SciDG: Benchmarking Scientific Dynamic Graph Queries

Chenglin Zeng
Computer Network Information Center,
Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
zengchenglin@cnic.cn

Chuan Hu
Computer Network Information Center,
Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
huchuan@cnic.cn

Huajin Wang*
Computer Network Information Center,
Chinese Academy of Sciences
Beijing, China
wanghj@cnic.cn

Zhihong Shen*
Computer Network Information Center,
Chinese Academy of Sciences
Beijing, China
bluejoe@cnic.cn

ABSTRACT

Dynamic graphs are increasingly being utilized in domain knowledge modeling and large-scale scientific data management. Managing dynamic graph data requires a graph database system that can handle constantly changing volumes and data versions, while maintaining an acceptable query latency related to versioning. To understand how the design of storage structures affects database performance and assist scientific application developers in finding the optimal storage structure for their dynamic graph application scenarios, we have designed an easy-to-use benchmark framework called SciDG. We also conducted a study on the latencies of five fundamental version-related queries for various scientific application scenarios using SciDG. We evaluated the performance of databases based on three distinct storage principles: Sp-DB, Dp-DB, and Tp-DB. The experimental results indicate that SciDG is a valuable tool for assessing the strengths and weaknesses of different storage structures for dynamic graphs in various scenarios. Additionally, it assists scientists in selecting the most suitable dynamic graph database system for their work.

CCS CONCEPTS

• **Information systems** → *Information storage systems*; *Database design and models*; *Graph-based database models.*

KEYWORDS

version-related query, scientific data, dynamic graph database

ACM Reference Format:
Chenglin Zeng, Chuan Hu, Huajin Wang, and Zhihong Shen. 2023. SciDG: Benchmarking Scientific Dynamic Graph Queries. In *35th International Conference on Scientific and Statistical Database Management (SSDBM 2023), July 10–12, 2023, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3603719.3603724

*Corresponding authors.

SSDBM 2023, July 10–12, 2023, Los Angeles, CA, USA

ACM ISBN 979-8-4007-0746-9/23/07.
https://doi.org/10.1145/3603719.3603724

1 INTRODUCTION

Graph structures are widely used for modeling the relationships among building blocks of complex systems in biology and medicine[4]. With technological advancements, high-resolution devices now generate increasingly sophisticated data, enabling quantitative research on the time-varying relationships among biological entities within organisms[12, 25, 30]. Consequently, scientists require more expressive data structures to store these time-varying relationships. A typical choice for such a data structure is a dynamic graph, where relationships can change and the changes are retained. Examples of dynamic graphs in scientific scenarios include cellular component graphs and disease spreading graphs[32, 34].

We focus on dynamic graphs, which are graphs that can change over time due to various events, including not only changes to edges but also to nodes (such as node addition and deletion, node splitting, and node merging) in a continuous evolution[19]. It is crucial to keep track of versions of time-varying datasets and be able to retrieve them on demand. For example, we may want to understand how clusters in the protein-protein interaction (PPI) network evolve over time, determine the order of protein interactions over a period, or analyze how information spreads over time. Historical or version-related queries also consider the temporal aspect of the graph. For instance, researchers might inquire about who had the highest PageRank centrality in a citation network in 2018, which year between 2001 and 2004 had the smallest network diameter, or the number of new triangles formed in the network over the past year. These questions aid researchers in gaining a better understanding of the evolution of academic fields. Additionally, there is a need for visualizing graph changes over time[22, 23, 29, 39].

There are numerous graph database systems capable of storing and retrieving historical graph/network information at low costs. Some of these systems include *DeltaGraph*[20], *ImmortalGraph*[27], and *RailwayDB*[36]. *DeltaGraph* is a novel, extensible, highly tunable, and distributed hierarchical index structure

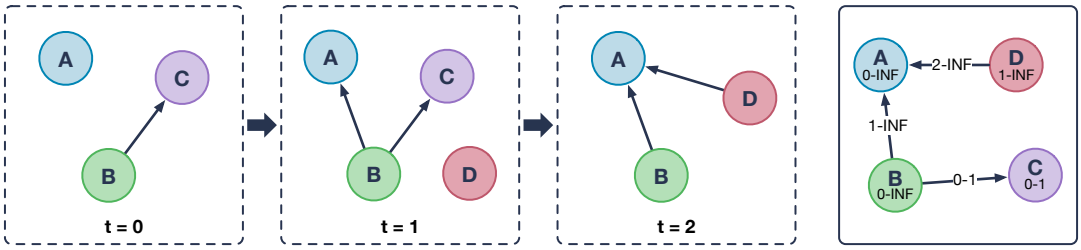

Figure 1: Dynamic Graph Instance. Independent model and timestamp model of 3 versions.

that efficiently records historical information. It supports the retrieval of historical graph snapshots for single-site or parallel processing. *ImmortalGraph*, along with recent systems such as *GreyCat*[14], *GraphOne*[21], *RisGraph*[9], *Raphtory*[37], *TeGraph*[3], and *Gradoop*[33], is a storage and computation engine specifically designed for temporal graphs. It retains all historical information of a graph, enabling efficient temporal graph queries and iterative temporal graph computations. Additionally, it introduces the concept of *snapshot groups* as a fundamental unit for storing and processing temporal graphs, capturing temporal graph information within specified time intervals. *RailwayDB* includes an entry point called interval queries, supported by the *Interval Query Index*, which indexes the temporal neighbor list time ranges of head vertices within each block. This index allows locating all head vertices (along with their block IDs) involved in interactions with timestamps within a given time range.

Each of the aforementioned systems has its unique storage design and target application scenarios. When conducting scientific research, scientists often face the challenge of selecting suitable systems to alleviate the burden of managing experimental data and contribute to their work. With numerous options available, it can be difficult to determine which system is best suited for a specific scenario. This is particularly true when dealing with complex scientific scenarios that require specialized knowledge and expertise. For instance, in the field of biology, efficient processing is essential. Queries that identify proteins interacting with each other over a specific period of time need to be conducted with low latency. In such cases, Snapshot-based Policy and Timestamp-based Policy are prudent choices. However, if the Delta-based Policy is chosen, processing delays could have adverse effects on biological research. Therefore, it is crucial for scientists to have access to reliable information and resources that assist them in making informed decisions regarding which systems to employ in their research.

