TEMPO

Management and Processing of Temporal Networks

H.F.R.I. Project No. 03480

D1.2: Project Scientific/Technical Plan

Computer Engineering & Informatics Department
University of Patras
Greece
01/04/2023

Scientific/Technical Plan of TEMPO

Alejandros Spitalas and Kostas Tsichlas

April 01, 2023

Abstract

This document is the technical plan of the project TEMPO. It mainly focus on our plan related to the development of the temporal graph management system (database + query engine). To justify our choices and the partial divergence from the initial proposal we make an extensive survey of temporal graph management systems and try to capture their main characteristics. It is noteworthy, that most of these databases where developed after 2020, which makes necessary the changes to our initial plan. We also discuss our plan with respect to community and outlier detection.

Keywords: Temporal Graphs, Distributed Storage, Parallel Databases, Community Detection, Outlier Detection

1 Introduction

In recent years there is an increasing focus on time-evolving networks that produce a considerable amount of data. Networks, such as citation networks, traffic networks, and social networks are, naturally represented as graphs and they are usually dynamic. For example, in a citation network, new nodes and edges are constantly added due to the publication of new papers. An important challenge that arises in these time-evolving networks is the efficient management of their history in order to be able to reason about its whole evolution and not only about its latest state. This allows us to answer queries such as "What is the average connectivity of author X in the citation network between 2010 and 2015?".

There have been quite a lot of systems developed since 2016 for historical graph management. Many of them are distributed, since evolving graphs are extremely demanding with respect to space usage and query/update time. A rather outdated (2016) related survey can be found in [78]. They focus mainly on the models used for temporal graphs and the techniques available to query them. Another recent survey is [8] (2021) that analyzes graph streaming systems. In this survey the differences and similarities between graph streaming systems and historical graph systems are explicitly discussed. In general, graph streaming systems tend to use snapshots as the stable (latest) instance of the graph, since it may be the case that recent updates have not been registered. However, in principle, these snapshots may be stored and allow for historical queries as well, although in most of these systems this functionality is not considered but holds in principle. Some graph streaming systems explicitly - although it is controversial to what extent - support historical queries on such a rather small number of snapshots of their evolution.

The current technical plan has a twofold contribution. First, we provide a concise but comprehensive discussion on the systems developed from 2016 up to 2023 (before 2016, systems are covered by the survey in [78]). We do not discuss extensively these systems but state their key features. At the same time, we focus mainly on distributed systems making a simple reference to non-distributed ones. To the best of our knowledge, there is no other up-to-date comprehensive reference to such systems. Our second contribution, which required this state-of-the-art review, is the proposal of the high-level architecture of a distributed system for managing time-evolving graphs. The architecture is based on the ideas set by the authors in previous papers [37, 36, 69] as well as by the most recent developments in the area of historical graph management, as laid out in this concise but comprehensive survey.

The rest of the paper is structured as follows. In Section 2, we provide a review of historical graph management systems after 2016. In Section 3 we discuss the basic characteristics that the architecture of the system we intend to implement for managing historical graphs must have. In Sections 4-7 various architectures for the graph management and processing system are discussed while in Section 8 we briefly discuss the architecture that we decided for MAGMA based on the previous discussion. Finally, in Section 9 we discuss the analytics queries and in Section 10 we provide an updated timeplan based on the choices we have made.

2 A Survey on Existing Systems for Historical Graph Management

Historical graphs have to utilize multiple dimensions resulting in many possible directions for such a system. Most systems are concerned with the storage and query of the evolution of the attributes as time evolves and some try to utilize the evolution of the topology for better partitioning or for reasons related to efficiency. In Table 2, we provide, without further discussion, some basic characteristics of non-distributed historical graph management systems. Some terminology is in order to understand the following tables:

1. **transaction time vs valid time:** Transaction time represents the time that an event takes place (i.e., the moment that a node is inserted or deleted to/from a network), whereas valid time signifies the time period in which an object was valid (i.e., the time interval that a node existed in a database). In the transaction time setting updates can only occur in an append-like manner (i.e. an update in a field changes the value of the most recently stored value), whereas in the valid time setting updates can refer to any time point.

2. **time as a property vs snapshots:** In a rather crude manner, we get basically two different representations of time-evolving networks: a) snapshots, which correspond to a copy+log method; that is, the network is stored at specific time instances and in between a log is kept with the changes and b) time as a property, which corresponds to incorporating the notion of time as another special property of the objects/annotations within a network. There are many variations of these two basic representations.

3. **offline vs online vs streaming:** In an offline setting, we get all the history of the graph beforehand. In an online setting, the graph evolves and with it the database, while queries can be made at any time. In a streaming setting, we have an online setting with restrictions as to how much space and time is allowed for each update. In the literature, streaming is not usually related to historical information but more to computational restrictions on the processing of the stream due to its high velocity and massiveness. One can get as a by-product a rudimentary transaction time temporal graph processing system.

4. **time-dependent and time-independent algorithms:** If the algorithm on the temporal graph can be applied without time constraints then it is time-independent (e.g., pagerank computation at time instance t). If there are time constraints, then the algorithm is time-dependent (e.g., shortest path that respects time intervals on nodes/edges and the journey is time-consistent).

In Table 2 we show non-distributed systems without further discussing their characteristics. In Table 2 we show all distributed systems for historical graph management after the year 2016. Since our proposed system falls under this category we are going to discuss briefly some of these systems, which according to our opinion are quite important and have nice properties.

HINODE was the first pure vertex-centric system with respect to the storage model. It was introduced in [37] and supports valid time as well as extensions like multiple universes. It was implemented within the G^* system [40] by replacing its storage subsystem. They exhibited gains in space usage, which is an immediate consequence of the pure vertex-centric approach. They supported

Summarizing the Characteristics of Non-Distributed Temporal Graph Management Systems			
Systems	Memory	Storage Model	Time-related characteristics
InteractionGraph [22]	Main Memory (old graph in disk)	Custom	Transaction time
STVG [46]	Main Memory	Neo4j	valid time, offline, restricted to transit networks
ASPEN [18]	In-Memory/parallel	extends Ligra	Streaming
GraphOne [39]	in-memory NVMe SSD	Custom	Streaming, can't get arbitrary historic views if transaction time is assumed
Auxo [24]	Main and External Memory	Custom	Transaction time
[10]	Main Memory	Custom	Transaction time, Snapshot-based, focus on space savings
[4]	Main Memory	Neo4j	Valid time, In addition to entity evolution it supports schema evolution
TGraph [28]	Main and External Memory	Neo4j	Support ACID Transactions, slow topological updates but fast property updates, Transaction time
VersionTraveller [31]	Main Memory	based on PowerGraph static graph management system	Offline Snapshot-based, Focus on switching between snapshots
NVGraph [44]	Non-Volatile Main Memory and DRAM	Custom	Online Snapshot-based, Transaction time

Table 1: Non-distributed systems for historical graph management.

Summarizing the Characteristics of Distributed Temporal Graph Management Systems		
Systems	Storage Model	Time-related characteristics
Portal [51]	Spark	Offline, time as a property, Valid time
GDBAlive [47]	Cassandra	Transaction time
Graphsurge [62]	Custom	offline snapshots, focus on differential computation across multiple snapshots
TEGRA [30]	Custom	Transaction time, based on persistent trees, incremental computation model, window analytics
GraphTau [29]	Custom	Streaming
Immortalgraph [48]	Custom	Transaction time, Snapshot-based, Focus on locality-aware (w.r.t. time and topology by replication) batch scheduling for computation
HGS [35]	Cassandra	Transaction Time, Sophisticated Snapshot-based
SystemG-MV [72]	IBMs SystemG	Relaxed transaction time
Raphtory [70]	Custom + Cassandra for archiving	Transaction time, streaming
Chronograph [13]	MongoDB	offline, time as a property, Focus on graph traversals
Graphite [21]	Apache Giraph	offline, Time-dependent and time-independent algs
Granite [58]	Based on Graphite	focus on temporal path queries, partition techniques to keep everything in main memory
Tink [43]	Apache Flink	Online, Valid time
Gradoop - TPGM [61, 17, 60]	Apache HBase/ Accumulo	Valid and Transaction time (bitemporal), Fully-fledged system ranging from a graph analytical language to the storage model
Greycat [26]	NoSQL Database + custom	Valid time, No edge attributes
PAST [19]	based on key/value stores (e.g., Cassandra)	Streaming Spatio-temporal graphs, bipartite graphs, only edges with time-points, spatiotemporal-specific query workloads
HINODE [37, 36, 69]	Custom (other versions are based on Cassandra and MongoDB)	Online, time as a property, Valid time (allows more general notions of time), pure vertex-centric storage model

Table 2: Distributed systems for historical graph management.

local queries (e.g., 2-hop queries) as well as global queries (e.g., clustering coefficient). In addition, this vertex-centric model was also adapted for NoSQL databases by creating two models, SingleTable (ST) and MultipleTable (MT). In the former, all data fit in one table and a row has the data of a Diachronic Node, while in the latter, data are split in different tables - thus distancing from the pure vertex-centric approach. Two implementations were made, one in Cassandra [36] and later one in MongoDB [69] for comparison reasons. In MongoDB, the authors tried to take advantage of indices and iterative computation to reduce memory usage.

Portal In [51] they discuss about interval-based and point-based models preferring the interval-based model with sequenced semantics. As a data model, they use TGraph that uses the property graph model while they also discuss sequenced semantics in a distributed environment (e.g., partitioning, time-window operations). In PhD Thesis [50] they propose a Temporal Graph Algebra (TGA) and a temporal graph model (TGraph) supporting TGA. In addition, in [49] they propose a declarative language (Portal) based on the previous model and built on top of a distributed system (Apache Spark). Portal has SQL-like syntax following SQL:2011 standard. They also discuss possible algorithms on temporal graphs among which are node influence over time, graph centrality over time, communities over time, and spread of information. TGraph is a valid time model that extends the property graph model (each edge and vertex is associated with a period of validity), while all relations in Graph must meet 5 criteria: uniqueness of vertices/edges, referential integrity, coalesced, required property and constant edge association. TGA is both snapshot and extended snapshot reducible presenting a new primitive (resolve) while supporting operators like trim, map, and aggregation. Portal uses Spark for in-memory representation and processing while it uses Apache Parquet for on-disk data layout using node files and edge files (but it doesn't support an index mechanism). They experimented with different in-memory representations, SnapshotGraph(SG) that stores the graph as individual snapshots, MultiGraph(MG) that stores one single graph by storing one vertex for all periods and one edge for every time period and OneGraph that stores each edge and vertex only once (they also discuss combinations between these representations like MGC and OGC). It has distributed locality like Immortalgraph, experimenting with different partitioning methods (the equi-depth partitioning is more efficient in most experiments) but stores materialized node/edges instead of deltas and they also experimented with both structural and temporal locality, concluding that temporal locality is more efficient (among other reasons due to the lack of sufficient discrimination in the temporal ranges of the datasets).

