Computing manuscript No.
(will be inserted by the editor)

Hinode: Implementing a Vertex-centric Modelling Approach to Maintaining Historical Graph Data

Andreas Kosmatopoulos · Anastasios Gounaris · Kostas Tsichlas

the date of receipt and acceptance should be inserted later

Abstract Over the past few years, there has been a rapid increase of data originating from evolving networks such as social networks, sensor networks and others. A major challenge that arises when handling such networks and their respective graphs is the ability to issue a historical query on their data, that is, a query that is concerned with the state of the graph at previous time instances. While there has been a number of works that index the historical data in a time-centric manner (i.e. according to the time instance an update event occurs), in this work, we focus on the less-explored vertex-centric storage approach (i.e. according to the entity in which an update event occurs). We demonstrate that the design choices for a vertex-centric model are not trivial, by proposing two different modelling and storage models that leverage NoSQL technology and investigating their tradeoffs. More specifically, we experimentally evaluate the two models and show that under certain cases, their relative performance can differ by several times. Finally, we provide evidence that simple baseline and non-NoSQL solutions are slower by up to an order of magnitude.

Keywords Historical Queries · Historical Graphs · Evolving Graphs · Vertex-centric

A. Kosmatopoulos (✉) · A. Gounaris · K. Tsichlas
Department of Informatics, Aristotle University of Thessaloniki, Greece
E-mail: akosmato@csd.auth.gr

A. Gounaris
E-mail: gounaria@csd.auth.gr

K. Tsichlas
E-mail: tsichlas@csd.auth.gr

1 Introduction

A key characteristic of the past decade in data management is the ever increasing amount of data generated by real world applications on a daily basis. A significant portion of the produced data originates from *networks*, i.e. distinct entities and the relationships formed between them. Most of these networks such as social media networks (Facebook, Twitter etc.), communication networks, collaboration networks and others can be modeled using a *graph data structure* with entities corresponding to *vertices* and the relationships between them corresponding to *edges* in the graph; both the vertex and edge elements may be annotated by *attributes* such as name and weight respectively.

A common feature of such graphs 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 (*snapshots*) in a spatially and temporally efficient way. More specifically, consider a graph corresponding to a social network that is comprised of millions of users (vertices) and the friendship relationships between them (edges). By examining the graph through two subsequent days we witness a number of newly created, removed and altered vertices or edges and a significant fraction of the graph that has remained the same across the two days. A system that aims to efficiently store all the snapshots of such a graph should employ techniques that mitigate the presence of unaltered data between different snapshots (i.e. take advantage of the commonalities between snapshots and refrain from storing duplicate data across snapshots).

There have been two main approaches with regard to a system's design [7], the *time-centric* approach and the *entity-centric* approach. 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.

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 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. In the first case, a query aims to evaluate a measure at a particular time instance (e.g. shortest path length between two vertices at a particular time instance), while in the latter case a query's objective is to extract information regarding a measure's evolution through snapshots (e.g. average shortest path length between two vertices in the ten first snapshots).

From the above paragraphs we expect that systems built following the time-centric approach are more suited towards evaluating global queries. 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.

In our previous work in [9], we introduced the first purely entity-centric, and more specifically, vertex-centric model for maintaining graph historical data, termed as *HiNode*. Its strongest point is that it builds upon a theoretical storage model that is asymptotically space-optimal. Its initial implementation, hereafter termed as *HiNode-*G^*[1], was based on extensions to the G^* [10,18] parallel graph database. This design choice incurred severe limitations regarding the efficiency and scalability of the *HiNode-*G^* prototype. In this work, we propose to leverage NoSQL as the underlying database technology thus forming the second version of *HiNode*, called *HiNode-NoSQL* hereafter. However, simply switching to a different underlying technology does not necessarily imply performance improvements in retrieval tasks. The technical contribution of this work is to (i) investigate and compare different NoSQL design techniques for the *HiNode-NoSQL* system, and (ii) provide concrete insights into the strengths and weaknesses of each alternative in terms of supporting retrieval queries on evolving networks.

More specifically:

1. We investigate two approaches to vertex-centric modelling and storage with different strengths and weaknesses.
2. We propose querying models on top of these models.
3. We show that under certain cases, their relative performance can differ by several times.
4. Finally, we provide evidence that simple baseline and non-NoSQL solutions are slower by up to an order of magnitude.

The rest of the paper is structured as follows: Section 2 provides a detailed background on the vertex-centric approach in [9]. Section 3 reviews related work in the field of historical graph data. Section 4 describes the alternative models in detail. Section 5 compares the proposals through extensive experimental evaluation. Finally, in Section 6 we conclude our work.

2 Background and Motivation

We begin by more formally defining the terms used throughout the rest of this work and then move on to describe the overall problem motivation and our approach.

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

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 direction. Under this notion of time $\mathcal{G} =< G_1, G_2, \ldots >$ corresponds to an *evolving graph sequence* of snapshots that are to be stored and maintained appropriately. The absence of a "final" or last snapshot in $\mathcal{G}$ denotes that the sequence is constantly evolving as time progresses. Note that given the definition of deltas, a snapshot G_i can be conceptually obtained by applying a set of deltas (vertex or edge insertions or removals) to snapshot G_{i-1} and vice versa.

Without loss of generality, throughout the rest of this work we will use a particular graph as a running example. In this example, each graph vertex possesses two attributes: a name and a color. Furthermore, graph vertices are connected through edges with each edge having a label and a weight attribute. Figure 1 depicts three consequent snapshots of the example graph.[2] Snapshot G_1 is obtained by changing the name of v_4 in G_0 from d to lbl and snapshot G_2 is obtained from G_1 by inserting v_5 and an edge from v_2 to v_5.

In order to efficiently store and handle evolving graph sequences we developed the HiNode system prototype. The primary focus of the HiNode system is on storage, and more specifically how storage requirements can be minimized while efficiently supporting retrieval tasks. Retrieval tasks are either complete queries on evolving graphs or part of them, when more complex graph analyses on evolving graphs are performed.

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*. The diachronic node utilizes external memory Interval Trees and B-Trees to model a vertex's history as a collection of intervals that permit geometric operations upon them. 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. As an example, for the graph in Figure 1 the diachronic node for b at the time of G_2 would contain an Interval Tree for all changes related to b as well as three B-Trees for b's label, color and edge to e. Furthermore, due to the local handling of history, HiNode performs well on local queries and the authors further demonstrate that HiNode-G^* is competitive on global queries as well [9].

The G^* parallel graph processing system [10] is, in essence, a combination of the Copy+Log storage method (see Section 3) and the vertex-centric modeling approach. The system aims to exploit the commonalities found between successive snapshots in the history of a graph to reduce the total space footprint by only storing each version of a vertex only once regardless of the number of snapshots it can be found in. The indexing mechanism of G^* is

[2] For reasons of clarity edge labels, vertex colors and edge weights are not shown.