To address this challenge, SciDG has been introduced as a new benchmark framework that serves as a foundational tool for evaluating the efficiency of graph database systems in handling dynamic graph data. By utilizing SciDG, scientists and practitioners can assess the effectiveness of existing policies and strategies more effectively, while also exploring novel approaches to improve performance. This benchmark provides a rigorous evaluation framework that enables fair comparisons of different systems and algorithms, fostering the development of more innovative and effective solutions for managing dynamic graph data. Consequently, SciDG represents a significant advancement for scientists aiming to leverage cutting-edge technologies and methodologies to advance their work in this critical field.

In this paper, we make several contributions to the field of dynamic graph storage and querying. Our contributions are as follows:

- We provide a theoretical blueprint (Section 3) for categorizing version-related or historical graph queries and query operators in various scientific experimental scenarios. This analysis serves as an important foundation for understanding the challenges associated with managing and retrieving dynamic graph data.
- We introduce SciDG (Section 4), a benchmark framework that utilizes real-world dynamic graph data to generate test datasets and enables the execution of various version-related or historical graph queries. By leveraging SciDG, scientists and practitioners can effectively evaluate the performance of different data storage policies and strategies when dealing with dynamic graph data.
- To set a baseline (Section 5 and Section 6) and illustrate our conclusions (Section 8), we have implemented current dynamic graph storage policies and evaluated them using SciDG. Through these evaluations, we demonstrate the strengths and limitations of existing data storage policies and strategies, while also highlighting opportunities for improvement.

Taken together, our contributions provide valuable insights into the challenges and opportunities associated with managing and retrieving dynamic graph data. They serve as an important resource for scientists and practitioners seeking to advance the state-of-the-art in this critical area.

2 PRELIMINARIES

We briefly summarize the important findings of our previous work, where we surveyed current storing techniques for dynamic graph data. One use case is depicted in Figure 1, which illustrates an instance of a graph archive with multiple versions:

- The first version, denoted as V_0, consists of three nodes (A, B, C) and one relation between B and C (B->C).
- In the second version, V_1, one node (D) and one relation (B->A) were added.
- In the latest version, V_2, a new relation (D->A) was added, while the node C with its relation (B->C) were deleted before this version.

The right part of Figure 1 provides a clear visualization of the creation and deletion of nodes and relations in each version.

2.1 Dynamic Graph Data Model

The most basic model of a graph over a period of time is as a collection of graph snapshots, where each snapshot corresponds to a specific time instance (assuming discrete time). Each graph snapshot consists of a set of nodes and a set of edges. Nodes and edges are assigned unique IDs at the time of their creation, which are not reassigned even after components are deleted. If a component is deleted and then re-inserted, it will be assigned a new ID. A node or an edge can be associated with a list of attribute-value pairs. The list of attribute names is not fixed, and new attributes can be added at any time. Additionally, an edge contains information about whether it is a directed edge or an undirected edge.

In our paper, we adopt the traditional approach of representing (static) graph data without temporal information using ordered pairs:

$$G = (V, E) \tag{1}$$

Here, V represents the set of vertices or nodes, E represents the set of edges or relations, and each element e in E is a binary tuple consisting of two elements u and v from V, written as *e=(u, v)*.

For dynamic graph data, we define it as follows:

$$G[t_i, t_j] = \left\{G_{t_i}, G_{t_{i+1}}, \cdots, G_{t_j}\right\} \tag{2}$$

$$G_k = (V_k, E_k) \quad (k \in [t_i, t_j]) \tag{3}$$

and each $G_x(x \in [t_i, t_j])$ represents a snapshot of dynamic graph data, also known as a version at time t_x. We define an event as the recording of an atomic activity in the network. An event can involve the creation or deletion of an edge or node, or a change in attribute values of a node or an edge. For example, let's consider the graph snapshots G_k and G_{k-1} at times t_k and t_{k-1} respectively. If Δ(delta) represents the set of all events that occurred between t_{k-1} and t_k, we can express:

$$G_k = G_{k-1} + \Delta \qquad G_{k-1} = G_k - \Delta \tag{4}$$

The + and - operators indicate the application of the events in Δ(delta) in the forward and backward directions, respectively.

2.2 Storage Strategies

Main research efforts addressing the challenge of dynamic graph storing fall in one of the following three storage strategies:*snapshot-based policy (SP), delta-based policy (DP) and timestamp-based policy (TP).*

- `Snapshot-based Policy (SP)`: This is a basic policy that manages each version separately, ensuring an isolated dataset to maintain consistency and prevent conflicts between different versions of the same data. However, as more versions are added, duplicating the static information across all versions can pose scalability issues. This is because whenever a new version is created, all the static information needs to be copied, resulting in increased storage requirements and potential performance slowdowns. Additionally, apart from simple retrieval operations like single-version queries, more complex operations would demand significant processing efforts.
- `Delta-based Policy (DP)`:
  This partially addresses the previous scalability issue by computing and storing the differences (deltas) between versions. In this policy, only the changes are recorded in the database, annotated by time. While this approach is space-optimal and supports O(1)-time updates (for transaction-time databases), answering a query may require scanning the entire list of changes, which can be time-consuming. Generally, a combination of these two approaches, known as "Copy+Log" [24, 35], where a subset of the snapshots is explicitly stored, is often a better idea.
- `Timestamp-based Policy (TP)`: This can be seen as a specific case of time modeling in graph databases, involving the annotation of each element (node or relationship) with its temporal validity, representing the time period during which it is considered valid. To avoid duplicating information and conserve storage space, some practical proposals suggest annotating elements only when they are added or deleted, utilizing two distinct fields: the "created" timestamp and the "deleted" timestamp (if applicable). This approach allows for storing only the modified elements, along with their corresponding created and/or deleted timestamps, when creating a new version, rather than duplicating the entire dataset. Consequently, this technique significantly reduces storage costs and enhances performance, particularly for larger and more intricate datasets.