ImmortalGraph [48] is a parallel in-memory storage and computation system for multicore machines and distributed settings designed for historical graphs. It focuses more on locality optimizations, both in saving the data and in the execution of the queries using locality-aware batch scheduling (LABS). They make a clear distinction and a very nice discussion between the time-centric layout and the structure-centric layout. It supports parallel temporal graph mining using iterative computations while they prefer those computations to be in memory. ImmortalGraph supports both global and local queries at a point in time or a time window. Data are stored in snapshot groups with the use either of edge files or vertex files, depending on the application. A snapshot group organizes together snapshots of a time interval by storing the first one and the changes that happened throughout the time interval. This can be stored either with the use of time locality by grouping activities associated with a vertex (and a vertex index) or with the use of structure locality by storing together neighboring vertex (and a time index). Instead of choosing between the possible trade-off from structure and time locality, they replicate the needed data and decide which technique to use according to the type of query and how far is the starting point from the start of the snapshot group. LABS favors partition-parallelism from snapshot-parallelism, so they prefer batch operations of vertex/edges achieving better locality and less inter-core communication. They also experimented with iterative graph mining and iterative computations. In the former, they reconstruct the needed snapshots in memory favoring time locality (and they compare both push, pull, and stream techniques), while in the latter they compute the first snapshot and the later $N-1$ snapshots in batch (achieving better locality). They also implemented both low-level and high-level query interfaces, the latter used for iterative computations. An earlier implementation of ImmortalGraph is Chronos [25] with the main difference being that it only focuses on time locality. Finally, they provide a low-level as well as a high-level programming interface (APIs) that in fact define their analytics engine. They experimented with algorithms like Pagerank, graph diameter, SSSP, connected components, maximal independent sets, and sparse-matrix vector multiplication.

Historical Graph Store (HGS) [35] is a cloud parallel node-centric distributed system for managing and analyzing historical graphs. HGS consists of two major components, Temporal Graph Index (TGI) that manages the storage of the graph in a distributed Cassandra environment, and Temporal Graph Analysis Framework (TAF) that is a spark-based library for analyzing the graph in a cluster environment. TGI combines Partitioned Eventlists, which stores atomic changes, with Derived Partitioned Snapshots, which is a tree structure where each parent is the intersection of children deltas (used for better structure locality storing neighborhoods). Both of them are partitioned, while they are also combined with Version Chain to maintain pointers to all references of nodes in chronological order. TGI divides the graph into time spans (like snapshot groups of ImmortalGraph) with micro-deltas which are stored as key-value pairs contiguously into horizontal partitions at every time span. In this way, it can execute in parallel every query to many Query Processors and aggregate the result to the Query Manager or to the client. It can work both on hash-based and locality-aware partitioning by projecting a time range (time-span) of the graph in a static graph. TAF supports both point in time queries and time-window queries; some of the supported queries are subgraph retrieval with filtering, aggregations, pattern matching, and queries about the evolution of the graph. An earlier implementation of TGI is DeltaGraph [34] that focuses on snapshot retrieval.

ChronoGraph [13] is a temporal property graph database built by extending Tinkerpop and its graph traversal language Gremlin so as to support temporal queries. It stores the temporal graph in persistent storage (MongoDB), and then loads the graph in-memory and traverses it. Their innovation is not in the storage model but in how they support traversal queries efficiently on top of it. It exploits parallelism, the temporal support of Tinkerpop to increase efficiency, and lazy evaluations to reduce memory footprints of traversals. Its main focus is on temporal graph traversals but can also return snapshots of the graph. They distinguish point-based events and period-based events because of their semantics and their architectural needs. They use aggregation to convert point-based events to period-based events so as not to have two different semantics in order to improve time efficiency in query execution. They achieve this by using a threshold as the maximum time interval that may exist between time points so as to group them together. A graph is composed of a static graph, a time-instance property graph, and a time-period property graph. They also use event logic, where an event might be either a vertex or an edge, on a period or a time instant. They exeprimented with temporal implementations of BFS, SSSP, and DFS, while they don't recommend DFS on their system because of Gremlin's recursive logic. One more thing they discuss is that when you store the temporal graph in snapshots, there will be some loss of information because a snapshot may contain data based on a specific time granularity (e.g., an hour) according to the needs of the problem, while when you store them using time intervals, one gets a more accurate representation of the graph. An extension of Chronograph by using time-centric computation for traversals is given in [11].

Tink [43] is an open-source parallel distributed temporal graph analytics library built on top of the Dataset API of Apache Flink and uses Gelly as a language. It extends the temporal property graph-model focusing on keeping intervals instead of time-points by saving nodes as tuples. It depends on Flink to use parallelism, optimizations, fault tolerance, and lazy-loading and supports iterative processing. It also uses functions from Flink like filtering, mapping, joining, and grouping. Most algorithms use Gelly's Signal/Collect (scatter-gather) model that executes computations in a vertex-centric way. It also provides temporal analytic metrics and algorithms. For the latter, they implemented shortest path earliest arrival time and shortest path fastest path while for temporal metrics they provide temporal betweenness and temporal closeness.

Gradoop (TPGM) TPGM [61] [17] [60] is an extension of Gradoop's EPGM model (model for static graph processing, presented in a series of papers from 2015, e.g., see [32]) to support temporal analytics on evolving property graphs (or collection of graphs) that can be used through Java API or with KNIME. Gradoop is an open-source parallel distributed dataflow framework that runs on shared-nothing clusters and uses GRALA as a declarative analytical language and TemporalGDL as a query language. Gradoop supports Apache HBase, and Apache Accumulo to provide storage capabilities on top of HDFS, while other databases can also be used with some extra work. TPGM supports bitemporal time by adding to vertices, edges and the graph as a whole, the logical attributes for start and end time for both valid and transaction time (some of these logical attributes may not be used). While TPGM provides an abstraction, Apache Flink is used for handling the execution process in a lazy way and it provides several libraries. GRALA provides operators both for single graphs and graph collections, it supports retrieval of snapshots, transformations of attributes or properties, subgraph extraction, difference of two snapshots, time-dependent graph grouping, temporal pattern matching, and others. For some more complex algorithms, it also supports iterative execution using the Apache Flink's Gelly library. Lastly, they have implemented a set of operations for their analytics engine and have implemented them in Flink - by using Flink Gelly. For further investigation, it should be mentioned that they provide an extensive description of their architecture while they also provide a *Lessons Learned* section that contains valuable information with respect to their design choices.

SystemG-MV In [72] they propose an OLTP-oriented distributed temporal property graph database (dynamically evolving temporal graphs). It is built on top of IBM's SystemG, which is a distributed graph database using LMDB (B-tree based key-value store). Data are stored in tables with key/value pairs allowing to query part of the graph efficiently without retrieving whole snapshots. Different tables exist for vertices, edges, and properties, while it supports updates only on present/future timestamps like transaction-time models. Therefore, changing previous values of the graph is not allowed explicitly, but it is possible to change past events by using low-level methods. In this model, they save two timestamps for the creation/deletion of vertices/edges but they don't allow edges to be recreated with the same id, although multiple edges can exist between a pair of vertices. For vertices, they keep the deleted vertices in a different table, while for properties they keep it simplified by keeping only one timestamp for the update as the rest can be calculated. Alongside the historic tables, they keep one table with the current state of the graph for more efficient queries.

GraphOne [39] is an in-memory data store with a durability guarantee on external non-volatile memory NVMe SSD, while it was solely implemented in C++. Its objective is to be able to perform both real-time analytics or diverse data access while synchronous updates are applied to the database. To achieve that, GraphOne uses a hybrid model which is composed of a circular edge log and an adjacency store. The adjacency store has a multi-versioned degree array and an adjacency list with chained edges, which is used to permanently store the data regarding to snapshots. On the other hand, the edge log is used to temporary store the incoming data as edges so as to later move them in parallel to the adjacency store and improve the ingestion time. In brief, an epoch in GraphOne consists of 4 stages: logging, archiving, durable, and compaction. At logging phases, records are inserted in the edge log at their arrival order, and when the inserted edges reach the archiving threshold the multi-threaded archiving phase starts in parallel with the logging phase. At the start of the archiving phase, it shards non-archived edges to multiple local buffers so as to keep the data ordering intact, and then the edges are being archived in parallel to the adjacency store, while also new degree nodes are allocated. In short, in the durable phase, data are being appended to a file, while in the compaction phase deleted data are being removed. We notice that despite the fact that GraphOne is designed to store evolving graphs, it is not designed to access arbitrary historical views from the adjacency store.

TEGRA [30] is a distributed system with a compact in-memory representation (using their own storage model) both for graph and intermediate state. Its main focus is on time window analytics for historical graphs, but it can also be used for live analytics as the data are ingested in the database. An interesting feature is the ICE computational model that takes advantage of the intermediate state of computations saving it, so as to use it in the same or different queries. Computations are being made only in subgraphs affected by updates at each iteration. This has some overhead on finding the correct state and also the extra entities that should be included in the query when there is large number of updates at each iteration or while trying to use ICE on different queries. Tegra also uses TimeLapse, an API for high-level abstraction which also allows what-if questions that change the graph creating different histories, suited for data analytics purposes. The storage model behind TEGRA is DGSI, which uses persistent data structures to maintain previous versions of data when modified. It uses persistent adaptive radix trees to store edges and nodes separately with path copying. It uses simple partitioning strategies to distribute the graph to nodes. Each node has two pART for nodes and edges respectively. Log files are being used to store updates between snapshots, which are stored in turn in the two pARTs. The branch and commit primitives are really interesting as well as the GAS (Gather - Apply - Scatter) model [23]. It allows also changing any version thus leading to a branched history (like a tree - full persistence). Lastly, TEGRA also uses an LRU policy to periodically remove versions that have not been accessed for a long time.

STVG [46] is a prototype framework that focuses on fast-evolving graphs. It is built on top of Neo4j and supports both point and time-window queries while its main use is to analyze evolutionary transit networks. It is based on the whole-graph model for representing the graph, which is composed of subgraphs that facilitate the conceptual modeling of the connectivity between entities and the time-graph of Neo4j that is responsible for keeping track of time evolution. Subgraphs are connected to the time-graph to keep track of the evolution of the whole-graph, while nodes belonging to different subgraphs are linked with complementary connectivity edges. Since this framework is used for evolutionary transit networks it is assummed that the the graph is always connected while edges can't recur over time. Projected graphs are used to materialize and retrieve the graph both at a time-window or a sliding window. They have implemented also graph metrics used to analyze a transit network, graph density, network diameter, and average path length having in mind their specific application. In general, this framework has some good ideas but it is tailored for transit networks.

Graphite [21] is a distributed system for managing historical graphs (fully evolved and using valid time) by using an interval-centric computing model (ICM) built over Apache Giraph. They assume data are given in ascending time order and any vertex can exist only once for a contiguous time-interval. It also has the ability to execute both time-independent and time-dependent historical queries (temporal queries on a time-window), while they tried to create a unifying abstraction that scales to both and at the same time simplifies algorithm design and detach user logic using ICM and time-warp operator. ICM uses Bulk Synchronous Parallel (BSP) execution for every active vertex of a query until it converges. They use two stages of logic, compute and scatter, where compute does the computations needed for a vertex, and scatter transfers it with messages to neighbor vertices as needed. Time-warp operator is applied at the alternating compute scatter steps to help sharing of calls and messages across intervals. A key aspect of it, is that it groups input guaranteeing correctness of grouping and no duplication, while it returns the minimum possible triples. They also designed and constructed a plethora of time independent (TI) and time dependent (TD) algorithms for their system with a very detailed experimental evaluation.