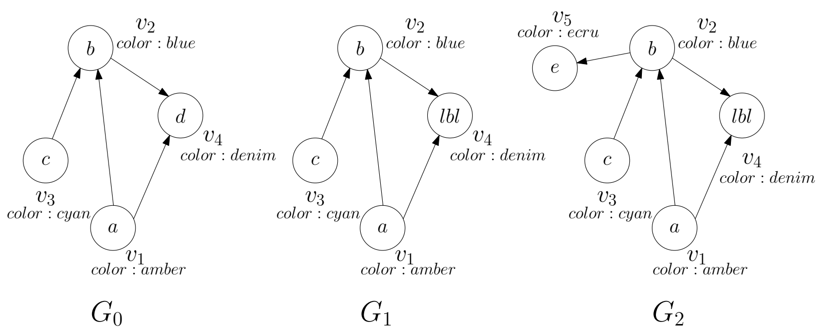


Fig. 1 Example graph sequence

a set of *"(vertexID, diskLocation)"* pairs for each vertex version, formed into collections based on the vertices stored by each G^* server. As an example, for the graph in Figure 1 a G^* server assigned to store vertex b would store two versions of b: One for $\{G_0, G_1\}$ and one for G_2. Finally, it is worth noting that the index of G^* resides in memory while the vertex data is stored on disk. HiNode-G^* was essentially built on top of the G^* query evaluation modules by substituting the above indexing and storage modules with a pure vertex-centric module.

This work aims to substitute the storage technology of HiNode-G^* with NoSQL databases in order to avoid the underlying limitations of the G^* storage engine. More specifically, the G^* system was built on the premise of storing the aforementioned pairs in a distributed fashion utilizing slotted pages. HiNode-G^* could only operate on datasets of restricted size due to the overall architecture and implementation of the storage and indexing modules. By adopting the use of NoSQL databases as the fundamental method of storing data we aim to achieve a threefold goal: (i) take advantage of the natural key-value nature found in a graph's entities (e.g. an edge has a specific name and weight at a specific time instance), (ii) take advantage of the NoSQL database's inherent scalability especially for data-intensive applications and graphs; and (iii) to benefit from the engineering maturity of NoSQL tools. With regard to HiNode-NoSQL we propose two design techniques (Single-Table and Multi-Table) that each possesses its own tradeoffs between queries.

Finally, to underline the necessity of a NoSQL approach we present some indicative results in Figure 2. Both of the two vertex-centric models for HiNode-NoSQL that will be described in Section 4 significantly outperform HiNode-G^*[3] even in the case of a dataset[4] with a relatively limited number of vertices and edges thus solidifying the need for a more scalable approach. The adoption of a NoSQL approach in favor to HiNode-G^* is not a trivial task and comes with several design and query execution issues for which their addressing constitutes the result of this work.

[3] The queries executed were Average Vertex Degree, Degree Distribution and One-Hop Neighborhood Retrieval on the last snapshot of the sequence - see Section 5.1

[4] citHep-Th SNAP Dataset [11] - see Section 5.1

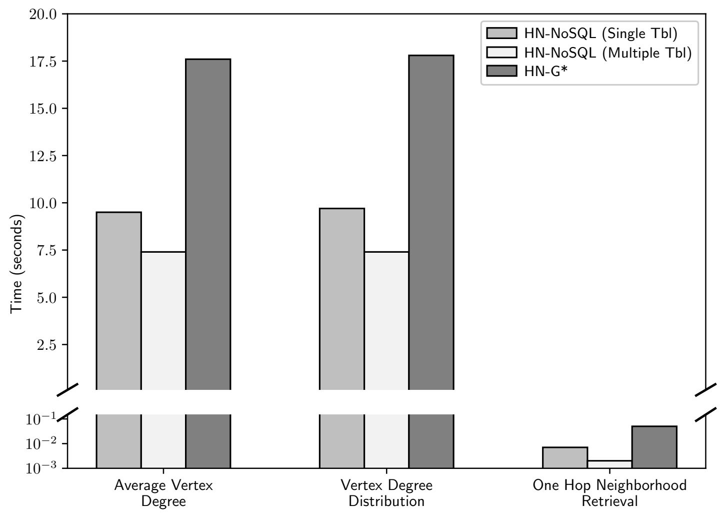


Fig. 2 Comparison of HiNode-G^* and HiNode-NoSQL

3 Related Work

Over the past few years there has been a plethora of graph processing systems or libraries developed to handle the increasingly large graphs that are produced by modern real world applications. Systems such as GraphX [4], Giraph [2], Pregel [12], Trinity [17] and others are mainly used to efficiently store, retrieve and query single snapshots of large graphs. However, in most cases these systems do not naturally support historical graph data and are generally not suited to handling evolving graphs. The proposals that are the closest to our work are systems and modelling approaches that are able to handle graphs together with their historical information. A survey that reviews work conducted in maintaining historical graph data can be found in [8].

One of the first approaches to handling historical graph data originates from Salzberg and Tsotras [14]. In their work, they present the *Copy* and *Log* methods which suggest storing the entire snapshot whenever a change is performed and storing a list of all the changes as they occur in the graph respectively. These two methods usually serve as guidelines and can be combined into the *Copy+Log* method (i.e. periodically storing the entire snapshot every set number of changes) that serves as a compromise between total time and space cost.

Khurana and Deshpande proposed DeltaGraph [6] a system which employs a tree-like index structure that can perform singlepoint and multipoint graph retrieval. The authors furthermore proposed the Temporal Graph Index (TGI) [7] which extended DeltaGraph by incorporating mechanisms that enabled vertex-centric operations. Labouseur et al. [10,18] proposed the parallel graph database G^* which takes advantage of the commonalities found between snapshots to reduce the total space cost. As explained in detail in the previous section, Kosmatopoulos et al. developed HiNode [9], which is asymptotically space-optimal and the first implemented purely vertex-centric system but has been only implemented in a simplified form on top of G^*. The main novelty of this work is that it makes a deep dive into implementation issues and related design choices when switching to NoSQL technology in order to improve efficiency and scalability.

There have also been various attempts at indexing the historical data so as to enable evaluation of specific queries and measures. Ren et al. [13] and Huo et al. [5] focus on shortest path query evaluation and its evolution through the graph history. Akiba et al. [1] focus on evaluating historical distance queries between two vertices and reporting the time instances of distance change. Yang et al. [19] target the problem of finding the most frequently changing components in an evolving graph. Finally, Semertizidis et al. tackle the problem of historical reachability queries [16] and durable graph pattern queries [15].

4 Modelling

In this section, we propose two vertex-centric models for maintaining historical graph data and provide an implementation through Apache Cassandra. We describe the logic behind each model and discuss advantages and potential trade-offs using the example graph sequence found in Figure 1.