It is evident that these strategies have their respective advantages and disadvantages. Storing policies play a crucial role in scientific data management as they dictate how data is stored and accessed over time. When selecting a storing policy, careful consideration must be given to the specific scientific scenarios and types of queries that will be conducted. Different scientific disciplines have unique data management requirements that should be taken into account when formulating a storing policy. Additionally, the anticipated types of queries performed on the data should also influence the choice of storing policy. Other factors to consider include data storage costs, security requirements, and data ownership and access rights. These considerations are vital in ensuring proper management and preservation of data over time while meeting the needs of scientists who rely on this data for their research.

Overall, selecting the appropriate storing policy necessitates careful consideration of various factors, such as scientific scenarios, query requirements, data storage costs, and security and access considerations. By adopting a thoughtful and comprehensive approach to storing, scientists can guarantee that data is stored and managed in the most efficient and effective manner, while also adhering to any pertinent regulatory or compliance requirements.

3 RETRIEVAL FUNCTIONALITY

Given the relative novelty of storing and querying dynamic graph data, retrieval needs are not fully described or widely implemented in practical scenarios. We propose a classification that distinguishes five different types of retrieval needs. This categorization considers a version of dynamic graph data as a set and primarily relies on set operations, without considering the internal filters of a single graph for convenience.

- `Single-version queries`: Retrieving a complete version of dynamic graph data is often an essential requirement for effective analysis and scientific research. This demand involves

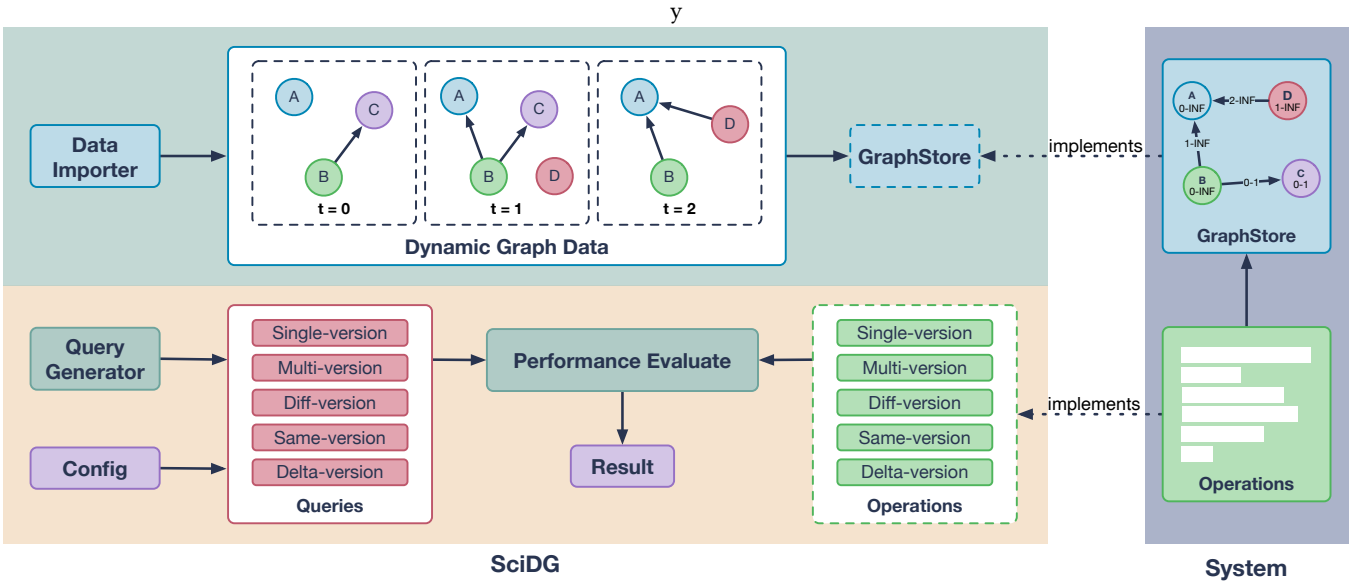

Figure 2: The Architecture of SciDG. SciDG has three parts Data Importer, Query Generator, and Performance Evaluate, two interfaces GraphStore and Operations for System to implement. Data Importer serves for GraphStore, Performance Evaluate calls the Operations functions implemented by System and consumes the Queries generated by Query Generator. Users can set parameters in Config part.

selecting a specific set from all available sets, enabling scientists to access and analyze the complete history of changes to the graph over time. In fact, this is a common feature provided by revision control systems and other large-scale archives. For example, it is used to visualize and analyze the evolution of networks[6]. Given a specific graph data version t_i that we want to query, the resulting graph G_q can be described as follows:

$$G_q = \{(v, e) | v \in V_{t_i}, e \in E_{t_i}\}$$

- `Multi-version queries`: These queries retrieve nodes and relations that exist in at least one of the assigned versions. For example, they are used to query interaction relations between proteins with high weights during specific time periods for further research in the field of biology. Given the versions t_i,t_j and t_k as examples, the resulting graph G_q can be described as follows:

$$V_{ijk} = (V_{t_i} \cup V_{t_j} \cup V_{t_k})$$
$$E_{ijk} = (E_{t_i} \cup E_{t_j} \cup E_{t_k})$$
$$G_q = \{(v, e) | v \in V_{ijk}, e \in E_{ijk}\}$$

- `Diff-version queries`: Retrieving nodes and relations that belong to the first version but not the second version is often used to query newly increased coauthor relations compared to the previous year. Given the first version t_i and the other version t_j, the resulting graph G_q can be defined as follows:

$$G_q = \{(v, e) | v \in V_i \wedge v \notin V_j, e \in E_i \wedge e \notin E_j\}$$

- `Same-version queries`: Retrieving nodes and relations that exist in all of the given versions, which can be thought of as the intersection of sets, offers the ability to query long-term stable partners of an author in the coauthor network. Additionally, given the versions t_i,t_j and t_k, the resulting graph G_q is presented here:

$$V_{ijk} = (V_{t_i} \cap V_{t_j} \cap V_{t_k})$$
$$E_{ijk} = (E_{t_i} \cap E_{t_j} \cap E_{t_k})$$
$$G_q = \{(v, e) | v \in V_{ijk}, e \in E_{ijk}\}$$