Granite [58] is a distributed engine for storing and analyzing temporal property graphs (supports temporal path queries) made on top of and as a sequel to Graphite focusing on path queries. Its basic assumption is that the updates are infrequent while queries are frequent. They extend the previous model by adding a temporal aggregation operator, indexing, query planning and optimization, while they prefer to relax ICM so as to work beyond time respecting algorithms. Granite handles both static temporal graphs and dynamic temporal graphs while it uses interval-centric features only in the latter case. An interesting point is that to optimize path queries they split them and execute them concurrently, while they also keep statistics about the active nodes at each time point so as to optimize the query planning. While Graphite makes hash partitioning at query execution, Granite first partitions every entity according to its type and then it performs a topological partition to its independent group of entities of the same type and splits them into workers using the round-robin technique. They also use a result tree so as not to send duplicate paths across the system (some parts of the path might be the same). Lastly, they propose a query language for path queries.

NVGRAPH This is a rather interesting system from a hardware perspective. NVGRAPH [44] is an in-memory data structure focused on exploiting the different advantages of NVMM and DRAM, combining them into a C++ library implementation. The major issue they try to tackle in NVMM is

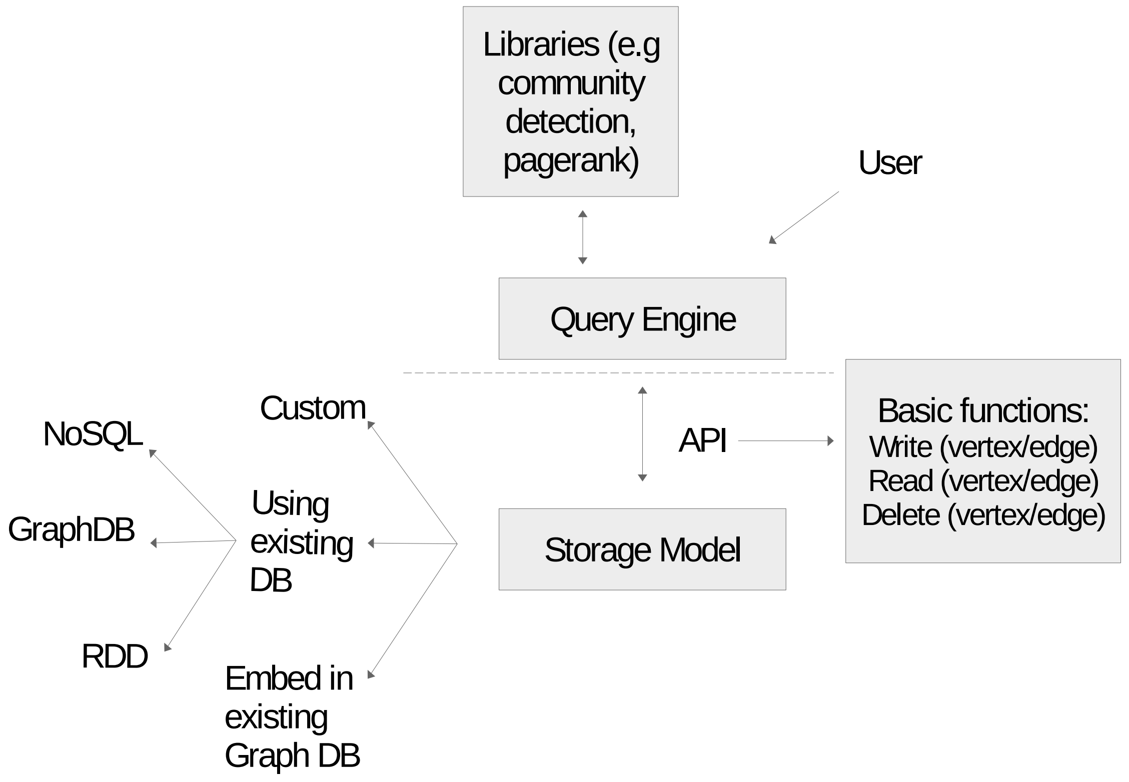


Figure 1: A view of MAGMA with the possible storage directions.

providing crash consistency while they argue that simply using NVMM without considering its issues is a sub-optimal solution. They focus on creating an architecture that uses both DRAM and NVMM to hide the issues of NVMM while they are exploiting its advantages. NVGraph stores the graph as a series of continuous snapshots by storing the first snapshot and deltas for the next snapshots. They also implemented 4 algorithms for evaluation: Pagerank, BFS, influence maximization, and rumor source detection.

3 Different Architectures for Historical Graph Management Systems

In this section, we describe the general characteristics of the proposed temporal graph management and processing system (MAGMA), the possible directions we could take implementing it as well as the possible obstacles we need to overcome.

An immediate observation from the previous systems is that each one of them focuses on different aspects of historical graph management, resulting in a different appropriate solution for each application. This is because the management and processing of historical graphs span multiple design dimensions forbidding the existence of one system to rule them all. Our approach is towards creating a purely vertex-centric and storage optimal (asymptotically) distributed system called MAGMA with the ability to update/query efficiently the history and apply graph algorithms on arbitrary time periods rather than on specified snapshots. Following HiNode, MAGMA will be more efficient in local than global queries due to its vertex-centric structure. However, we also wish to efficiently execute global queries (e.g., pagerank) by exploiting our vertex-centric architecture and implementing modern techniques (e.g., thinking like a vertex) for efficient and effective parallel computation. Another important aspect that needs to be addressed in a later stage of the development of MAGMA, is the system's API. In particular, we need to design the system in a way that guarantees its simplicity with respect to use, its efficiency, its scalability, its flexibility with respect to its functionality, and its compatibility with existing libraries (for static or temporal graphs).

The key part of the system is the efficient and effective vertex-centric storage of the graph. From a design perspective, a diachronic node contains the whole history of a particular node in the sense that it stores all changes and their time intervals related to this node, such as a change in an incoming edge or a change in a property of the node. To this end, we employ three fundamental operations in order to update and query the diachronic nodes: write, read and delete. All three operations are applied on diachronic nodes that contain all relevant information (edges, properties, etc.). More complex updates and query operations can be built on these fundamental operations that will serve mainly the online management and processing of the historical graph.

Regarding the storage model, we have narrowed our options into either creating a custom database for storing the historical graph into servers or by extending an existing database and applying our model to them. In any case, we will always stick to the pure vertex-centric approach proposed in HiNode and adapt it appropriately to fit the design choice of the storage model. In the case of creating a custom database, we have complete freedom with respect to designing the storage model to fit HiNode, but on the other hand, it will require considerably more effort for the implementation as well as to ensure compatibility with existing libraries. On the other hand, one could use an existing database, either a NoSQL database like Cassandra and MongoDB or a Graph database (e.g., GraphX or SystemG). In this case, it is easier to build the system and take advantage of the optimizations and functionality that already exist within this database (e.g., fault-tolerance, partitioning and concurrency), but there is less freedom in applying the storage model of HiNode. Another option, in this case, is to extend an existing graph database (e.g., GraphX) to support natively the management and processing of historical graphs based on a pure vertex-centric approach. This is a harder task, but it has the merit of sharing and using existing libraries within this particular graph database. In addition, the visibility of such a solution will be much higher across the community.

Since MAGMA is a distributed system, the partitioning strategy is of paramount importance for the efficiency of the system. Most systems use either a simple hash-based partition or a chronological or topological partitioning. In our case, the topological partitioning is more natural but we also need to take into account the temporal evolution of the graph. In topological partitioning, we want to place in the same machine, nodes that are connected or that are relatively close to each other. One problem we might encounter with topological partitioning is that in different timestamps, the distance between nodes changes, and as a result, different partitions may be more appropriate in different time instances. This is problematic in our case since a diachronic node contains all the history of the node and thus naturally all history is stored in a single machine. Two possible solutions for this issue are either by using different metrics for partitioning combining the whole history of the graph or by dividing parts of a node to different machines. Another possible solution, which could also be combined with the previous one, is the duplication of some nodes across machines. However, in this case, care should be taken with respect to space usage and synchronization.

Another critical part of the system is the query engine and the libraries that will be available. Regarding the libraries, we intend to implement algorithms on temporal graphs like temporal shortest path (journeys) and community detection and evolution while also supporting algorithms for static snapshots. This can be achieved either by using the abstraction provided from the API or by exploiting the system's architecture and creating them from scratch. For the former task, we first want to create a query engine able to handle more demanding tasks that supports parallelism and provides the user with an easy-to-use API. To do so, our processing unit needs to apply one of the following approaches: "thinking like an edge" (TLAE), "thinking like a vertex" (TLEV), "thinking like a neighborhood" (TLAN), "thinking like a subgraph" (TLAS) or "thinking like an interval" (TLAI). We need to further investigate these approaches and decide which one would be more efficient in our system, although we can deduce straightforwardly that some of these will probably not fit our vertex-centric architecture. On the other hand, TLEV techniques seem as the most promising at the moment, in order to take advantage of Hinode's vertex-centric structure, while TLAN or TLAS approaches could also fit our model depending on the partition strategy used. At a later stage, these approaches will be used for iterative computations.

3.1 Different Approaches for MAGMA

Our goal is to develop a storage model and a query engine for storing, managing and processing historical graphs in a distributed setting. The requirements for MAGMA are the following:

1. A node-centric approach for the storage of the historical graph that will also support efficiently constant updating.

2. The system must be developed in a distributed system.

3. Support of OLTP queries, which are simple queries that typically involve a few records. The emphasis is on fast processing, because OLTP databases are read, written, and updated frequently. If a transaction fails, built-in system logic ensures data integrity.

4. Support of OLAP queries, which are complex queries that usually refer to a large part of the graph. The emphasis here is on efficiency tackling the queries without worrying or even prohibiting updates of the graph.

In Sections 4-6 we use existing Database(s) as building block(s). The idea is to use one or more databases and combine them accordingly so as to create the MAGMA system. In this case, the databases need to have the appropriate data structures for the efficient implementation of the system and at the same time need to provide the appropriate functionality for implementing operations (e.g., ACID transactions for OLTP). In Section 7, we propose the use of an existing graph database in order to design and implement MAGMA, either by using time as an additional but special-purpose feature or by changing the graph database in order to natively support historical queries and updates.

An initial and rudimentary set of operations that we wish to support is the following:

1. **(OLTP) Insert/Delete Node/Edge:** The system must be able to cope with updates of the historical graph anywhere in the graph and anytime.

2. **(OLTP) Query the Neighborhood of a Node:** Given a node v the goal is to return the neighborhood of v at a specific time instance, an interval, or a combination of intervals (one may assume propositional logic).

