for both global queries.

Space issues. While Cassandra requires less space to store the data since it builds fewer indices and adopts a different storage approach, MongoDB requires less memory on the client while executing the query. This is the result of the iterative approach that was adapted in MongoDB as well as from the *ID* query processing method. The space required for the three datasets (in increasing size) in Cassandra ST was 31.0 MB, 37.4 MB and 1.83 GB, and for MT 45.7 MB, 55.5 MB and 3.10 GB. The space for MongoDB ST was 89.70 MB, 107.37 MB,

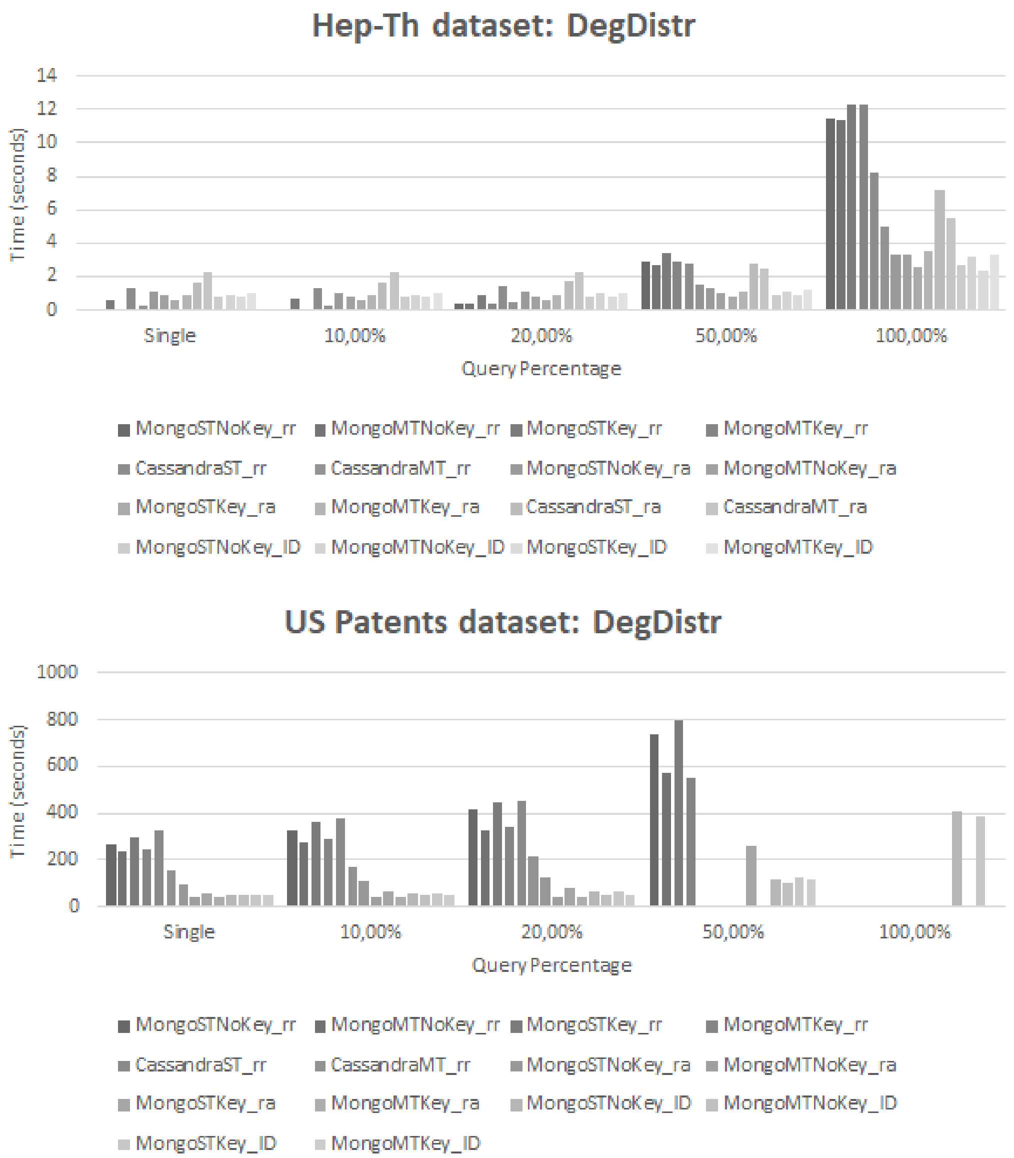

Figure 5 Results for the vertex degree distribution query on the hep-Th (top) and US Patents (bottom) datasets

4.84 GB, and for MT 218.34 MB, 260.87 MB and 10.96 GB, respectively. On the other hand, MongoDB has exhibited a speedup above 2× for insertions.

5 Summary

In this work, we have shown how the HiNode vertex-centric approach for storing time-varying graphs can be implemented in MongoDB. We have achieved significant improvements for global queries compared to the previous NoSQL based implementation (over 4X in some cases); the speedups were lower for local queries, but such queries are already performed efficiently in any vertex-centric implementation. Our vision is broader. We aim to develop a complete historical graph data management system through extending the described storage layer with more sophisticated query processing and optimization techniques.

References

1. Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
2. U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, 2016.
3. Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 997–1008, 2013.
4. Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, pages 77–88, 2016.
5. Andreas Kosmatopoulos, Kalliopi Giannakopoulou, Apostolos N Papadopoulos, and Kostas Tsichlas. An overview of methods for handling evolving graph sequences. In *Algorithmic Aspects of Cloud Computing*, pages 181–192. Springer, 2016.
6. Andreas Kosmatopoulos, Anastasios Gounaris, and Kostas Tsichlas. Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data. *Computing*, 03 2019. `doi:10.1007/s00607-019-00715-6`.
7. Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura. Hinode: an asymptotically space-optimal storage model for historical queries on graphs. *Distributed and Parallel Databases*, 2017. URL: `https://doi.org/10.1007/s10619-017-7207-z`.
8. Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, 2015.
9. Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
10. Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, 2012.
11. Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.
12. Betty Salzberg and Vassilis J Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.
13. Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. Timereach: Historical reachability queries on evolving graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 121–132, 2015.
14. Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013*, pages 505–516, 2013.
15. Sean R. Spillane, Jeremy Birnbaum, Daniel Bokser, Daniel Kemp, Alan G. Labouseur, Paul W. Olsen, Jayadevan Vijayan, Jeong-Hyon Hwang, and Jun-Weon Yoon. A demonstration of the G_* graph database system. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1356–1359, 2013.
16. Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Vittoria Colizza, Lorenzo Isella, Corinne Régis, Jean-François Pinton, Nagham Khanafer, Wouter Van den Broeck, and et al.

Simulation of an seir infectious disease model on the dynamic contact network of conference attendees. *BMC Medicine*, 9(1), 2011.

17 Yajun Yang, Jeffrey Xu Yu, Hong Gao, Jian Pei, and Jianzhong Li. Mining most frequently changing component in evolving graphs. *World Wide Web*, 17(3):351–376, 2014.