- `Delta-version queries`: These queries retrieve the differences between two or more given versions. This functionality is largely related to the changes that involve added and deleted elements (nodes and relations) during the graph evolution process. For instance, it can identify which proteins have recently formed new interactions and which proteins have lost connections with others over a given period of time. If given the versions t_i and t_j, the delta between them can be represented mathematically as follows:

$$V_{\Delta^-} = \{v | v \in V_i \wedge v \notin V_j\} \quad E_{\Delta^-} = \{e | e \in E_i \wedge e \notin E_j\}$$
$$V_{\Delta^+} = \{v | v \in V_j \wedge v \notin V_i\} \quad E_{\Delta^+} = \{e | e \in E_j \wedge e \notin E_i\}$$
$$V_\Delta = \{v | v \in V_{\Delta^+} \wedge v \in V_{\Delta^-}\} \quad E_\Delta = \{e | e \in E_{\Delta^+} \wedge e \in E_{\Delta^-}\}$$
$$G_q = \{(v, e) | v \in V_\Delta, e \in E_\Delta\}$$

Additionally, when it comes to retrieving data from a dynamic graph, traditional graph queries are often not considered, as many graph databases are capable of efficiently processing these queries.

4 SCIDG: BENCHMARK DESCRIPTION

Previous considerations on dynamic graph storage policies and retrieval functionality lay the foundation for future directions in evaluating the efficiency of on-demand retrieval. This section introduces our SciDG benchmark, which applies our blueprints to real-world scenarios. We first provide a detailed description of the benchmark architecture, data configuration, and query set that covers basic retrieval needs. The next section evaluates the SciDG benchmark using state-of-the-art storage policies. All benchmark data, queries, policy implementations, and additional results can be accessed at the SciDG repository[1].

4.1 Benchmark Architecture

This section provides an overview of the architecture of SciDG, which is an extensible and configurable benchmark that easily allows the addition of datasets and preparation of queries. Figure 2 illustrates the architecture of SciDG. The benchmark flow starts with the *Data Importer*, which reads the dataset and provides the store engine with graph data for each version. This data is accepted by the *GraphStore* implemented by the database. The *QueryGenerator* then reads the dataset configuration to generate queries that can be optimized by users according to their evaluation requirements. Subsequently, the *Performance Evaluator* reads the previously generated queries and feeds them to the *Query Engine Operations* implemented by the database.

SciDG offers extensibility through its abstract interfaces and resilient configurations. It provides abstractions for database access, workloads, and data importers, making it easy to extend for additional benchmark scenarios. For example, while SciDG uses three distinct datasets by default, additional datasets can be added by providing implementations of the *Data Importer* and *Config* interfaces. Similarly, additional queries and new database servers can be incorporated by extending the relevant interfaces, namely *Operations* and *GraphStore*.

4.2 Benchmark Data

While evolving graph data is ubiquitous, few works systematically provide and maintain a clear and large corpus of graph versions over time. The SciDG benchmark data consists of three independent dynamic graph datasets.

One of them is the Dynamic coauthor network [40], which constructs an evolving coauthor network from ArnetMiner5. This dataset is collected from 1,768,776 publications published between 1986 and 2012, involving 1,629,217 authors. Each year is considered as a time stamp, resulting in a total of 27 time stamps. At each time stamp, an edge is created between two authors if they have coauthored at least one paper within the most recent 3 years, including the current year. The originally undirected coauthor network was converted into a directed network by considering each undirected edge as two symmetric directed edges. Details of the nodes and relations in the coauthor data are illustrated in Figure 3.

Next in line is the DPPIN [11]. In this proposed repository, DPPIN consists of a total of 12 individual dynamic network datasets at different scales. Each dataset represents a dynamic protein-protein interaction network that describes protein-level interactions in yeast cells. A protein-protein interaction network is represented as an undirected graph, where each node represents a gene coding for a protein, and each edge represents a protein-protein interaction. The details of each dataset in DPPIN are shown in Table 1.

The last framework is designed for the generation of synthetic, evolving data represented in the property graph model. It builds upon the EvoGen Benchmark[26], which provides capabilities for generating synthetic data and workloads. The EvoGen Benchmark is based on the widely adopted Lehigh University Benchmark[15]. Both benchmarks offer data represented in the RDF model, which may not be suitable for most graph databases for archival purposes. Consequently, the data generation framework implements parameters for instance evolution[26] and provides data in the property graph model stored using CSV files.

Additionally, the proposed data generation framework allows control over the change rate and version counts of data evolution, as well as the ability to set the change type (added or deleted). Figure 4 presents the details of the dynamic graph data generated by the framework using parameters such as a change rate of 0.3 and a version count of 26. This generated data is also utilized in our subsequent benchmark experiments, referred to as 'Generated data'.

4.3 Benchmark Queries

The challenging task for any benchmark is to provide meaningful and comprehensive queries that allow for testing a wide range of features. As suggested in the previous blueprints (Section 2), we have decided to initiate our dynamic graph storage benchmark by sampling fundamental lookup queries, denoted as *Q*. These queries encompass various forms such as query Single Version, query delta between versions, query intersection of versions, query union of versions, and query differences between versions. By starting with these fundamental lookup queries, they can serve as a foundation for developing more complex and sophisticated queries. Furthermore, these lookup queries can also be utilized to leverage the functionality design of dynamic graph version management systems.

Assuming that the data is defined using the constructs of the property graph model[16], the data retrieval needs can be described using the Cypher language[10]. In order to enhance convenience, we have added certain keywords. For instance, the keyword ***at*** indicates a specific assigned data version, the keyword ***except*** signifies keeping the differences in the results of two *match clauses*, and the keyword ***intersection*** denotes an operation for obtaining the common parts from two ***match clauses***.