3. **(OLAP) Basic graph-wide query operations like Diameter and Pagerank:** In this case we need to support queries that involve a large part of the graph at specific time instances, time intervals or combinations of time intervals.

4. **(OLAP) Community Detection:** Find a partition at specific time instances or intervals or under other time restrictions like the ones used in time-related shortest path queries (if applicable).

5. **(OLAP) Anomaly Detection:** Find whether a node has a behavior (the semantics of "behavior" are defined with respect to the evolution of communuties) that is different to a large set of nodes (possibly its community).

4 OLTP-focused with Secondary OLAP Functionality

In this case our distributed system natively supports OLTP queries and additional mechanisms are required to support OLAP queries. In this framework, distributed ACID-compliant transactions should be supported in an efficient manner. Until recently, the first choice for OLTP database was an SQL RDBMS-like database. However, more recently, some NoSQL databases started supporting ACID transactions although issues remain with respect to the scalability of those transactions. To solve this problem, NewSQL databases were created. Thus, in this case, we have to either choose an RDBMS database that is also distributed or a NewSQL database.

4.1 RDBMS-like

RDBMS-like databases are more suitable for OLTP queries but on the other hand most of them are not distributed. It could be an SQL table database or something similar to MariaDB, or PostgreSQL-BSD that also adopts a multi-master approach, but in any case, it should be a distributed database and be able to handle OLTP transactions. As for the OLAP queries, we should be able either to implement them in the primary database (which is not efficient or straightforward, as most OLTP databases do not provide such functionalities) or when a complex query arrives, we should create an instance in a secondary database that can handle OLAP queries efficiently. This seems to be inefficient as well, since creating such an instance requires the extraction of a large portion of data, which is not suitable for RDBM databases.

In any case, for the RDBMS database we can use the SingleTable or the MultipleTable model created in HiNode [36, 69], respecting our node-centric approach for the storage model.

Known Implementations In [57] they benchmark different storage systems (including PostgreSQL), having on top a Spark system for querying, while they also test three different schemes. Their experimental evaluation shows that PostgreSQL doesn't have the best performance when OLAP queries are performed, compared to other NoSQL systems.

Applying this Strategy to Historical Graphs In this case, we have many choices between RDBMS-databases like PostgreSQL or MariaDB (some are distributed others not, but the final system is distributed). The implementation of the OLAP queries can be carried out in two ways: either try to implement them in the RDBMS database (which is not only inefficient for the OLAP query but also halts the OLTP transactions and we need a way to save them in a buffer) or extract the required instance from the RDBMS and apply the OLAP query on this instance. The latter approach is called Extract-Transform-Load (ETL).

Advantages - Disadvantages In the following we state succinctly the pros (+) and cons (−) of this approach.

+ Good support of ACID
− Many design choices can affect the performance of the system.
− Partitioning of the historical graph may be quite restricted.
− SQL is not suitable for volume, velocity, and variety data, rendering it highly inefficient for a cloud-based application
− Inefficient usage of space (NoSQL or custom, in general can have better storage efficiency alongside with the fact that if we want multiple labels, some nodes may not have them, thus in RDBMS some columns may be empty)
− Normal RDBMS scale vertically

4.2 Use NewSQL DB

Newsql can be defined as a class of modern relational DBMSs that seek to provide the same scalable performance of NoSQL for OLTP workloads and simultaneously guarantee ACID compliance for transactions as in RDBMS. In other words, these systems want to achieve the scalability of NoSQL without having to discard the relational model with SQL and transaction support of the legacy DBMS.

Having said that, NewSQL databases follows all specification requirements needed for our database. So we could either use one as a storage or ”extend” one so as to implement HiNode model for historical graphs and at a later stage, find efficient ways to execute analytical queries directly on them or with ETL techniques.

We should be cautious as it is relatively new and most reviews are in their favor. Examples of such databases are CouchBase, CockroachDB, RocksDB, VoltDB and NuoDB. Many NewSQL databases are built on top of RocksDB key-value store.

An Existing Implementation [52] (which is also based on a master-master architecture) is based on CouChDB, which is a NewSQL database. The main problem that they tackle is the partition of the graph in a distributed system so as not to do many I/Os in different partitions. The proposed solution to this problem is based on aggressive and lazy replication. Additionally, in [20] they created mechanisms to answer SPARQL queries on top of Couchbase.

Applicability to Historical Graphs NewSQL databases are key-value databases or document distributed store databases that also support ACID transactions. As a result, by adopting them for our system we get ACID transactions and also analytical queries on top of them. For the analytical queries, there exist also ways to implement SPARQL queries on top of Couchbase. The only problem is that if an analytical query is a relatively large one, it is better to extract it and execute it in a different system as it may halt the execution of transactions for a long period of time. This means that the system will automatically try to understand when an ETL approach is needed based on the massiveness of the query.

Advantages - Disadvantages: In the following we state succinctly the pros (+) and cons (−) of this approach.

+ Easy architecture
+ The implementation of the system can commence pretty fast
+ Horizontal and Vertical scalability
+ Can be implemented without the use of many different components (e.g., different databases, different buffers)
− NewSQL is relatively new and not as well studied as other solutions.
− In case OLTP transactions are postponed during OLAP executions, the transaction log may become pretty large while starvation may be an issue in case of many successive and parallel OLAP ueries.

5 OLAP-focused with Added OLTP Functionality

In this case, MAGMA will be mainly based on an existing OLAP-oriented database, extending it to support OLTP queries. There are two basic approaches towards attaining this combination that are based on replication schemes (both of them support synchronous and asynchronous variations):

1. **Master–Slave:** In this case, the data modification operations are processed only on the master node, and the updates are synchronously or asynchronously propagated to slave nodes. The data may be read both from the master node (which always contains the latest data version) and slave nodes that may contain outdated data if the replication is performed asynchronously.
2. **Master–Master or multimaster:** In this case, all the nodes can process update operations and propagate these updates to other nodes. In this method, it is difficult to implement synchronous replication, and delays due to network communications can be significant. If the updates are performed asynchronously, there is another difficulty related to the fact that conflicting data versions may occur, which require techniques for detecting and resolving conflicts (automatically or on the application level).

As previously mentioned, the system architecture will be suited for OLAP queries, supporting OLTP queries, either by default or with modifications. In the following, we will discuss in more detail these two options.

5.1 Master-Slave

In this case we have a system structure similar to the implementation of HiNode [36]. There is a master node where client sends the queries, and many distributed slave nodes that execute it. Two basic implications when supporting OLTP queries come up: a) how to guarantee ACID (or some relaxation) for OLTP queries and b) how to execute OLAP queries without stopping OLTP transactions. We can overcome the first problem by using a distributed database that also has some support for ACID queries. However, the latter problem requires either a way to store the OLTP transactions while the database is busy doing the analytical query, or by instantiating the database (e.g., copy the related to the OLAP query to a different container) and execute the OLAP query on it while allowing OLTP transactions. In this case, the second approach is not so much appropriate, as the DB is more OLAP oriented and mitigating the data to do the OLTP transactions in an OLAP oriented DB is a waste of resources, and thus we focus on the first approach.

An Existing Implementation In [33] they show best practices for implementing transactions in MongoDB and the results of TPC-C benchmark with small changes so as to adapt to best practices. They describe the transactions allowed in MongoDB. MongoDB can also benefit from denormalized schema and reducing roundtrips to DB, even though they are not allowed in the TPC-C benchmark. From the above paper we can observe the scalability of the system as well as the benefits from MongoDBs indices and the denormalized schema. On the other hand, MongoDB doesn't fully implement ACID transactions and probably also need much more storage for replication. [1]

Applicability to Historical Graphs In this case, we have two major choices for the database: MongoDB or Yugabyte[2]. Both databases are open-source, with the difference that YugaByte DB supports distributed ACID while MongoDB supports single-shard ACID. An early MongoDB implementation of HiNode [69] can be used as a basis for setting an API to support ACID transactions and allow for OLAP queries simultaneously with OLTP transactions. As previously mentioned, this can be implemented by postponing the execution of transactions until the OLAP query is completed or by instantiating part of the database in a different container so that the OLAP query is unaffected by future transactions.

Advantages - Disadvantages: In the following we state succinctly the pros (+) and cons (−) of this approach.

+ Easy architecture
+ The implementation of the system can commence pretty fast
+ Good support since this approach is based on known databases
− Reduced efficiency of OLTP transactions
− In case OLTP transactions are postponed during OLAP executions, the transaction log may become pretty large while starvation may be an issue in case of many successive and parallel OLAP queries.
− Hard to support OLTP transactions and OLAP queries at the same time
− Distributed transactions have in general poor performance.
− Partitioning of the historical graph among the slaves will be quite restricted

In general, this approach is not so flexible since many design choices have been already determined by the underlying database.

5.2 Master-Master

There are multiple variations of multimaster but their common characteristic is that at least two nodes/machines process transactions (there are at least two masters). A variation corresponds to active/passive mode where the master nodes switch between these modes and only the active can process a transaction. Another variation is the active/active mode where two or more master nodes are active simultaneously and can process transactions at the same time. The multimaster scheme supports easier OLTP transactions on a distributed NoSQL environment[3].

By using this architecture, we can either use a multimaster database that supports OLTP transactions and find a way to co-execute OLAP with OLTP and by applying the HiNode data structure, or we could have a custom made multi-master scheme in the front for the support of ACID transactions and on the background any distributed database that fits better our proposal. In the second case the basic problem is that we have to make a system with support and proof of ACID transactions, which is a more complex application, so in this section we will only focus on using a multi-master database. The basic problem of this architecture, similarly to 5.1, is how to support OLTP transactions while a time exhausting OLAP query is being executed.

[1]Chronograph [13] uses MongoDB for the system, although they don't seem to support ACID transactions.

[2]YugaByte is Redis and Cassandra compatible. We could use YugaByte with Redis, but further investigation of compatibility and support operations is required.

[3]Multi-Master can support distributed ACID `https://medium.com/yugabyte/6-signs-you-misunderstand-acid-transactions-in-distributed-databases-43dcaba24485`.

Implementation with this Architecture In [52] they built a framework on top of CouchDB focusing on the management of large dynamic graphs. It is mostly focused on using a hybrid approach with partitioning techniques while they also use clustering so as to reduce the needed replication. At brief, they first decided on doing a clustering of the nodes in different machines so as to keep close the data that interconnect and reduce the I/Os of a query, and then they apply a lazy replication scheme based on the access patterns of each node, using histograms and prediction algorithms. This approach helps minimize the communication cost and have a balanced system in terms of CPU usage. Although this system is based on dynamic graphs the replication cost may be too much for historical graphs as the size grows considerably fast. It is also harder to find a generally good clustering technique for historical graphs. Lastly, CouchDB supports the change of the partition on top level.