4.1 Single Table Model

The first vertex-centric model, called the Single Table model (ST), uses a single table to model all history in the graph. More specifically, each row in the table corresponds to a particular vertex's entire history and is equivalent to the contents of a diachronic node in HiNode.

```
CREATE TYPE histgraph.attribute (value text, start text,
          + end text);
CREATE TYPE histgraph.edge (label text, weight text,
          + otherEnd text, start text, end text);

CREATE TABLE histgraph.dianode (vid text,
          + start text, end text,
          + name list<frozen<attribute>>,
          + color list<frozen<attribute>>,
          + incoming_edges map<text, frozen<list<edge>>>,
          + outgoing_edges map<text, frozen<list<edge>>>,
          + PRIMARY KEY (vid, start, end));
```

In Apache Cassandra we create new data types for attributes and edges and name the single table “dianode”. The “attribute” data type corresponds to

vid	start	end	name	color	incoming_edges	outgoing_edges
1	0	2	[{value: ’a’, start: ‘0’, end: ‘2’}]	[{value: ’amber’, start: ‘0’, end: ‘2’}]	null	‘2’: [{label: ’elbl1’, start: ‘0’, end: ‘2’}], ‘4’: [{label: ’elbl2’, start: ‘0’, end: ‘2’}]
2	0	2	[{value: ’b’, start: ‘0’, end: ‘2’}]	[{value: ’blue’, start: ‘0’, end: ‘2’}]	‘1’: [{label: ’elbl1’, start: ‘0’, end: ‘2’}], ‘3’: [{label: ’elbl5’, start: ‘0’, end: ‘2’}]	‘4’: [{label: ’elbl3’, start: ‘0’, end: ‘2’}], ‘5’: [{label: ’elbl4’, start: ‘2’, end: ‘2’}]
3	0	2	[{value: ’c’, start: ‘0’, end: ‘2’}]	[{value: ’cyan’, start: ‘0’, end: ‘2’}]	null	‘2’: [{label: ’elbl5’, start: ‘0’, end: ‘2’}]
4	0	2	[{value: ’d’, start: ‘0’, end: ‘1’}, {value: ’lbl’, start: ‘1’, end: ‘2’}]	[{value: ’denim’, start: ‘0’, end: ‘2’}]	‘1’: [{label: ’elbl2’, start: ‘0’, end: ‘2’}], ‘2’: [{label: ’elbl3’, start: ‘0’, end: ‘2’}]	null
5	2	2	[{value: ’e’, start: ‘2’, end: ‘2’}]	[{value: ’ecru’, start: ‘2’, end: ‘2’}]	‘2’: [{label: ’elbl4’, start: ‘2’, end: ‘2’}]	null

Table 1 The contents of ST for the graph sequence of Figure 1. To facilitate readability, the “otherend” and “weight” edge attributes are not shown

an interval of a particular value that is valid between “start” and “end”. The incoming edges of a vertex are maps that store “(source_vertex, list_of_edges)” key-value pairs where “list_of_edges” is a collection of all the edges that have occurred at some time in the history between these two vertices. A similar definition applies to the outgoing edges.

Finally, we note that by this primary key declaration, the rows are partitioned to a server according to their vertex ID thus resulting in each vertex’s history completely residing in a single server. This results in each change being stored only once in the corresponding vertex row and enables faster single vertex query and local query evaluation since we avoid any unnecessary communication between servers. The downside when using ST is that whenever we need to access a vertex at a particular time instance the system must “unpack” all the collections after retrieving the relevant row. Even though this is performed on the client side it adds a significant time cost. Additionally, for application domains that are characterized by a high attribute update rate, the collection size might exceed the maximum collection size supported by Cassandra. This can be aleviated by stopping the use of a particular collection after a set amount of items and creating a new one, thus retaining the space efficiency of the ST model in general.

The contents of ST for the graph sequence of Figure 1 are shown in Table 1.

4.2 Multiple Tables Model

To avoid the time slowdowns that occur with the employment of collections, we propose the second vertex-centric model called Multiple Tables model (MT). In MT we use a single table for each vertex attribute and we differentiate between incoming and outgoing edges by using a single table for each of their corresponding attributes. Additionally, MT uses three tables to denote the "lifetime" of vertices and outgoing edges and incoming edges respectively.

```
CREATE TABLE histgraph.vertex (vid text,
        + start text, end text,
        + PRIMARY KEY (vid, start, end));
CREATE TABLE histgraph.vertex_name (vid text,
        + name text, timestamp text,
        + PRIMARY KEY (vid, timestamp)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC);
CREATE TABLE histgraph.vertex_color (vid text,
        + color text, timestamp text,
        + PRIMARY KEY (vid, timestamp)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC);
CREATE TABLE histgraph.edge_outgoing (
        + start text, end text,
        + sourceID text, + targetID text,
        + PRIMARY KEY (sourceID, start, end, targetID));
CREATE TABLE histgraph.edge_label_outgoing (
        + label text, timestamp text,
        + sourceID text, targetID text,
        + PRIMARY KEY (sourceID, timestamp, targetID)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC,
        + targetID DESC);
CREATE TABLE histgraph.edge_weight_outgoing (
        + weight text, timestamp text,
        + sourceID text, targetID text,
        + PRIMARY KEY (sourceID, timestamp, targetID)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC,
        + targetID DESC);
CREATE TABLE histgraph.edge_incoming (
        + start text, end text,
        + sourceID text, targetID text,
        + PRIMARY KEY (targetID, start, end, sourceID));
CREATE TABLE histgraph.edge_label_incoming (
        + label text, timestamp text,
        + sourceID text, targetID text,
        + PRIMARY KEY (targetID, timestamp, sourceID)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC,
        + sourceID DESC);
CREATE TABLE histgraph.edge_weight_incoming (
        + weight text, timestamp text,
        + sourceID text, targetID text,
        + PRIMARY KEY (targetID, timestamp, sourceID)
        + ) WITH CLUSTERING ORDER BY (timestamp DESC,
        + sourceID DESC);
```

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

(a) vertex

vid	timestamp	name
1	0	a
2	0	b
3	0	c
4	0	d
4	1	lbl
5	2	e

(b) vertex_name

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

(c) vertex_color

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

(d) edge_incoming

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

(e) edge_outgoing

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

(f) edge_label_incoming

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

(g) edge_label_outgoing

Table 2 The contents of MT for the graph sequence of Figure 1. Since the edge "weight" tables are similar to their edge "label" counterparts they are not shown

In MT we elect to store time instances of change on a vertex or edge attribute ("timestamp") as opposed to explicit intervals since the underlying intervals can be trivially inferred. The strong point of MT is that basic queries can be directly evaluated through querying Apache Cassandra with minimal client involvement as opposed to ST (e.g. to retrieve a particular vertex at a specific time instance the system queries "vertex_name", "vertex_color" and "edge_label_outgoing|incoming", "edge_weight_outgoing|incoming" for all the relevant edges). Two weak points of MT are that in order to guarantee that the system partitions all the vertex information into the same server some of the data need to be repeated. Furthermore, even though MT avoids the use of collections, it requires a greater amount of read operations to implement a basic operation compared to ST. This could result in a significant time cost if the underlying graph has vertices or edges with a large number of attributes.

The contents of MT for the example in Figure 1 are shown in Table 2.

4.3 Querying Modes

The two models proposed in the previous section are vertex-centric and thus inherently suited for the execution of local queries. 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*), 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 $[t_s, t_e]$ (Algorithm 1).