With this schema, the queries and their equivalents in the Cypher Query Language (CQL) can be described as follows:

(Q1) Single Version Data Fetching (Coauthor data): Retrieve all the nodes and relationships of a specific version. This query is used to visualize and analyze the evolution of coauthor networks.

```
MATCH (n:author)-[:coauthor]->(m:author)
AT v1
RETURN n, n.name, m, m.name
```

(Q2) Multiple Versions Data Fetching (Coauthor data): Retrieve all the nodes and relationships across multiple versions. This query

[1] https://github.com/codeBabyLin/SciDG.git

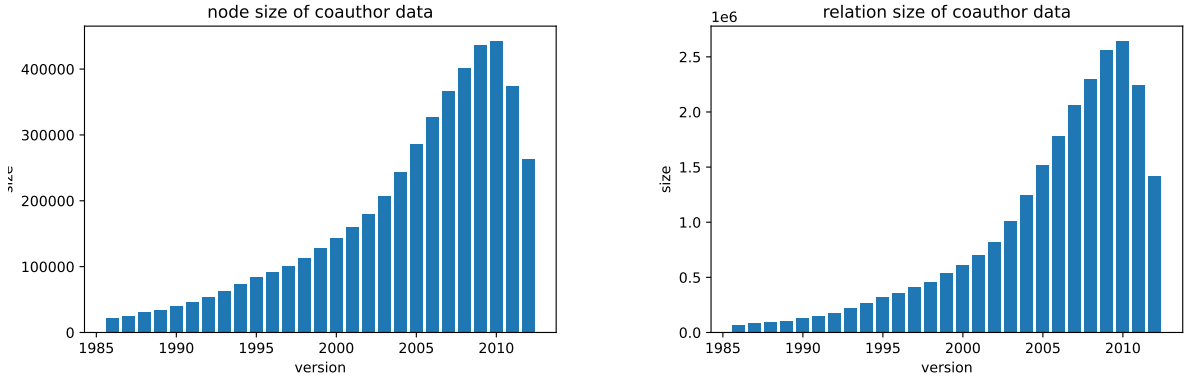

Figure 3: Scales of nodes and relations of Coauthor data, left subfigure presents the size of nodes in each version and right subfigure shows the size of relations in each version, versions are defined with years from 1986 to 2012

Table 1: Dynamic Network Data sets in DPPIN

Dynamic PPINs	Nodes	Edges	Node Features	Edge Features	Node Label Rate	Timestamps
DPPIN-Uetz	922	2,159	Y	Y	921/922 (99.89%)	36
DPPIN-Ito	2,856	8,638	Y	Y	2854/2856 (99.93%)	36
DPPIN-Ho	1,548	42,220	Y	Y	1547/1548 (99.93%)	36
DPPIN-Gavin	2,541	140,040	Y	Y	2538/2541 (99.88%)	36
DPPIN-Krogan (LCMS)	2,211	85,133	Y	Y	2208/2211 (99.86%)	36
DPPIN-Krogan (MALDI)	2,099	78,297	Y	Y	2097/2099 (99.90%)	36
DPPIN-Yu	1,163	3,602	Y	Y	1160/1163 (99.74%)	36
DPPIN-Breitkreutz	869	39,250	Y	Y	869/869 (100.00%)	36
DPPIN-Babu	5,003	111,466	Y	Y	4997/5003 (99.88%)	36
DPPIN-Lambert	697	6,654	Y	Y	697/697 (100.00%)	36
DPPIN-Tarassov	1,053	4,826	Y	Y	1051/1053 (99.81%)	36
DPPIN-Hazbun	143	1,959	Y	Y	143/143 (99.89%)	36

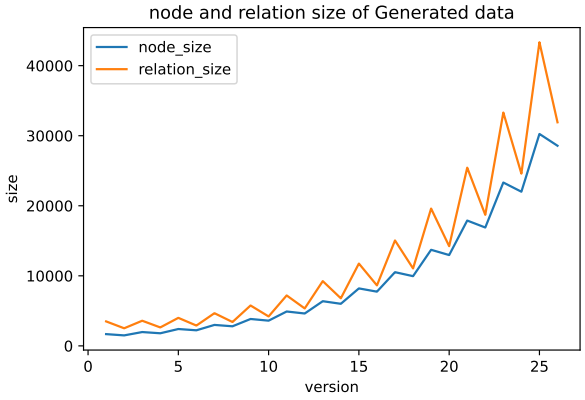

Figure 4: Scales of nodes and relations of data generated by generation framework with parameters change rate = 0.3 and version number = 26

is used to determine the year in which an author had the highest number of papers published.

```
MATCH (n:author) AT v1
WHERE n.name = n1
WITH n, size((n)-[]->()) AS numOfpapers
RETURN n.name, numOfPapers
UNION
MATCH (n:author) AT v2
WHERE n.name = n1
WITH n, size((n)-[]->()) AS numOfpapers
RETURN n.name, numOfPapers
ORDER BY numOfPapers DESC LIMIT 1
```

(Q3) Different Data of Versions Fetching (Coauthor data): Retrieve the differences in graph data between two versions. This query is often used to identify newly established coauthor relationships compared to the previous year.

```
MATCH (n:author)-[:coauthor]->(m:author)
AT v1
RETURN n, n.name, m, m.name
EXCEPT
MATCH (n:author)-[:coauthor]->(m:author)
AT v2
RETURN n, n.name, m, m.name
```

(Q4) Same Data of Versions Fetching (Coauthor data): Retrieve the identical nodes and relationships from graph data across versions. This query allows for identifying long-term stable partners of an author.

```
MATCH (n:author)-[:coauthor]->(m:author)
AT v1
WHERE n.name = n1
RETURN n, n.name, m, m.name
INTERSECTION
MATCH (n:author)-[:coauthor]->(m:author)
AT v2
WHERE n.name = n1
RETURN n, n.name, m, m.name
```