Magma [41] is a replacement for couchstore and they use data structures that can be utilized in our system (like LSM trees), adopting an asynchronous distributed multi-master architecture. However, they don't explicitly mention anything about transactions while it is an append-only structure and thus may not be appropriate for general historical graphs. It is focused at improving efficiency on write throughput as the system needs to rewrite blocks of data because of its append-only nature. Although this architecture can be utilized in our case, it needs many modifications and additional research. [4]

Looking at Historical Graphs Depending on the partition strategy we decide to adopt, we could take advantage of a multi master DB. For example, in a chronological partition where we have prior knowledge of which "master" holds which data, we could just propagate the query to the appropriate node like the postgres-BDR model. It just needs that every master has access to a dictionary with the chronological partitioning scheme, otherwise we could use a router that has knowledge of the partitioning scheme to propagate the data to the appropriate machine like [52]. By adopting this scheme, OLAP queries could also be supported in a BSP or "thinking like a vertex" way, managing good balance in the networks utilization and minimizing I/Os. In addition, with the cooperation of a good partition over the data (for particular types of queries like queries on snapshots in a specific time instance) it would drastically reduce the I/Os by adopting a "thinking like a vertex" strategy. The basic problem of this approach is the partition and the replication scheme applied so as to have good utilization of machines and minimization of the cost of queries and transactions, which becomes even harder in the case of historical graphs. In this approach, we follow a chronological partition strategy, which may be straightforward for the DB architecture but is not appropriate or straightforward for our HiNode model. Our model is more compatible with a topological partitioning approach, meaning that each fat node is fully stored in a machine potentially using shadow nodes and replication instead of being stored in pieces across many machines.

Still, there is the problem of supporting OLTP transactions at the same time as OLAP queries are being executed. Like 5.1, this can be tackled with a log buffer to store OLTP queries and at a later stage insert them as a batch into the system. In this architecture, we could also apply preprocessing on the data to know the appropriate machine where each data should be sent and minimize the I/Os that would happen in the normal insert mode.

Advantages - Disadvantages: In the following we state succinctly the pros (+) and cons (−) of this approach.

+ More flexible in choosing the distributed database.
+ More flexible in data structure for the in-memory database part, as we can create it from scratch.
+ More flexible to architectural optimizations
+ Probably more appropriate for ACID over distributed databases (compared to master-slave 5.1).
+ Probably better for implementing vertex-centric queries (thinking like a vertex)

[4]Cassandra also supports master-master and AID (not C).

– Needs much work on implementation.

– If we do something from scratch we need proof of ACID and a lot of research.

– All masters should either hold the entire snapshot or depending on the partition we may need a lot of replication of data, so there will be storage issues.

6 Hybrid

In this case, the system is composed from different storage components that are interconnected and create an all in one system. One idea is that the OLTP part could follow the streaming databases saving a log to send to the OLAP part, but keeping at any time the current state (in transaction time) of the graph for fast read. Another idea is that the OLTP part can be modified based on the application, e.g., we could have an application for IoT sensors where the OLTP part receives a lot of data, makes statistical analysis keeping some knowledge as the stream comes, but also instead of storing all the data, removes a part because sensors are storing too much information with noise (filtering). In general, the idea of a hybrid system is to have two fully functional system components that will interact in some way. There are many hybrid options but we are going to discuss only a few.

6.1 OLTP buffer + Store and OLAP System

We consider two possibilities. One possibility is to have as a first level a fast in-memory storage system like Redis (also ACID) to make a cache or buffer store for the data before sending them to a persistent storage like Cassandra, MongoDB or HBase. This way, we can have a threshold for batch or lazy insert the data when they are needed, while the OLTP transactions will be much more efficient than previous suggestions. The choice of the threshold constitutes a trade-off w.r.t. transaction and analytical query efficiency. The basic idea is that there are two interconnected systems with two-way communication (not only one way) allowing for more efficient choices for the OLTP part (e.g., using an event log or use compression techniques that have been studied before). One system to handle the OLTP transactions and one system to handle the OLAP queries, which sometimes will need to fetch data to be updated from the OLTP part. While we use batch updates to the OLAP part, we can use that time to make operations on the data and save time later. Incremental partitioning techniques can be employed while moving data from the OLTP system to the OLAP system and query operations can be also performed on them during this movement.

A different possibility is to have two mostly independent systems that each one is better at different tasks and an abstraction layer on top that will redirect the query to the appropriate database. In this way, we achieve better performance on the cost of replication. For example, this has been done before (e.g., immortalgraph) where they kept one system with topological partitioning and one with chronological and pass the query to the best at handling it. In our case, one of the two systems should also handle the OLTP transactions and thus the OLAP part will probably be outdated for some time before passing along the new updates.

One more advantage of hybrid systems, is that the one system can be deployed in a different place than the second one, so for IoT applications, we could have many mini OLTP systems and a big central general authority for the Distributed OLAP part.

Implementation with this Architecture Sprouter [1] is a dynamic graph processing system over data streams at scale. It uses Cassandra for OLTP and does OLAP by bulk loading data to Spark using micro-batches efficiently in bulks. This idea has been used to systems like GraphOne.

A hybrid database approach using graph and relational Database can be found in [73] that basically follows the second possibility using two heterogeneous databases. They argue that the key advantage of graph databases is performance:

> "In contrast to relational databases, where the query performance on data relations decreases as the dataset grows, the performance of graph databases remains relatively constant ... Hybrid database approach or multi database system is defined as integrated data system composed of collections of two or more autonomous datasets and/or databases. There are multiple issues that must be addressed like developer needs to learn multiple technologies, multiple query languages and many more."

Implementation Tailored to Historical Graphs Adopting the first possibility, we could use an in-memory database for the OLTP part, either Redis or another in-memory data storage and create either a buffer or in general a temporary storage of data, so as to achieve fast updates. Periodically, we move all data in batch to the persistent storage. For the persistent storage, we could try something like HBase or RocksDB or we could use another persistent Database that will handle all OLAP and most read operations (most recent events will appear only in the buffer part).

A slightly different version to the above is that instead of using an event buffer in the OLTP part, we could construct a data structure that could handle efficiently the in-memory part. Maybe a lightweight or compressed HiNode as we don't need all of its abilities. It could even be just a sketch of the graph so as to answer queries about the structure.

For the second possibility, we need two fully functional databases, that both store the whole graph (or part of the graph if we decide that it is possible) and an abstraction layer that will direct appropriately the queries. In general, for this to work we need to create a new data structure that can handle efficiently the parts that HiNode can't.

Advantages - Disadvantages In the following we state succinctly the pros (+) and cons (−) of this approach.

+ Much more flexible, compared to previous approaches.
+ Can become much more efficient, as we don't prioritize one of the two systems, although this could have some trade-offs.
+ Can more easily be transformed and changed according to the characteristics of the workload.
+ Creates the possibility of later improving or replacing a component without changing the rest.
+ After creating the first prototype, we can create multiple combinations for different ideas.
− Technically more challenging as it uses more technologies.
− Increase storage if replication is used.
− More time for construction as more times is needed to create a hybrid system.

6.2 Discussing Different Threshold Choices

What we need to discuss for the first system above, is that the use of different threshold techniques and variations in memory storage, can change the whole structure of the database. This is because, as the threshold gets larger, data are moving more sparsely from one storage to the other and as a result, we have the ability to delay a little the process of batch insert so as to make a better partitioning (sometimes incremental partitioning is much harder, especially when you have to move previously inserted values). On the other hand, if we have a smaller threshold, the batch insertion will be much faster and won't delay the transactions. In addition, when an analytical query comes, it will probably have to deal only with data inserted to the persistent storage or with a small portion of the data in the buffer. However, we need a better incremental partitioning and a faster connection between the two stores to rapidly send the data to the persistent storage.

7 Graph Databases

There is a very interesting taxonomy on graph databases in [9] which also explains the different design choices in the creation of the graph databases, from the storage architecture to the OLTP/OLAP[5] and ACID compliance. Using the taxonomy (which does not include historical graph databases) our system is closer to a wide-column store or a document store, but in either way we need indices on the attributes, otherwise we need to implement a native graph store based on the LPG model. The hybrid model is like the data hubs from the taxonomy.

7.1 Using an Existing Graph Database

It is possible to use an existing graph database, but it needs to be one that supports multiple labels both on edges and on nodes, while allowing the creation of indices over these labels. Other than that, it would help to use existing libraries of the system.

Implementation with this Architecture There are some historical graph systems that were implemented using Neo4j. One example is STVG [46], although all systems that use Neo4j have been labelled as non-distributed systems[6].

Implementation in Historical Graphs We can use many different databases like Neo4j, JanusGraph or ArangoDB (the last one uses a document store, so it could be the counterpart of MongoDB). Our requirements for these databases include being distributed, OLTP compliant (we can compromise on ACID compliance) and OLAP compliant, or we should at least be able to create OLAP compliance on top of it.

Most probably, our main choices are either wide-column stores, document stores or native graph stores. On wide-column stores, the main problem is in indexing and the way attributes are stored in the columns, as we need to comply with HiNode. On document stores, we can easily implement the HiNode model in the same way it was implemented in MongoDB but we need the store to support indices and OLTP transactions as not all of them do. Additionally, in the previous two categories, we should check the variants of LPG model they use (e.g., ArangoDB only allows one label per vertex). Lastly, native graph stores have the advantage of using the LPG model; the problem lies in their implementation schemes since we require our implementation on top of them to be as close as possible to the HiNode model.

Advantages - Disadvantages .

+ Many libraries can be reused, thus we can have faster and more robust implementation.
− More research time is required on every category and DB.
− Its not the best idea to have on top of a store a graph DB and on top of that a historical graph DB compared to implementing the last on top of a store (probably this is why this approach has been adopted only a handful of times).
− More difficult to support all our needs as they already have done some compromises.
− Functionality already implemented for graphs is not useful for historical graph management, thus creating unnecessary overhead.

7.2 Extend an Existing Graph Database

An interesting idea is that historical graphs can be seen as an "extension" of graphs and thus we could take an open-source graph database and extend it so as to support historical graphs using the HiNode model. Such a system could be easier adopted from the community (as they are already using the current version of the graph DB), although

[5]Note that OLTP for graphDB has a slightly different meaning.
[6]Recently, Neo4j can be deployed as a distributed system although with certain restrictions.

we have to follow the community rules and guidelines to do so. It is a rather interesting idea as we could also at times extract snapshots of the historical graph that could be directly processed or analyzed using the pre-existing functions of the current system.

Implementation with this Architecture The only known pre-existing work on this idea is Gradoop - TPGM [61, 17, 60] where they took the previous implementation of Gradoop EPGM and change it so us to support temporal graphs. Both Gradoop EPGM and Gradoop TPGM have been developed by the same team.

TinkerGraph is quite close to this idea as well.

Application to Historical Graphs In general, there are many design choices as to extending a graph database including the choice of the database as well as how it will be extended and which of the initial functionality will be transferred to the historical graph system. In the following, we discuss possible graph databases for this scheme with some comments.

Neo4j ← is one of the most famous open-source graph databases that is ACID compliant and also uses the LPG model.

Redisgraph ← is developed by RedisLabs from scratch on top of Redis, with the help of Redis Modules API with extended commands and capabilities. It stores data in RAM to achieve memory efficiency as well as fast indexing and querying.