Algorithm 1 *retrieve_all*

```
Input: Time range $[t_s, t_e]$
Output: Data from vertices that exist in $[t_s, t_e]$
 1: if model == ST then
 2:     $\mathcal{S} \leftarrow$ "SELECT * FROM dianode"
 3:     for each diachronic node $D_v \in \mathcal{S}$ do                         ▷ Performed client-side
 4:         if $D_v.end \leq t_s$ or $t_e \leq D_v.start$ then
 5:             $\mathcal{S} \leftarrow \mathcal{S} \setminus \{D_v\}$
 6:         end if
 7:     end for
 8: else if model == MT then
 9:     for each table $T$ in MT (i.e. "vertex", "edge_outgoing" etc.) do
10:         $\mathcal{S} \leftarrow$ "SELECT * FROM T"
11:         for each row $r \in \mathcal{S}$ do                                ▷ Performed client-side
12:             if $r.end \leq t_s$ or $t_e \leq r.start$ then
13:                 $\mathcal{S} \leftarrow \mathcal{S} \setminus \{r\}$
14:             end if
15:         end for
16:         Join remaining results from each table $T$ on vertex ID   ▷ Performed client-side
17:     end for
18: end if
19: return $\mathcal{S}$
```

In the second mode (termed *retrieve_relevant*), each model first determines the vertices that are "alive" at $[t_s, t_e]$ and then retrieves them. More specifically, in the ST model, we first query for the "start" and "end" timestamp of each diachronic node in "dianode" and then, we retrieve from "dianode" all diachronic nodes whose "start" or "end" value belong in $[t_s, t_e]$, while in the MT model we follow a similar approach using the *vertex* table (Algorithm 2). Mode *retrieve_relevant* takes into account the provided time range and is more versatile compared to *retrieve_all* which should be mainly used for queries that involve the majority of the vertices in the sequence (i.e. "What is the average vertex degree in each snapshot of the sequence stored so far?").

While in ST the implementation of *retrieve_relevant* 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 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 range queries for two or more clustering columns (i.e. "start" and "end") for the sake of efficiency, we fetch the relevant data with

Algorithm 2 *retrieve_relevant*

```
Input: Time range $[t_s, t_e]$
Output: Data from vertices that exist in $[t_s, t_e]$
 1: if model == ST then
 2:     $\mathcal{V} \leftarrow$ "SELECT vid, start, end FROM dianode"
 3:     for each row $r \in \mathcal{V}$ do                                  ▷ Performed client-side
 4:         if $r.end \leq t_s$ or $t_e \leq r.start$ then
 5:             $\mathcal{V} \leftarrow \mathcal{V} \setminus \{r\}$
 6:         end if
 7:     end for
 8:     $\mathcal{S} \leftarrow$ "SELECT * FROM dianode WHERE vid IN $\mathcal{V}$"
 9: else if model == MT then
10:     $\mathcal{V} \leftarrow$ "SELECT * FROM vertex"
11:     for each vertex $v \in \mathcal{V}$ do                               ▷ Performed client-side
12:         if $v.end \leq t_s$ or $t_e \leq v.start$ then
13:             $\mathcal{V} \leftarrow \mathcal{V} \setminus \{v\}$
14:         end if
15:     end for
16:     $\mathcal{S} \leftarrow$ "SELECT * FROM MT.{all tables} WHERE {vid, sourceID, targetID} IN $\mathcal{V}$"
17:     Filter from $\mathcal{S}$ all data with timestamp $\notin [t_s, t_e]$   ▷ Performed client-side
18: end if
19: return $\mathcal{S}$
```

a timestamp larger than t_s and then filter, all data with a timestamp less than t_e at the client side. In the following experimental evaluation section, we primarily employed the *retrieve_relevant* method due to its overall time efficiency (see Section 5.6).

5 Experimental Evaluation

In this section, we experimentally evaluate the two proposed vertex-centric models and compare their relative efficiency.[5] We begin by providing query definitions, dataset descriptions before moving on to the models' comparison on a single node setting. We start with a single-node setting in order not to allow automated parallelism and elasticity features of Cassandra to be involved and therefore, draw clearer observations about each model's performance. Furthermore, we repeat the experiments in the presence of a cluster data center and report our findings. Finally, we outline the efficiency of a state-of-the-art graph DBMS in the context of historical graph data and conclude with some overall observations.

5.1 Query Definitions, Experimental Environment and Dataset Description

We demonstrate the different trade-offs by each of our proposed models by performing experiments using four different queries. More specifically, we implemented the queries shown on Table 3 and executed them on query ranges

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

Query	Query Description
Vertex History Retrieval VerHist(vID, <Snapshots>)	For a given vertex vID and a given range of snapshots, return the state of vID at each snapshot
One Hop Neighborhood OneHop(vID, <Snapshots>)	For a given vertex vID and a given range of snapshots, return the neighbors of vID at each snapshot
Average Vertex Degree AvgDeg(<Snapshots>)	For each snapshot in the query range, compute the average degree of its vertices and report it
Vertex Degree Distribution DegDistr(<Snapshots>)	For each snapshot in the query range, count the vertices with each possible degree and report them

Table 3 Query Definitions

Dataset	Vertices	Edges	Snapshots
hep-Th	27770	352807	156
hep-Ph	34546	421578	132
US Patents	3774768	16518948	444
Synthetic	5100000	30600000	100

Table 4 Evaluation datasets. The number of vertices and edges refer to the last snapshot of the sequence

of varying size (i.e. 1 snapshot of the sequence and 10%, 20%, 50%, or 100% of the sequence's snapshots).

We performed experiments on real world datasets originating from the SNAP Dataset Collection [11]. We selected three datasets with the first two constituting a graph representation of an arXiv citation network for two different scientific domains; "hep-th" (from January 1990 to April 2003) and "hep-ph" (from February 1992 to February 2003) and the last one ("USPatents") a cross-citation network of US utility patents granted between 1963 and 1999. In each case, we split the citations based on their date of occurrence and ended up with a sequence of monthly snapshots.[6]

In addition to the aforementioned settings, we aimed to discover how the two models handle synthetic scale-free graphs. To this end, we produced a synthetic dataset that follows the Barabási-Albert (BA) [3] scale-free graph model and simulates plenty of real world phenomena and circumstances. In the Barabási-Albert model, each snapshot in the sequence is generated by inserting new vertices that are connected with a set of already existing vertices. The selection of the already existing vertices that will serve as endpoints is done in a preferential manner with vertices having a large degree being more probable of being selected. Table 4 summarizes the datasets used in this work.

The two models were implemented using Apache Cassandra and all experiments were performed through a client application written in Java. All the single-node experiments were run on an Intel Xeon CPU E5-2620, 64GB RAM machine while the multi-node experiments were run on two machines AMD FX-9370 and Intel Core i7-3770K with 32GB RAM each respectively. In the single-node setting the client application was run on a i7-3770K, 32GB RAM

[6] Due to difficulties in assigning some specific edges to a particular snapshot we removed 0.4% of the total edges in the "hep-th" and "hep-ph" datasets and 0.04% edges of the "US-Patents" dataset

machine while in the multi-node setting in a E5-2620, 64GB RAM machine. In all cases the Apache Cassandra nodes were connected through a private 1Gbit network.