(Q5) Delta of Versions Fetching (Coauthor data): Delta of Versions Fetching (Coauthor data): Retrieve the changes (including added and deleted elements) in graph data across versions. This query is primarily applied in scenarios where the evolutionary process of coauthor networks is being studied.

```
MATCH (n)-[r:coauthor]->(m)
AT v1
RETURN n, r, m
EXCEPT
MATCH (n:author)-[:coauthor]->(m:author)
AT v2
RETURN n, r, m

UNION
MATCH (n)-[r:coauthor]->(m)
AT v2
RETURN n, r, m
EXCEPT
MATCH (n:author)-[:coauthor]->(m:author)
AT v1
RETURN n, r, m
```

(Q6) Single Version Data Fetching (DPPIN data): Retrieve all the nodes and relationships of a given version. This query is used to visualize and analyze the interaction of Protein-Protein over time.

```
MATCH (n)-[r:interaction]->(m)
AT v1
RETURN n, n.value, m, m.value, r.weight
```

(Q7) Multiple Versions Data Fetching (DPPIN data): Retrieve all the nodes and relations across versions. This query is used to retrieve interaction relations between proteins that have high weights during a specific period of time for further research in the field of biology.

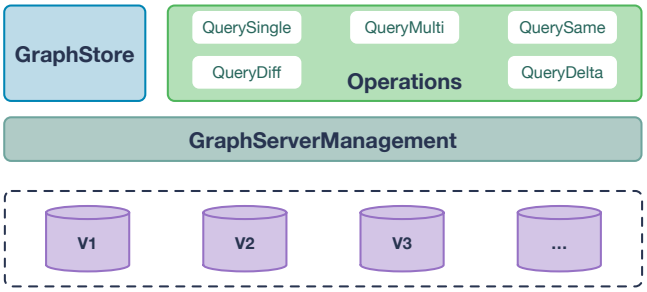

Figure 5: The Architecture of Sp-DB which implemented GraphStore and Operations interfaces of SciDG, in Sp-DB, each version of data is stored in an independent database instance

```
MATCH (n)-[r:interaction]->(m)
AT v1
WHERE r.weight >= w1
RETURN n, r,m
UNION
MATCH (n)-[r:interaction]->(m)
AT v2
WHERE r.weight >= w1
RETURN n, r,m
```

As for the generated data, it is primarily used to evaluate the space utilization efficiency of different storage policies. Here, we have designed some queries that are similar to those used with coauthor data but simpler in nature. In this section, we present our benchmark data and queries based on retrieval functionality derived from real scientific scenarios. We leverage these scenarios to develop proof-of-concept scenarios and evaluate systems that implement the SP, DP, and TP using Neo4j Graph Databases[38].

4.4 System Implementation

For the SP policy, each version is indexed in an independent Neo4j Graph Facade instance (Sp-DB), which means that each version of dynamic graph datasets can be seen as an independent static graph dataset. The architecture of Sp-DB is shown in Figure 5. The main component of this system is the *GraphServerManagement*, which implements the interfaces *GraphStore* used for ingesting dynamic graph data, and *Operations* containing five basic query functions. Dp-DB and Tp-DB share identical characteristics. When querying an assigned version of dynamic graph data (Single-version queries), the corresponding Neo4j instance is activated, and the query operations are performed. As for other query types (Multi-version queries, Diff-version queries, Same-version queries, and Delta-version queries) that are based on the results of previous Single-version queries, they are executed accordingly.

Similarly, for the DP policy, we create an index (delta store file) for each added and deleted element (delta), again for all versions, using independent Neo4j instances to store the versions with critical importance of dynamic graph (Dp-DB). The architecture of Dp-DB is displayed in Figure 6, where every 3 versions have one independent physical store instance and two delta files used to save

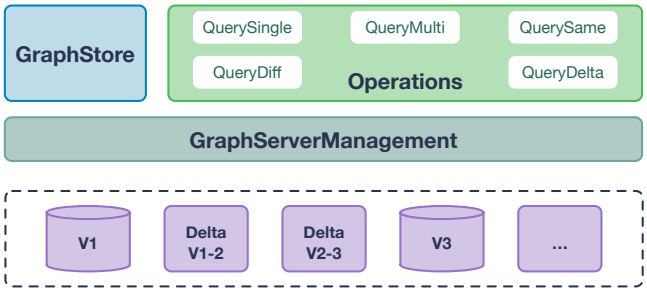

Figure 6: The Architecture of Dp-DB which implemented GraphStore and Operations interfaces of SciDG, in Dp-DB, every 3 version of data is stored in an independent database instance, and the others are stored with delta files

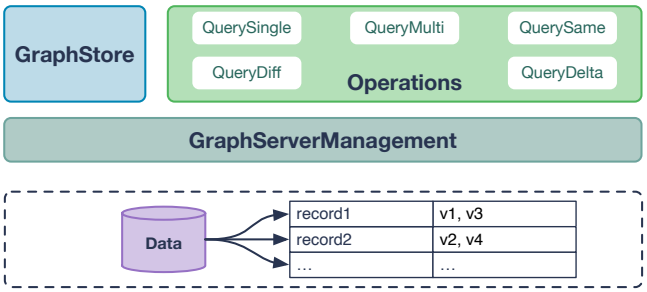

Figure 7: The Architecture of Tp-DB which implemented GraphStore and Operations interfaces of SciDG, in Dp-DB, all the versions of data are stored in an independent database instance, but every recorder is annotated with created version and deleted version

deltas. This design ensures that regardless of the version being queried, only one redo or undo operation is necessary, resulting in space savings and improved efficiency. When querying an assigned version of dynamic graph data (Single-version queries), it first loads the nearest data version closest to the given version and then performs redo or undo operations according to the delta files, before retrieving the graph data. When querying the changes between continuous versions (Diff-version queries and Delta-version queries), it can utilize the delta file indexing each added and deleted element to improve the data retrieval efficiency. As for other types of queries, it is similar to Sp-DB.

Lastly, for the TP policy, we followed the approach of [7] and annotated all the elements (nodes and relations) of the dynamic graph data with their created and deleted versions in a single Neo4j instance (Tp-DB). The architecture of Tp-DB is presented in Figure 7, where nodes and relations are annotated with two versions, indicating the start time and end time respectively. When querying an assigned version of dynamic graph data (Single-version queries), it scans all the elements and selects nodes and relations whose lived version time (versions between created and deleted versions) covers the given query version. When querying graph data existing in multiple versions or in every given version, it only needs to scan all the nodes and relations once and select eligible elements during the scanning process. As for other types of queries, they are similar to Sp-DB.