Titan ← Distributed on multi-machine clusters. Titan supports ACID and eventual consistency. For backend, it supports Apache Cassandra, Oracle BerkeleyDB, Apache HBase. Titan also supports native integration with TinkerPop. The project was stoped in 2015 but Janusgraph was forked from it.

Other possible choices are: ArangoGraph, GraphDB (also knowledge graph and RDF), memgraph and nebula.

Advantages - Disadvantages .

+ Existing libraries and functionality can be used, thus we can have faster and more robust implementation.
+ We can use many functions and customize them to fit our needs.
+ We can create a general API for both applications.
− More research time on every category and DB respectively.
− More work to understand previously created code and implement something that can be accepted by the community (after it exceeds the prototype version).

7.3 Create a new Graph Database

Like the previous described databases, instead of using an existing graph database, we could take a distributed data storage (it is actually a database, but with less functionalities, meaning less overhead) and create on top of it our historical graph database. This is the closest we could go to create from scratch a database and needs much research and working time/experience.

Many NewSQL databases are built on top of RocksDB key-value store. We could also create them in a later stage, if we create a good API to connect individual components and at a later stage replace them with custom built components specified for historical graphs. Other choices to be used as stores include Redis, HBase and Accumudo.

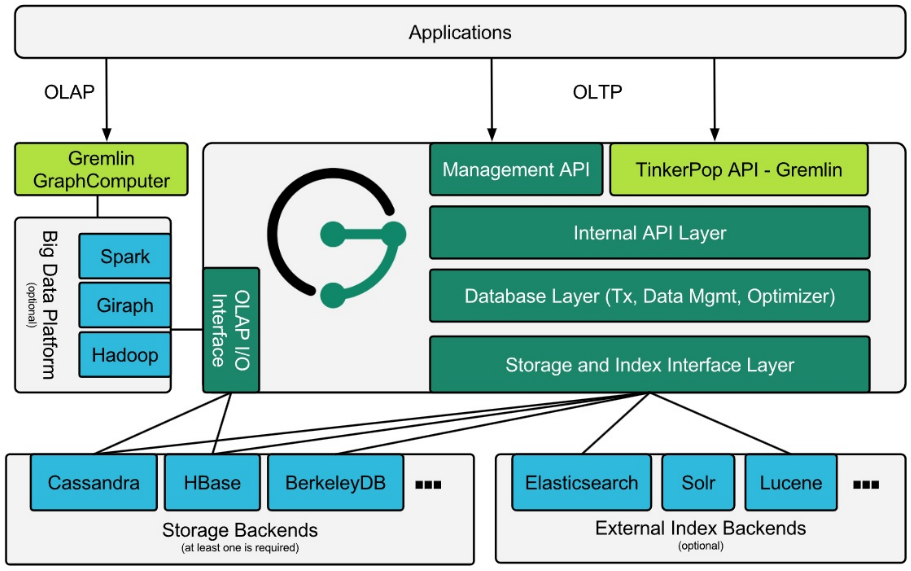

Applications
OLAP
OLTP
Gremlin GraphComputer
Management API
TinkerPop API - Gremlin
Internal API Layer
Database Layer (Tx, Data Mgmt, Optimizer)
Storage and Index Interface Layer
Big Data Platform (optional)
Spark
Giraph
Hadoop
OLAP I/O Interface
Cassandra
HBase
BerkeleyDB
Storage Backends (at least one is required)
Elasticsearch
Solr
Lucene
External Index Backends (optional)

Figure 2: Janusgraph architecture.

8 The MAGMA Approach

Based on the previous extensive discussion, we decided not to start from scratch building a historical graph database. Although such a choice would allow us great freedom in the design of the system, it would be impossible to implement something more than a pilot system with basic functionality (e.g., no concurrency). This is why we chose to use another approach. The most promising approach, to our opinion, is to extend an existing graph database. We have already discussed the pros and cons of such an approach, but what really made us choose this approach was that such a solution would have high visibility among the users of such graph databases. Among the many graph databases we looked at, two graph databases were the most appropriate: ArangoDB and JanusGraph. We finally chose to use Janusgraph, mainly because ArangoDB require a license in order to use its full functionality.

JanusGraph is an open source, distributed graph database under The Linux Foundation, which is highly scalable, optimized for storing and querying large graphs with billions of vertices and edges distributed across a multi-machine cluster. JanusGraph is a transactional database that can support thousands of concurrent users, complex traversals, and analytic graph queries. The storage layer is pluggable, meaning we can choose between different backeneds so as to store our data or even create a connection for a store more appropriate for our application. Having in mind all these advantages of Janusgraph, especially the adaptability to our needs and the full access to its source code, we decided it was the best possible approach so as to extend an already existing database and support also historical graphs with our own architecture. Additionally, JanusGrpah supports Tinkerpop and allows the use of Gremlin as a query language for both OLTP and OLAP. The architecture of Janusgraph is depicted in Figure 2.

9 Queries on MAGMA

Based on their scope [9] we consider the following types of queries.

1. **Local queries** that involve the retrieval or update of a single edge, single node or a property, subject to time restrictions.
2. **Neighborhood queries** that involve k-hop queries (for small k) around a node or an edge subject to time restrictions.

3. **Traversals** that explore part of a graph, like shortest paths, subject to time restrictions.
4. **Global graph analytics** that involve the whole graph, like community detection, subject to time restrictions.

These query types can be used to define various types of workloads that usually try to capture the queries applied in specific scenarios.

1. **Interactive workloads** that consist of short read-only queries (local and neighborhood queries), of complex read-only queries (mainly traversals) and of transactional update queries (local update queries like insert and delete).
2. **Business intelligence workloads**; these are read-only queries that span relatively large parts of the graph (subject to time restrictions) and usually make heavy use of summarization and aggregation operations (e.g., count the number of nodes with degree larger that k).
3. **Graph analytic workloads** that usually refer to read-only traversal and global graph analytics queries (e.g., community detection).
4. **Mixed workloads** that can be any combination of the above workloads.

All types of queries will be applied to MAGMA, although we will focus on a small subset of them. In particular, for local queries we will construct transactional workloads in order to test the efficiency of MAGMA in such scenarios. Small k-hop queries as well as traversal queries will also be tested, focusing for the latter case on time-restricted shortest paths (journeys). Finally, global graph analytics will mainly concern global community detection algorithms that are based on the "think like a vertex" paradigm.

Apart from such queries, we will also focus on local community detection that can be seen as an extended neighborhood query as well as on outlier detection. The former, will be based on the notion of the anchor introduced in [7] and initially will be tackled in a streaming environment before moving to a distributed setting compatible with the MAGMA architecture. The latter will be seen more as a by-product of the former, since one way of defining outliers in graphs is by looking as to how nodes differ with respect to the community structure. Both community and outlier detection will be designed in a distributed model exploiting the characteristics of MAGMA.

10 The Scientific/Technical Plan

The research required to finalize this document was concluded by the end of the 1st year of the project TEMPO (end of April 2023). The main reason, as shown in this document, was the extensive literature research required to choose the most appropriate architecture and models for storing and processing historical graphs. Based on the previous discussion our time-plan starting from the second semester of 2023 is as follows.

07-12/23 During this semester the following is expected to be accomplished:

MAGMA The research team will delve into details concerning the functionality of Janusgraph and its basic components: Tinkerpop as the main framework to express OLTP and OLAP queries, Janusgraph as the graph database for static graphs and its connection to the storage model (e.g., Cassandra).

Queries Design and experimentation on a streaming algorithm for community detection and outlier detection.

01-06/24 During this semester the following is expected to be accomplished:

MAGMA Use Gremlin to express time restrictions on simple queries related to historical graphs and identify the necessary changes that need to be made to Gremlin so that time intervals can be queried (we could follow the example of Chronograph [13]). Identify the changes that need to be made so that Janusgraph can support time not as another property but as a first-class citizen. Finally, identify the best storage model among the ones already supported and familiarize with the libraries needed to connect a new storage backend that is not currently supported.

Queries Design and analyze a distributed community detection algorithm based on the ”think like a vertex” paradigm and tailored to the vertex-centric approach of HiNode. Develop transactional historical graph workloads based on the well-known LDBC benchmark [5].

07-12/24 During this semester the following is expected to be accomplished:

MAGMA Change Gremlin appropriately so that it can express efficiently OLTP and OLAP queries with time restrictions. Make changes to Janusgraph so that time is inherent in the system and not as a simple property. If the storage backend required is not supported by Janusgraph, then develop the appropriate connector (e.g., a very promising storage backend is RocksDB). If needed, lay the grounds for developing a custom storage backend based on HiNode with minimal functionality.

Queries Complete the distributed community detection algorithm based on the ”think like a vertex” paradigm. Develop various workloads based on the LDBC Benchmark [5] and identify real historical networks that can be used to test MAGMA. To generate historical graph workloads of various types, two prossible apporaches can be taken. Either to post-process the workloads generated by LDBC to get historical graph workloads or extend the LDBC generator to have this functionality. Extend the distributed community detection algorithm to identify outliers.

01-06/25 During this semester the following is expected to be accomplished:

MAGMA Polishing of the MAGMA system and extensive experimentation and evaluation. Report on possible future extensions for the system.

Queries Implementation and experimentation over specific traversal (e.g., journeys) and graph analytics queries (e.g., pagerank) on MAGMA. Generation of different types of workloads for experimenting with MAGMA. Implementation and experimentain of distributed community detection algorithms and outlier detection (to a lesser extent) on MAGMA.

References

[1] Tariq Abughofa and Farhana Zulkernine. Sprouter: Dynamic graph processing over data streams at scale. In *International Conference on Database and Expert Systems Applications*, pages 321–328. Springer, 2018.

[2] H L Advaith, S Adarsh, G Akshay, P Sreeram, and G P Sajeev. A proximity based community detection in temporal graphs. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6, 2020.

[3] A. Aghasadeghi, V.Z. Moffitt, S. Schelter, and J. Stoyanovich. Zooming out on an evolving graph. *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT).*

[4] Landy Andriamampianina, Franck Ravat, Jiefu Song, and Nathalie Vallès-Parlangeau. A generic modelling to capture the temporal evolution in graphs. In *16e journées EDA : Business Intelligence & Big Data (EDA 2020)*, volume RNTI-B-16, pages 19–32, Lyon, France, August 2020.

[5] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The ldbc social network benchmark, 2023.

[6] Marcelo Arenas, Pedro Bahamondes, and Julia Stoyanovich. Temporal regular path queries: Syntax, semantics, and complexity. *CoRR*, abs/2107.01241, 2021.

[7] Georgia Baltsou, Konstantinos Tsichlas, and Athena Vakali. Local community detection with hints. *Appl. Intell.*, 52(9):9599–9620, 2022.

[8] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems, 2021.

[9] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Comput. Surv.*, jun 2023. Just Accepted.

[10] Kyoungsoo Bok, Gihoon Kim, Jongtae Lim, and Jaesoo Yoo. Historical graph management in dynamic environments. *Electronics*, 9(6), 2020.