As a final note, this work serves as a study of different implementation approaches to vertex-centric models and a comparison between their behavior under different type of queries. As a result, matters of efficient query execution (such as data distribution and replication, asynchronous query execution, paging etc.) are beyond the scope of this paper and will be tackled in future work. Furthermore, in order to maintain a design that is independent of the graph domain, we omitted using specific data types for any vertex or edge attributes.

5.2 A Baseline Approach

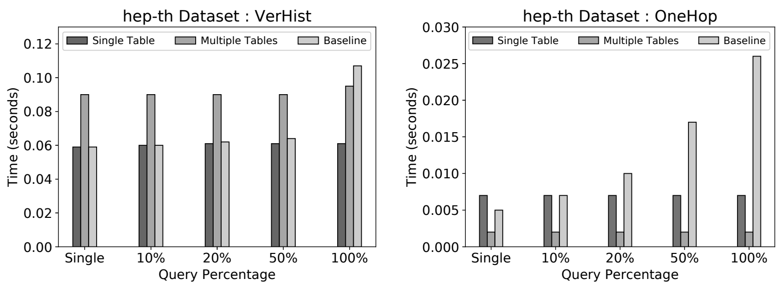
We implemented a modified version of the ST model that will serve as a simple baseline algorithm called HiNode-Baseline. This version, which can be conceived as a model similar to the Copy+Log model, splits the data to be indexed according to a set amount of time instances (e.g. every 20 snapshots).

For each resulting subset of the sequence's snapshots we build an independent ST model index based only on the entities (and their data) that exist on the corresponding snapshot range. Each vertex or edge that spans multiple snapshot ranges is indexed in each ST model for which it is valid. The resulting HiNode-Baseline index is comprised of multiple smaller ST indices each corresponding to its own set of assigned snapshots that are akin to smaller subsequences.[7] In the event of a query that spans multiple snapshot ranges, we query each ST index for their respective answer and combine the results. The baseline approach permits a lateral comparison between our two proposed models and a model designed on vertex-centric principles that also incorporates time as a design mechanism.

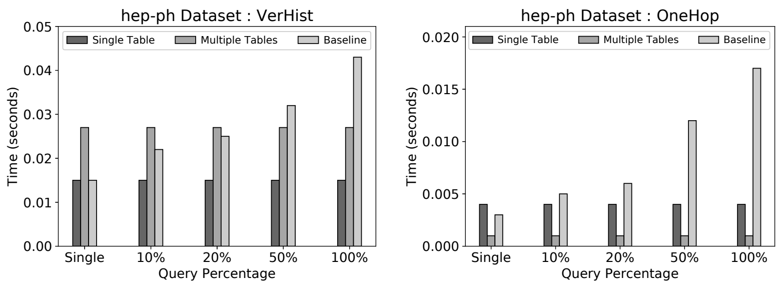
5.3 Results in a Single-Node Setting

We begin by reporting the results for the two local queries of Table 3 (VerHist and OneHop) in Figure 3, where the range of queried snapshots varies from a single snapshot to the complete history. To avoid any randomness induced by the query range selection, we repeated the experiments on query ranges from approximately the start, the middle and the end of the sequence and averaged over their times. Our primary focus is to investigate the relative difference in execution time (each execution time decrease is defined with respect to the slower model in each case). The key observations drawn from Figure 3 are the following:

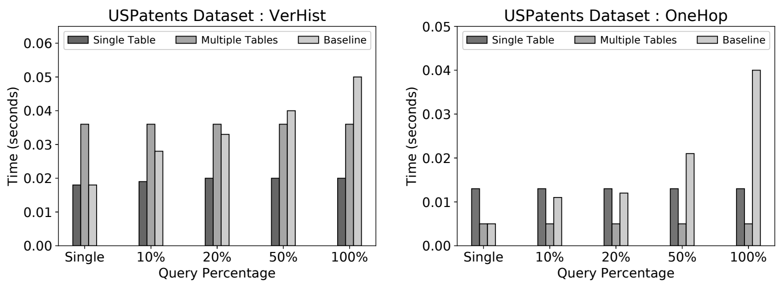
[7] As an example, a sequence of 100 snapshots that gets indexed every 20 snapshots would be comprised of five smaller ST indices whereas a vertex that exists in the first 75 snapshots would only be present in the first 4 smaller ST indices



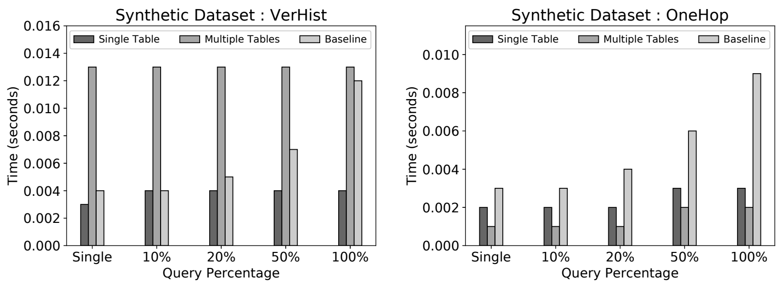
(a) Hep-th Dataset



(b) Hep-ph Dataset



(c) US Patents Dataset



(d) Synthetic Dataset

Fig. 3 Local queries on the datasets of Table 4 - Left Col.: VerHist, Right Col.: OneHop

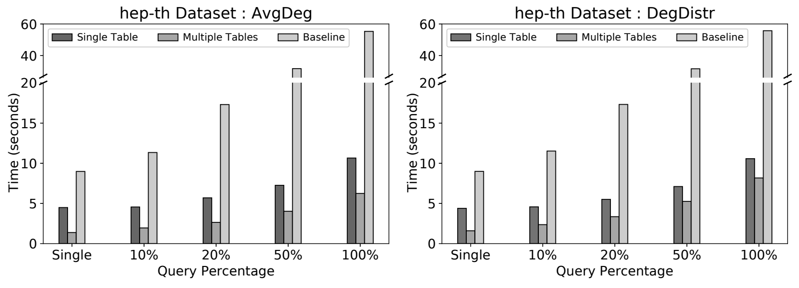
1. There is no clear winner between the two models for the two queries. ST outperforms MT in VerHist by up to 56%, 44%, 50% and 69% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.
2. MT executes OneHop faster than ST by up to 71%, 75%, 58% and 69% in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.
3. ST gradually outperforms the baseline in VerHist as query ranges grow larger by up to 70%, 65%, 64% and 66% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic” respectively. In OneHop, a similar pattern occurs with ST outperforming the baseline by up to 73%, 76%, 67% and 66% lower execution time in the four previous datasets.
4. ST and MT execution times remain relatively unaffected by the query range.