5 EXPERIMENTS SETUP

We use one machine (M1) for our benchmarks to perform our tests. It is equipped with an Intel(R) Core(TM) CPU i7-8700 @ 3.20 GHz, comprising 12 logical cores and 16 GB DDR3 RAM. We activate the databases (Sp-DB, Tp-DB, Dp-DB) one by one to execute the queries as denoted in **section 4.3**, while monitoring the task management to ensure that no other processes are exerting influence on it.

For our tests, we utilize the entire coauthor dataset, a subset of DPPIN data (DPPIN-Gavin), and a dataset generated by the framework with a change rate of 0.3 and a version count of 26. We consider the following evaluation indicators to assess the dynamic graph version management systems:

- `Import Time Cost`: The ingestion time of dynamic graph data is a crucial performance indicator for evaluating the update capacity of various systems. The ability to efficiently process and incorporate new data into the existing graph structure can greatly influence the overall efficiency and accuracy of the system. This metric is particularly significant in dynamic graph scenarios where the graph structure undergoes frequent changes and requires frequent updates. Systems with faster ingestion times are usually favored as they can keep pace with the rapidly evolving nature of dynamic graphs, enabling more timely and precise analysis and insights.
- `Disk Space Cost`: The disk space cost is a performance indicator that quantifies the amount of disk space needed for storing and managing dynamic graph data. It reflects how effectively a system utilizes disk space to accommodate large volumes of data. Systems that optimize their disk usage can store more data on the same hardware resources, resulting in enhanced performance and cost savings. Measuring disk space costs enables scientists and practitioners to compare different systems in terms of their efficiency and effectiveness in managing dynamic graph data. Ultimately, this facilitates the identification of the most suitable system for specific use cases and workload requirements.
- `Queries-Per-Second(QPS)`: Throughput is a critical performance metric for measuring the query processing capabilities of various systems. It reflects the system's ability to handle high volumes of queries and deliver prompt responses. In dynamic graph scenarios, it is crucial to have high query processing throughput to ensure timely and accurate data analysis. By evaluating and comparing the throughput of different systems, scientists and practitioners can select the most suitable system for specific use cases and workload demands.

Apart from the above-mentioned indicators, we also focus on monitoring CPU and memory usage during tests. High CPU and memory usage levels can indicate potential scalability issues or resource constraints that require attention. Monitoring these metrics is crucial as part of a comprehensive testing strategy to ensure optimal system performance under varying load conditions.

Table 2: Import Time(in ms)

database\dataset	Coauthor	Generated	DPPIN-Gavin
Sp-DB	8237998	950853	214735
Dp-DB	1061472	218850	26719
Tp-DB	816152	173554	26277

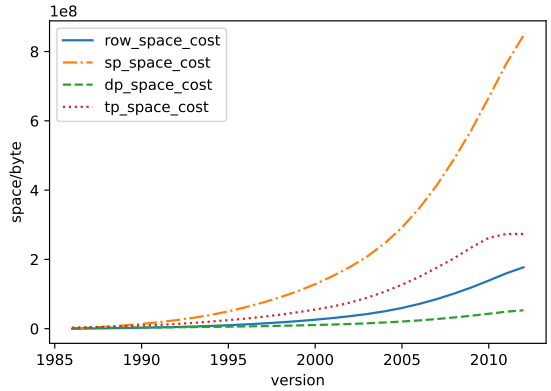

Figure 8: Space cost of different policies for storing Coauthor data

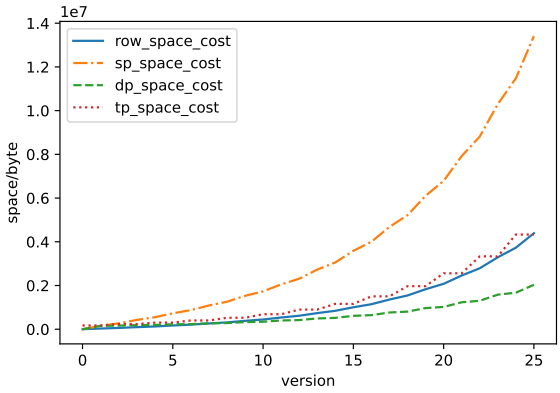

Figure 9: Space cost of different policies for storing Generated data

Table 3: Query Latency Statistics (in ms) for Q1

Database	Min.	Mean	Max.	QPS
Sp-DB	9095	9202	9388	1.0867
Dp-DB	43833	44039	44243	0.2260
Tp-DB	10158	10252	10349	0.9754

Table 4: Query Latency Statistics (in ms) for Q2

Database	Min.	Mean	Max.	QPS
Sp-DB	408683	411444	413929	0.0246
Dp-DB	783218	786085	788492	0.0128
Tp-DB	415062	419124	422619	0.0247

6 EXPERIMENTS RESULTS

6.1 Import Time and Space Results

Table 2 presents the import time cost of three different storage strategies. It is evident that larger data sizes require more time for databases to import. When comparing Sp-DB and Dp-DB to Tp-DB, it becomes apparent that Tp-DB has advantages in efficiently ingesting dynamic graph data.

Figures 8-9 illustrate the on-disk space requirements for the raw data and the various policies as the versions of dynamic graph data grow. By comparing these figures with the size of the different policies, we can describe their inherent overheads. The SP policy indexing (in Neo4j) requires several times more disk space than the raw data, primarily due to increasing data redundancy across versions. On the other hand, DP occupies the least amount of data space, thus reducing space needs as expected. Finally, TP reports a similar size to the raw data space. Combining the perspectives from Figure 4 and Figure 9, it can be observed that the more deleted elements there are, the less disk space TP requires.