[11] Jaewook Byun. Enabling time-centric computation for efficient temporal graph traversals from multiple sources. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[12] Jaewook Byun and Daeyoung Kim. Object traceability graph: Applying temporal graph traversals for efficient object traceability. *Expert Systems with Applications*, 150:113287, 2020.

[13] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2020.

[14] Diego Caro, M Andrea Rodríguez, Nieves R Brisaboa, and Antonio Fariña. Compressed kd-tree for temporal graphs. *Knowledge and Information Systems*, 2015.

[15] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proc. VLDB Endow.*, 14(10):1845–1858, June 2021.

[16] Xiaoying Chen, Chong Zhang, Bin Ge, and Weidong Xiao. Temporal query processing in social network. *Journal of Intelligent Information Systems*, 49(2):147–166, 2017.

[17] Lukas Christ, Kevin Gomez, Erhard Rahm, and Eric Peukert. Distributed graph pattern matching on evolving graphs. 2020.

[18] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 918–934, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Mengsu Ding, Muqiao Yang, and Shimin Chen. Storing and querying large-scale spatio-temporal graphs with high-throughput edge insertions. *arXiv preprint arXiv:1904.09610*, 2019.

[20] Hakim el Massari, Sajida Mhammedi, Noreddine Gherabi, and Mohammed Nasri. *Virtual OBDA Mechanism Ontop for Answering SPARQL Queries Over Couchbase*, pages 193–205. 01 2022.

[21] Swapnil Gandhi and Yogesh Simmhan. An interval-centric model for distributed computing over temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1129–1140, 2020.

[22] B. Gedik and R. Bordawekar. Disk-based management of interaction graphs. *IEEE Transactions on Knowledge & Data Engineering*, 26(11):2689–2702, nov 2014.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI'12, page 17–30. USENIX Association, 2012.

[24] Wentao Han, Kaiwei Li, Shimin Chen, and Wenguang Chen. Auxo: a temporal graph management system. *Big Data Mining and Analytics*, 2(1):58–71, 2019.

[25] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. 07 2017.

[27] Jelle Hellings and Yuqing Wu. Stab-Forests: Dynamic Data Structures for Efficient Temporal Query Processing. In Emilio Muñoz-Velasco, Ana Ozaki, and Martin Theobald, editors, *27th International Symposium on Temporal Representation and Reasoning (TIME 2020)*, volume 178 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[28] Haixing Huang, Jinghe Song, Xuelian Lin, Shuai Ma, and Jinpeng Huai. Tgraph: A temporal graph data management system. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, page 2469–2472, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the fourth international workshop on graph data management experiences and systems*, pages 1–6, 2016.

[30] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: Efficient ad-hoc analytics on evolving graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 337–355. USENIX Association, April 2021.

[31] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *2016 USENIX Annual Technical Conference (USENIX-ATC 16)*, pages 523–536, 2016.

[32] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, NDA '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Asya Kamsky. Adapting tpc-c benchmark to measure performance of multi-document transactions in mongodb. *Proc. VLDB Endow.*, 12(12):2254–2262, aug 2019.

[34] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 997–1008, 2013.

[35] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 65–76. OpenProceedings.org, 2016.

[36] Andreas Kosmatopoulos, Anastasios Gounaris, and Kostas Tsichlas. Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data. *Computing*, 101(12):1885–1908, 2019.

[37] Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura. Hinode: an asymptotically space-optimal storage model for historical queries on graphs. *Distributed Parallel Databases*, 35(3-4):249–285, 2017.

[38] Sudhindra Gopal Krishna, Sridhar Radhakrishnan, Michael Nelson, Amlan Chatterjee, and Chandra Shekaran. On compressing time-evolving networks.

[39] Pradeep Kumar and H. Howie Huang. G¡span class=”smallcaps smallercapital”¿raph¡/span¿o¡span class=”smallcaps smallercapital”¿ne¡/span¿: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4), January 2020.

[40] 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 Parallel Databases*, 33(4):479–514, 2015.

[41] Sarath Lakshman, Apaar Gupta, Rohan Suri, Scott Lashley, John Liang, Srinath Duvuru, and Ravi Mayuram. Magma: A high data density storage engine used in couchbase. *Proc. VLDB Endow.*, 15(12):3496–3508, aug 2022.

[42] Mo Li, Junchang Xin, Zhiqiong Wang, and Huilin Liu. Accelerating minimum temporal paths query based on dynamic programming. In *International Conference on Advanced Data Mining and Applications*, pages 48–62. Springer, 2019.

[43] Wouter Lightenberg, Yulong Pei, George Fletcher, and Mykola Pechenizkiy. Tink: A temporal graph analytics library for apache flink. In *Companion Proceedings of the The Web Conference 2018*, pages 71–72, 2018.

[44] Soklong Lim, Tyler Coy, Zaixin Lu, Bin Ren, and Xuechen Zhang. Nvgraph: Enforcing crash consistency of evolving network analytics in nvmm systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1255–1269, 2020.

[45] Ziyang Liu, Chong Wang, and Yi Chen. Keyword search on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 29(8):1667–1680, 2017.

[46] Ikechukwu Maduako, Monica Wachowicz, and Trevor Hanson. Stvg: an evolutionary graph framework for analyzing fast-evolving networks. *Journal of Big Data*, 6(1):1–24, 2019.

[47] Maria Massri, Philippe Raipin Parvedy, and Pierre Meye. Gdbalive: a temporal graph database built on top of a columnar data store. *Journal of Advances in Information Technology*, 12, 09 2020.

[48] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Trans. Storage*, 11(3), jul 2015.

[49] Vera Moffitt and Julia Stoyanovich. Portal: A query language for evolving graphs. 02 2016.

[50] Vera Zaychik Moffitt. *Framework for Querying and Analysis of Evolving Graphs*. PhD thesis, 07 2017.

[51] Vera Zaychik Moffitt and Julia Stoyanovich. Towards sequenced semantics for evolving graphs. In *EDBT*, pages 446–449, 2017.

[52] 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, page 145–156, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Mohammad Hossein Namaki, Yinghui Wu, Qi Song, Peng Lin, and Tingjian Ge. Discovering graph temporal association rules. New York, NY, USA, 2017. Association for Computing Machinery.

[54] Michael Nelson, Sridhar Radhakrishnan, and Chandra N. Sekharan. Queryable compression on time-evolving social networks with streaming. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 146–151, 2018.

[55] Michael Nelson, Sridhar Radhakrishnan, and Chandra N. Sekharan. Algorithms on compressed time-evolving graphs. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 227–232, 2019.

[56] Peng Ni, Masatoshi Hanai, Wen Jun Tan, and Wentong Cai. Efficient closeness centrality computation in time-evolving graphs. ASONAM '19, page 378–385, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Mohamed Ragab, Riccardo Tommasini, and Sherif Sakr. Benchmarking spark-sql under alliterative rdf relational storage backends. In *QuWeDa@ISWC*, 2019.

[58] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. Granite: A distributed engine for scalable path queries over temporal property graphs. *Journal of Parallel and Distributed Computing*, 151:94–111, 2021.

[59] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, Reynold Cheng, and David W. Cheung. Efficient processing of shortest path queries in evolving graph sequences. *Information Systems*, 70:18–31, 2017. Advances in databases and Information Systems.

[60] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. Distributed temporal graph analytics with gradoop. *The VLDB Journal*, May 2021.

[61] Christopher Rost, Andreas Thor, and Erhard Rahm. Analyzing temporal graphs with gradoop. *Datenbank-Spektrum*, 19(3):199–208, 2019.

[62] Siddhartha Sahu and Semih Salihoglu. Graphsurge: Graph analytics on view collections using differential computation. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1518–1530, 2021.

[63] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. Incremental execution of temporal graph queries over runtime models with history and its applications. *Software and Systems Modeling*, pages 1–41, 2021.

[64] Konstantinos Semertzidis. *Storage, Processing and Analysis of Large Evolving Graphs*. PhD thesis, 07 2018.

[65] Konstantinos Semertzidis and Evaggelia Pitoura. Durable graph pattern queries on historical graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 541–552, 2016.

[66] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2019.

[67] Konstantinos Semertzidis and Evaggelia Pitoura. A hybrid approach to temporal pattern matching. In *2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 384–388, 2020.

[68] Shalini Sharma and Jerry Chou. A survey of computation techniques on time evolving graphs. *International Journal of Big Data Intelligence*, 7(1):1–14, 2020.

[69] Alexandros Spitalas, Anastasios Gounaris, Kostas Tsichlas, and Andreas Kosmatopoulos. Investigation of database models for evolving graphs. In Carlo Combi,

Johann Eder, and Mark Reynolds, editors, *28th International Symposium on Temporal Representation and Reasoning, TIME 2021, September 27-29, 2021, Klagenfurt, Austria*, volume 206 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[70] Benjamin Steer, Felix Cuadrado, and Richard Clegg. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems*, 102:453–464, 2020.

[71] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proceedings of the VLDB Endowment*, 10(8):877–888, 2017.

[72] Warut D. Vijitbenjaronk, Jinho Lee, Toyotaro Suzumura, and Gabriel Tanase. Scalable time-versioning support for property graph databases. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1580–1589, 2017.

[73] H.R. Vyawahare, P.P. Karde, and V.M. Thakare. A hybrid database approach using graph and relational database. In *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, pages 1–4, 2018.

[74] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, Dec 2019.

[75] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Efficiently answering span-reachability queries in large temporal graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1153–1164, 2020.

[76] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156, 2016.

[77] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. On querying historical k-cores. *Proceedings of the VLDB Endowment*, 14(11):2033–2045, 2021.

[78] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7(2):573–582, 2016.

[79] Jingwen Zhao, Yunjun Gao, Gang Chen, and Rui Chen. Towards efficient framework for time-aware spatial keyword queries on road networks. *ACM Trans. Inf. Syst.*, 36(3), November 2017.

A Appendix

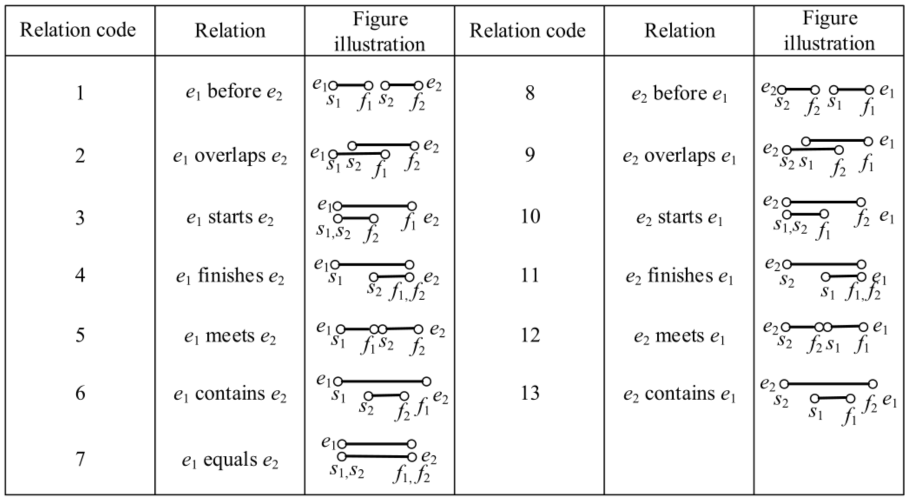
We provide some additional references to the literature that is more distantly related to historical graph management systems.