The above results can be attributed to the following reasons. With regard to VerHist, ST fetches the relevant row’s contents directly from its single table “dianode”, while MT queries each of its tables for any rows related to the queried vertex thus inducing an additional slowdown due to the extra Cassandra queries compared to ST. In OneHop, we only fetch the relevant column from ST (“dianode.outgoing_edges”) and we only query the relevant table (“edge_outgoing”) from MT, thus ST is burdened by the extra cost of handling and flattening a collection. Since we follow the *retrieve_relevant* querying method (Lines 8, 16, 17 in Algorithm 2) and the total execution time is dominated by the time required to fetch the relevant data from Cassandra, the local query times for ST and MT remain practically the same regardless of the query range. Finally, the baseline’s time cost can be attributed to the total number of smaller indices that have to be queried as query ranges become progressively larger and span multiple snapshot ranges

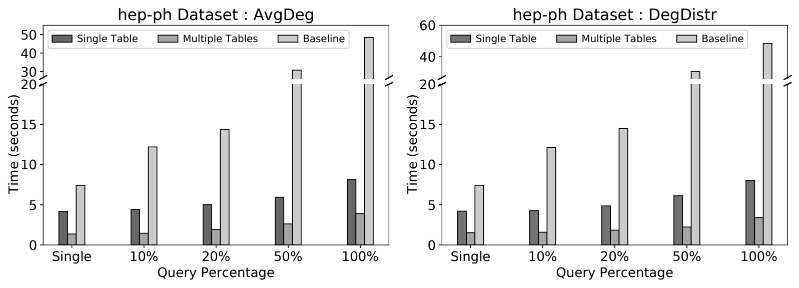
We move on to the results for the two global queries of Table 3 (AvgDeg and DegDistr) in Figure 4. The key observations to be made are:

1. MT is the best performing model in both global queries AvgDeg and DegDistr. MT outperforms ST in AvgDeg by up to 41%, 52%, 39% and 54% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.
2. Furthermore, MT executes DegDistr faster than ST by up to 63%, 57%, 58% and 59% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.
3. Baseline becomes progressively slower as the query range becomes larger.

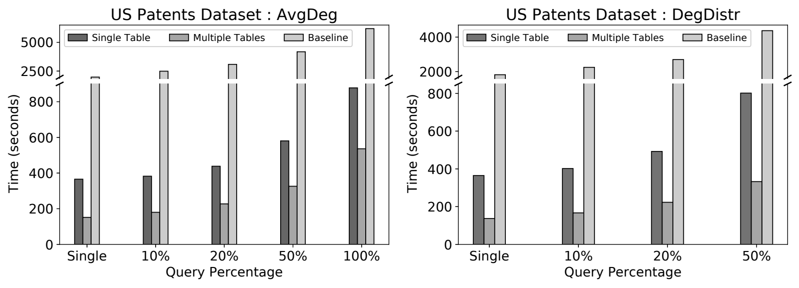
The observed results can be explained by the following reasons. In both AvgDeg and DegDistr, we follow the *retrieve_relevant* querying method and retrieve the outgoing edges of all relevant vertices (“dianode.outgoing_edges” in ST and “edge_outgoing” in MT). Since we restrict the fetched data to these particular columns and ST is burdened by the use and flattening of collections, MT outperforms ST in each case with both models requiring more time for larger snapshot ranges due to the fact that more results are returned as part of each Cassandra query. Additionally, similarly to the local queries, the baseline



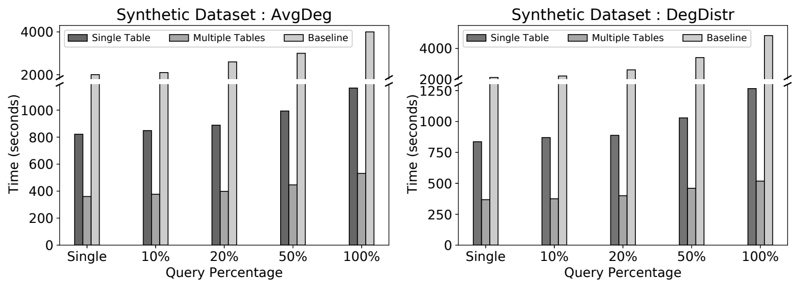
(a) Hep-th Dataset



(b) Hep-ph Dataset



(c) US Patents Dataset. DegDistr(100%) did not run due to insufficient memory



(d) Synthetic Dataset

Fig. 4 Global queries on the datasets of Table 4 - Left Col.: AvgDeg, Right Col.: DegDistr

approach becomes slower as the query ranges grow larger due to the need of fetching and handling more data (vertex "versions") as more snapshots become involved in each query.

Overall MT can improve on ST in all queries apart from the vertex history retrieval by a factor up to 4X. In addition, the improvements over the baseline approach is of an order of magnitude, which provides evidence that the problem of efficiently designing vertex-centric storage models for NoSQL is not trivial and different design choices have a high impact on performance.

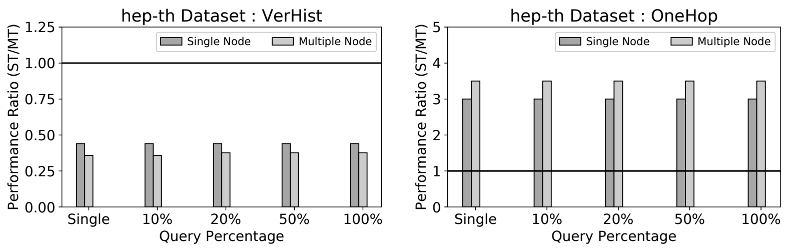
5.4 Results in a Multi-Node Setting

In this section, we study the relative performance of the two proposed models when employed in a Cassandra multi-node environment. More specifically, we measure the performance ratio of ST's execution time compared to MT's execution time[8] for varying query ranges. In a multi-node Cassandra approach each machine serves as an independent Cassandra node that is interconnected to other similar Cassandra nodes through a peer-to-peer architecture. To avoid any optimization issues that may arise from replication strategies as well as query node destination, each Cassandra node maintains a full replica of each dataset. The results for the local queries are presented in Figure 5 while those of the global queries are depicted in Figure 6.

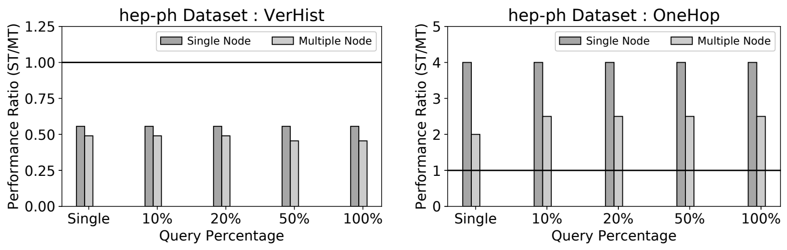
The main observations that can be derived are the following:

1. In the "Synthetic" dataset, each snapshot has relatively few changes with its predecessor/successor snapshot compared to the other three datasets which leads to less significant changes of the performance ratio over different query ranges.
2. In almost all cases of "VerHist" the two environment settings exhibit similar relative performance across all query ranges (with ST requiring approximately half the time of MT in most cases).
3. In the case of "OneHop" in the multi-node environment, MT becomes more efficient than ST as query ranges grow larger (by up to 3.5 times in the "hep-th" dataset).
4. With regard to "AvgDeg" in both single and multi-node environments, the relative difference of the two models tends to become smaller as query ranges grow larger (with the exception of the "Synthetic" dataset).
5. In the two larger datasets, "USPatents" and "Synthetic" the difference between MT and ST in the multi-node environment for "DegDistr" becomes more significant in the 50% and 100% query ranges (up to 2.9 times in the "USPatents" dataset).
6. Overall, MT still dominates in all queries apart from the vertex history, while improvements over the baseline are at the order of magnitude.