A survey of computational techniques on time evolving graphs and the DASH framework: In [68] they distinguish the Time Evolving Graphs (TEG) from their aspects on Graph analytics, graph algorithms and the graph framework. For graph analytics, they focus on graph mining techniques and especially centrality, community detection, link prediction and rare category detection, targeting mainly at incremental techniques. Regarding graph algorithms, they focus on where to compute (which portion of the graph has been updated) and how to compute (which techniques will be used). Regarding graph frameworks, they distinguish from the platform used to store the data (e.g., single node, distributed), the programming models used, the communication model (synchronous/asynchronous) and the data management strategy (e.g., partitioning strategy). Also they state the importance of graph repartitioning as the graph evolves.

DASH: is a distributed, asynchronous, heterogeneous and dynamic event driven framework that they propose to manage time evolving graphs. It takes events as input (e.g., addition/removal of edges), and instead of storing snapshots it reconstructs the graph while computing a query (loading only necessary data to decrease memory usage). This creates an overhead at query execution but it overlaps with the computation time while doing incremental execution. They also dynamically break jobs into tasks to be computed independently. The framework is divided in smaller components to deal with different system requirements independently. Their query engine follows the publish/subscribe model (pull-based) and it also supports work stealing between workers.

InTempo: INcremental execution of TEMPOral graph queries over runtime models with history and its applications In [63], they introduce a language that supports temporal queries, a querying scheme for historical runtime models, temporal logic in queries and an implementation based on the Eclipse Modeling Framework for two usecases. The model receives point events, like the creation/deletion of nodes. InTempo constructs a GDN for each temporal graph query, it executes it over the RTM^H and prunes RTM^H over elements that will not be involved in future query executions. It also supports postponing a decision if the query can be satisfied in the future. It supports self-adaption on behavior or structure. The first usecase is for a Smart Healthcare System (SHS) where they apply the temporal requirements of medical guidelines. The second usecase is based on a social network.

Time-Dependent Graphs: Definitions, Applications, and Algorithms In [74] they discuss the literature on evolving graphs. Quoting: "Many problems, which can be solved in a linear time or a polynomial time on static graphs, become NP-complete or NP-hard problems on time-dependent graphs, such as the connected component problem.". Two types of time-dependent graphs are discerned: time instance/interval. As for models, the consider discrete and continuous time dependent graphs. The former can be with labels for edges or snapshots or by transforming the graph into a static one by building copies of vertices. The latter can be represented with either an index function, called presence function, or they can be treated as flow-dependent graphs. They mention three types of shortest path problems: earliest arrival path problem, latest departure path problem and shortest duration path problem. They also discuss strong and weak connectivity and spanning trees in time-dependent graph. They also discuss single-criteria/ multi-criteria route planning, time-dependent social analysis and time-dependent subgraph problems (matching, mining). They distinguish two main directions in systems, snapshot-based and traversal-based, arguing that the former are not suitable



Relation code	Relation	Figure illustration	Relation code	Relation	Figure illustration
1	e_1 before e_2	e_1 s_1 f_1 s_2 f_2 e_2	8	e_2 before e_1	e_2 s_2 f_2 s_1 f_1 e_1
2	e_1 overlaps e_2	e_1 s_1 s_2 f_1 f_2 e_2	9	e_2 overlaps e_1	e_2 s_2 s_1 f_2 f_1 e_1
3	e_1 starts e_2	e_1 s_1,s_2 f_2 f_1 e_2	10	e_2 starts e_1	e_2 s_1,s_2 f_1 f_2 e_1
4	e_1 finishes e_2	e_1 s_1 s_2 f_1,f_2 e_2	11	e_2 finishes e_1	e_2 s_2 s_1 f_1,f_2 e_1
5	e_1 meets e_2	e_1 s_1 f_1 s_2 f_2 e_2	12	e_2 meets e_1	e_2 s_2 f_2 s_1 f_1 e_1
6	e_1 contains e_2	e_1 s_1 s_2 f_2 f_1 e_2	13	e_2 contains e_1	e_2 s_2 s_1 f_1 f_2 e_1
7	e_1 equals e_2	e_1 s_1,s_2 f_1,f_2 e_2			

Figure 3: Presentation of ALLEN's 13 time-dependent relations.[74].

for some temporal queries like traversal-based queries. Finally, they list some open issues for time-dependent graphs, such as the need for generative models, metrics for time-dependent networks and dynamical systems for time-dependent networks (instead of snapshot-based).

GTAR: temporal graph association rules In [53] they propose graph temporal association rules (GTAR) so as to discover complex events and patterns using graph mining algorithms. The temporal graph model investigated maintains timestamps instead of intervals, but association rules are being discovered over a time window. They also define a discovery graph mining algorithm (DisGTAR) to find association rules implemented in Java.

OTG In [12], they create an Object Traceability Graph (OTG) for the object traceability problem by using temporal graph traversals. They use the EPCIS standard for object traceability and apply it to Chronograph (a persistent version that uses MongoDB) so as to take advantage of temporal graph traversals and not execute expensive recursive queries. To accomplish this, they create a conversion model for the EPICS system by creating mapping rules from the EPICS document to temporal graphs elements.

TRPQ language In [6] they propose a general-purpose query language for interval-timestamped temporal property graphs focused on temporal regular path queries by extending the MATCH operator. It is implemented in Rust by using the Itertools library. Their models are being evaluated over two principles, snapshot reducibility and extended snapshot reducibility. They consider temporal and structural navigation, while queries are being evaluated in polynomial time. Temporal navigation (for next or previous timestamps) can be considered with the PREV and NEXT operators, while structural navigation (for moving from node to node over edges) is accomplished with the BWD and FWD operators.

Temporal Social Network In [16] they focus on storing indexing and querying temporal social networks. They implemented the algorithms on Java and experimented with KONECT dataset. They propose a storage model and two index structures, TUR-tree and TUA-tree, so as to efficiently answer three kind of temporal queries that they investigate: Friends of Interesting Activities (FIA), Users of Time Filter (UTF) and Group of Users with Relationship Duration (GURD). They have two types of nodes, users and activities (to denote events) as well as relationships between users or user and events associated with intervals or timestamps when a user participate in an activity. Users and activities are stored separately, and users are clustered into pages on disk. For efficient querying, they propose two index structures, one for Temporal Users and Relationships (TUR-tree) and one for Temporal Users and Activities (TUA-tree). The former needs to deal with time intervals that are associated both with users and relationships proposing the MVB-tree. The latter uses a B^+-tree and Bloom Filters, since activities are timestamped.

Zooming Out an Evolving Graph In [3] they discuss zoom operations on temporal graphs, modifying the resolution based on the analysis needs. They propose two operators, attribute-based zoom (aZoom) and temporal window-based zoom (wZoom) implemented with dataflow operations to experiment over Portal's TGraph models (different implementation for each model). The former modifies structural resolution (e.g., from nodes to neighborhoods) by computing new nodes for every occurrence of a certain given pattern. The latter, modifies the temporal resolution (e.g., hours vs days).

Compressing and querying compressed evolving graphs In [38] they propose a way to compress evolving graphs providing the ability to query them without decompression. Due to the nature of evolving graphs, data may not fit in main memory for analysis, which justifies compression. They exploit row-by-row compression for node data separately using CSR and CBT data structures. The graph is represented based on the neighbors of each node over time. Time is a stream of 0 and 1, indicating changes to the neighborhood of a node from the previous timestamp. They experiment in combining CSR, CBT and the bitpacking method. In [55], they focused on querying the compressed graph without decompressing it. More specifically, they focused on earliest arrival path query and time-evolving transitive closure, the former solved both on a point, incrementally or in a time-window. One disadvantage of all the previous algorithms, is that the compression is differential and the algorithms must be executed incrementally from time 0 independently of the query start time. In addition, in [54] they focus on querable compression on time-evolving social networks with streaming (meaning they can add/remove nodes/edges without decompression) based on previous techniques.

In [14], they proposed the compressed *kd*-tree ($ck^d - tree$) to store temporal graphs where all operations can be performed using orthogonal range search. They also discuss different data structures for temporal graphs like EdgeLog, EveLog or TGCSA and they compare them. They divide the temporal graphs to point-contact or interval graphs and they also divide them based on the extent of their dynamics to incremental graphs, decremental and partially dynamic.

A.1 Solutions for Particular Temporal Problems

In [65] - (analyzed in more detail in his PhD thesis [64]) they discuss pattern matching in node-labeled temporal graphs. They created and implemented an algorithm that compactly finds patterns that exist continuous or collectively across different snapshots. In [67] they propose an approach to detect patterns that use an edge-based representation of temporal graph while they use filtering based both on time and structure. In [66] they create algorithms for top-k graph pattern matching on temporal graph using the previous algorithm.

In [59] they discuss how to store and query evolving graphs with respect to shortest path queries. In [71] they present SAMS; an approach for automatic algorithm transformation so as to execute the same algorithm concurrently in multiple snapshots. They also provide an implementation in C++ by storing the graph in CSR format.

In [77] they propose an index-based solution (PHC-index) to find historical k-cores in a temporal graph. In [75], the same authors define the span-reachability model for temporal graphs, where in contrast to time-respecting paths models, they do not require paths to follow a non-decreasing order but to be in a small time-interval close to each other. They also propose ways to identify relationships between entities and implemented algorithms to answer the reachability problem in C++.

In [27] they create internal memory temporal join algorithms focused on temporal skewed datasets with the use of stab-forest index.

In [15] they discuss and find algorithms for answering reachability and path queries on temporal bipartite graphs by creating a TBP-Index that is based on 2-hop labeling. In [76] they create TopChain as an indexing method for answering reachability and time-based path queries on temporal graphs by transforming the temporal graph to DAG and decomposing it into chains and then ranking them.

In [56] they discuss the closeness centrality problem on evolving graphs and design algorithms to compute them (evolving graphs are modeled as a sequence of snapshots)

In [79] they propose the TG-index to deal with time-aware spatial keyword (TSK) and top-k queries (caring about both distance/time to reach a POI and text similarity with the query) focusing on applying it to road networks. To optimize the queries, they index data to both textual and temporal information and employed pruning algorithms.

In [45] they propose keyword search on temporal graphs, using a path iterator to find the best path between two nodes (according to three ranking factors) and furthermore develop top-k queries. They also create a query syntax that follows the syntax of TSQL2.

In [42], they design dynamic programming algorithms to solve temporal path queries, and compare them to one-pass algorithms (they use intervals while one-pass algorithms use points in time). More specifically, they focus on shortest (SDP) and fastest (FDP) path algorithms and restricted minimum temporal paths (REDP and RLDP) (restricted-earliest arrival and latest-departure).

In [2] they do proximity based community detection on temporal graphs by converting the temporal graph to static graph and apply the Louvain algorithm for community detection.