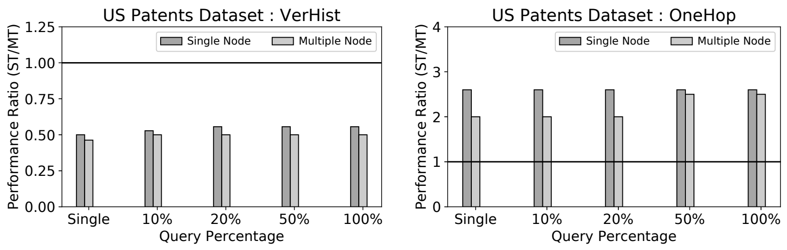
[8] e.g. a performance ratio of 2 corresponds to MT requiring half the execution time of ST



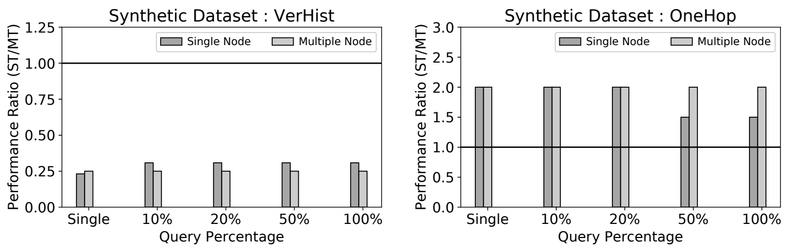
(a) Hep-th Dataset



(b) Hep-ph Dataset

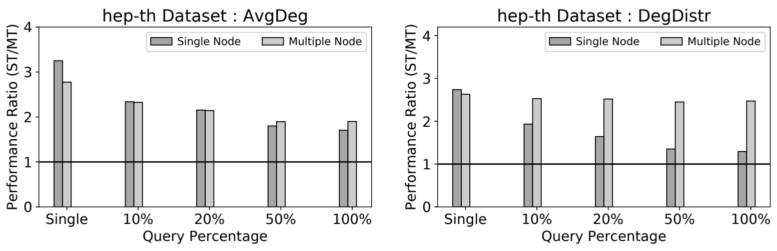


(c) US Patents Dataset

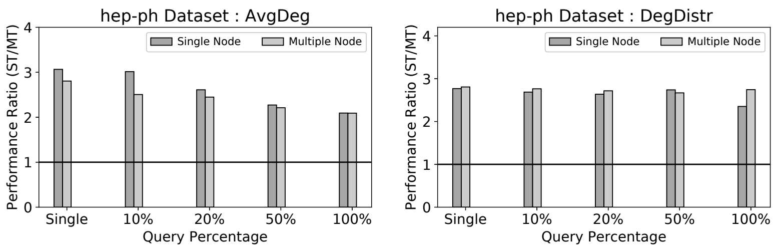


(d) Synthetic Dataset

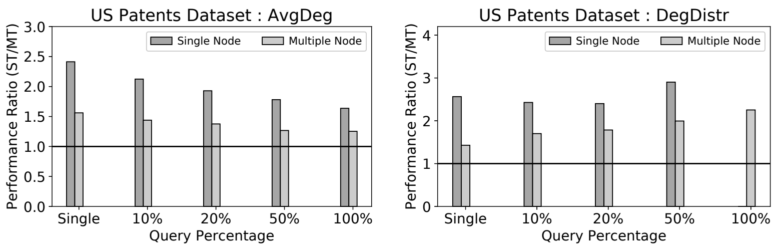
Fig. 5 Performance ratio on local queries (Table 4) - Left Col.: VerHist, Right Col.: OneHop



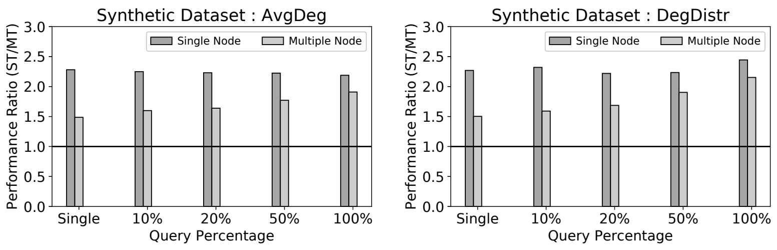
(a) Hep-th Dataset



(b) Hep-ph Dataset



(c) US Patents Dataset. DegDistr(ST-100%) did not run due to insufficient memory



(d) Synthetic Dataset

Fig. 6 Performance ratio on global queries (Table 4) - Left Col.: AvgDeg, Right Col.: DegDistr

Model \Query	AvgDeg	VerDistr	OneHop
MT Model	7.5 seconds	7.5 seconds	∼0.001 seconds
Neo4j	9 seconds	8.7 seconds	0.1 seconds

Table 5 Neo4j Time Cost Comparison

5.5 Results Compared to a State-of-the-Art Graph DBMS

A question may arise as to whether Cassandra is the most appropriate NoSQL system. To investigate this issue, we employ Neo4j[9], which is tailored to graph data management. More specifically, we modeled the SNAP "hep-th" dataset (Section 5.1) using Neo4j in two ways.

First, we explicitly store each of the sequence snapshots as new, single graphs, implicitly ignoring the commonalities between vertices and edges. This resulted in a prohibitive space cost since just the two last snapshots acquired space that was nearly equal to that of the ST model for the whole history (see Section 5.6 - Table 6).

Second, we implemented a vertex-centric representation of the sequence with each Neo4j graph node maintaining all of its relevant data by utilizing range intervals in a similar fashion to the ST model and the diachronic nodes of *HiNode-G**. This resulted in a comparable (albeit higher) space cost to that of ST and MT (Section 5.6 - Table 6): 61MBs which is twice as much space as the ST model and 33% more compared to MT. The main drawback however was slower query execution time for the queries and settings depicted in Figure 2; Table 5 summarizes the results.

Overall, the vertex-centric model we advocate is more efficiently implemented in state-of-the-art key-value-based NoSQL systems, such as Cassandra rather than using graph-oriented systems, which, albeit, are incapable of handling the graph evolution.

5.6 Concluding Remarks

Complementary to the previous experiments, we investigated the potential trade-offs between the two retrieval methods that were defined in Section 4.3 (i.e. *retrieve_all* and *retrieve_relevant*). Figure 7 shows a selection of indicative results in the MT and ST models for two of the datasets used in this section. In both datasets and models, *retrieve_relevant* was faster than *retrieve_all* for almost all query ranges with the relative performance of the two approaches becoming smaller as the query range got wider. The two models exhibit roughly similar times when handling query ranges of 100% sequence snapshots, with *retrieve_all* being slightly faster than *retrieve_relevant* in the MT model.

Furthermore, in the previous sections we focused on the relative time efficiency of the proposed models without taking into account the total space

[9] `https://neo4j.com/`

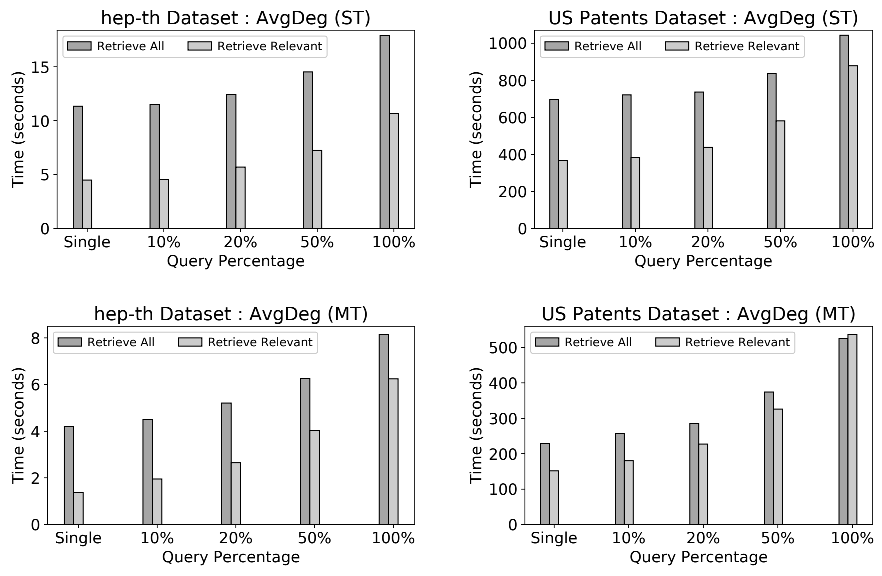


Fig. 7 Querying Modes Execution Times

	ST	MT	Base
Hep-th	**31.0 MB**	45.7 MB	86.1 MB
Hep-ph	**37.4 MB**	55.5 MB	104.8 MB
USPatents	**1.83 GB**	3.10 GB	14.7 GB
Synthetic	**929 MB**	2.28 GB	4.5 GB

Table 6 Space Utilization

required by each of them. Table 6 showcases the space required by each model for each of the datasets in Table 4.[10] In all cases the ST model (which more closely represents the original HiNode vertex-centric model of [9]) achieves better space utilization than its counterparts.

Our overall findings can be summarized as follows:

– The design approach to a vertex-centric system is not straight-forward. First off, ST outperforms MT in VerHist by up to 69% lower execution time with respect to the execution time of MT.
– Furthermore, ST has a lower space utilization than MT by up to a 59% decrease of MT's space.
– On the other hand, MT achieves faster execution time in OneHop, AvgDeg and DegDistr by up to a 75%, 54% and 63% lower execution times with respect to the execution time of ST (i.e., 4X, 2.17X, 2.7X speed-up), respectively.
– In all cases the baseline algorithm was slower than either one or both of our proposed models by up to an order of magnitude.

[10] Measured through the "nodetool" utility

- Typical state-of-the-art graph DBMS are not inherently suited for handling historical graph data. In this work, we provide concrete evidence that Cassandra outperforms Neo4j.
- The querying method *retrieve_all* offers comparable and sometimes better execution time than *retrieve_relevant* for 100% query range sizes but is slower for all other query ranges.
- The two proposed models tend to have similar performance for local queries when operating on a single and on a multi-node environment, while their relative difference for global queries tends to become smaller as query ranges grow larger.

6 Conclusions and Future Work

In this work, we tackled the design decisions and intricacies that arise when we follow a vertex-centric approach to handling historical graph data. Vertex-centric models are inherently more suited to handling local queries; thus, we adapted our previously suggested HiNode prototype system [9] and made use of NoSQL technology to propose and implement two alternative storage models. We performed an experimental evaluation of both models and demonstrated that the selection of a single design scheme for vertex-centric graph historical data is not a trivial task due to the potential trade-offs that occur for query execution times and space utilization.

An interesting future work direction would be optimizing the query execution procedures by employing intuitive data partitioning and replication strategies and making use of asynchronous calls to the underlying NoSQL DBMS at specific points during each query evaluation. Another fruitful area for further work would be the study of a user-parameterized hybrid design model, such as one that does not denormalize vertex and edge attributes, that could lead to improved results regarding query times and space utilization. Finally, the choice between different NoSQL approaches (e.g. Cassandra, MongoDB, HBase etc.) is not trivial since it could impact the overall system performance. Thus, a future work objective is to develop practical solutions that are geared towards exploiting the advantages of each particular NoSQL database system.

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: 23rd International World Wide Web Conference, WWW '14, pp. 237–248 (2014)
2. Apache Giraph. `http://giraph.apache.org/`
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. science **286**(5439), 509–512 (1999)
4. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: OSDI, vol. 14, pp. 599–613 (2014)

5. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: Conference on Scientific and Statistical Database Management, SSDBM '14, pp. 38:1–38:4 (2014)
6. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, pp. 997–1008 (2013)
7. Khurana, U., Deshpande, A.: Storing and analyzing historical graph data at scale. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, pp. 65–76 (2016)
8. Kosmatopoulos, A., Giannakopoulou, K., Papadopoulos, A.N., Tsichlas, K.: An overview of methods for handling evolving graph sequences. In: Algorithmic Aspects of Cloud Computing, pp. 181–192. Springer (2016)
9. Kosmatopoulos, A., Tsichlas, K., Gounaris, A., Sioutas, S., Pitoura, E.: 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`
10. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J., Han, W.: The g* graph database: efficiently managing large distributed dynamic graphs. Distributed and Parallel Databases **33**(4), 479–514 (2015)
11. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (2014)
12. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146. ACM (2010)
13. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. PVLDB **4**(11), 726–737 (2011)
14. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Computing Surveys (CSUR) **31**(2), 158–221 (1999)
15. Semertzidis, K., Pitoura, E.: Durable graph pattern queries on historical graphs. In: 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016, pp. 541–552 (2016)
16. Semertzidis, K., Pitoura, E., Lillis, K.: 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., pp. 121–132 (2015)
17. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 505–516 (2013)
18. Spillane, S.R., Birnbaum, J., Bokser, D., Kemp, D., Labouseur, A.G., Olsen, P.W., Vijayan, J., Hwang, J., Yoon, J.: A demonstration of the G_* graph database system. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013, pp. 1356–1359 (2013)
19. Yang, Y., Yu, J.X., Gao, H., Pei, J., Li, J.: Mining most frequently changing component in evolving graphs. World Wide Web **17**(3), 351–376 (2014)