6.2 Retrieval Performance

Next, we evaluate the overall retrieval performance of the storage policies. Based on our blueprints, we use the queries specified in **Section 4.3** as the target query for each case. In general, our evaluation confirmed our assumptions and assessments regarding the characteristics of the policies. The SP and TP policies exhibited a consistently stable behavior in our tests, whereas the DP policy had higher retrieval times compared to the others due to its reduced disk space usage. The main difference between SP and TP lies in the slightly higher retrieval time of SP, which is attributed to the operation of scanning all elements. We will then proceed to present and discuss selected plots for each query operation.

Q1. Single Version Data Fetching (Coauthor data). It queries the database to fetch the specific version data of coauthor data. There are a total of 10 versions randomly generated within the range from 1986 to 2012. Table 3 presents the query latency statistics in milliseconds for Q1. The lowest query latency is observed on Sp-DB, with a maximum value of 9388 ms and an average value of 9202 ms. Naturally, Sp-DB also has the highest QPS value of 1.0867. Tp-DB performs as the second-best, with a maximum value of 10349 ms and an average value of 10252 ms, closely followed by a secondary QPS value of 0.9754, which is similar to that of Sp-DB. Dp-DB ranks third, with a maximum value of 44243 ms and an average value of 44039 ms. It also has a lower QPS compared to Sp-DB and Tp-DB.

Q2. Multiple Versions Data Fetching (Coauthor data). We query the database to fetch multi-version data of coauthor data, with each multi-version query consisting of several randomly generated versions. The query latency statistics for Q2 are presented in Table 4. Sp-DB achieves the first-place position with a maximum value of

Table 5: Query Latency Statistics (in ms) for Q3

Database	Min.	Mean	Max.	QPS
Sp-DB	37341	38722	39675	0.2713
Dp-DB	150723	157694	168607	0.0657
Tp-DB	37452	38283	39711	0.2569

Table 6: Query Latency Statistics (in ms) for Q4

Database	Min.	Mean	Max.	QPS
Sp-DB	215994	217083	218559	0.0479
Dp-DB	658354	661437	665348	0.0154
Tp-DB	230352	234213	237111	0.0451

Table 7: Query Latency Statistics (in ms) for Q5

Database	Min.	Mean	Max.	QPS
Sp-DB	61415	64385	69056	1.5532
Dp-DB	166733	167331	167970	0.5976
Tp-DB	58007	58700	59647	1.7035

413929 ms and an average value of 411444 ms. However, the QPS of Sp-DB is much lower at 0.0246 compared to that of Q1. Tp-DB ranks second, with a maximum value of 422619 ms and an average value of 419124 ms. Dp-DB has the highest latency values, with a maximum value of 788492 ms and an average value of 786085 ms. It is worth noting that all tested systems exhibit low QPS when handling complex queries like Q2.

Q3. Different Data of Versions Fetching (Coauthor data). We query the database to calculate the differences between pairs of versions. The query latency statistics in milliseconds for Q3 are presented in Table 5. Sp-DB achieves the first-place position in Q3 performance, with a maximum value of 39675 ms and an average value of 38722 ms. Additionally, it has a QPS of 0.2713. Tp performs better than Dp-DB, with a maximum value of 39711 ms and a QPS of 0.2569, while Dp-DB records a maximum value of 168607 ms and the lowest QPS of 0.0657.

Q4. Same Data of Versions Fetching (Coauthor data). We query the database to summarize the data that exists over assigned versions. The query latency statistics in milliseconds for Q4 are presented in Table 6. Even with a complex query like Q4, Sp-DB maintains its first-place ranking, achieving a maximum value of 218559 ms and a QPS of 0.0479. Tp-DB and Dp-DB also perform reasonably well, with Tp-DB recording a maximum value of 237111 ms and Dp-DB recording a maximum value of 665348 ms. However, Dp-DB has the lowest QPS among the tested systems, with a value of 0.0154.

Q5. Delta of Versions Fetching (Coauthor data). We query the database to calculate the delta between denoted versions. The delta consists of two parts: added and deleted. Table 7 presents the query latency statistics in milliseconds for Q5. Among the systems tested, Tp-DB demonstrates the best performance, achieving a maximum value of 59647 ms and a QPS of 1.7035. In this query, Sp-DB outperforms Dp-DB, with Dp-DB recording a maximum value of 69056 ms and a QPS of 1.5532, while Sp-DB achieves a maximum value of 167970 ms and a QPS of 0.5976. It is evident that Tp-DB performs better than Sp-DB in this particular query.

Table 8: Query Latency Statistics (in ms) for Q6

Database	Min.	Mean	Max.	QPS
Sp-DB	28	29	33	344.8276
Dp-DB	414	415	417	24.2130
Tp-DB	36	38	42	270.2703

Table 9: Query Latency Statistics (in ms) for Q7

Database	Min.	Mean	Max.	QPS
Sp-DB	1195	1233	1258	8.3110
Dp-DB	7499	7640	7886	1.3742
Tp-DB	1090	1147	1227	8.8574

Q6. Single Version data Fetching(DPPIN data). We query the database for the PPIN Data. This query type is similar to Q1, but the size of DPPIN data is smaller than the coauthor data. Table 8 displays the query latency statistics in milliseconds for Q6. The QPS of every system is much higher than that in Q1 due to the smaller size of DPPIN data. Among the systems tested, Sp-DB achieves the lowest query latency with a maximum value of 33 ms and a QPS of 344.8276. Tp-DB follows closely with a maximum value of 42 ms and a QPS of 270.2703. Dp-DB still meets the deadline, recording a maximum latency of 417 ms and a QPS of 24.2130.

Q7. Multiple Versions Data Fetching (DPPIN data). We query the database to collect data from multiple versions of the PPIN data. Table 9 shows the query latency statistics in milliseconds for Q7. Among the systems tested, Tp-DB achieves the best performance in Q7 with a maximum value of 1227 ms and a QPS of 8.8574. Sp-DB outperforms Dp-DB significantly, recording a maximum value of 1258 ms and a QPS of 8.3110, while Dp-DB exhibits a higher latency with a maximum value of 7886 ms and a QPS of 1.3742.

Sp-DB provides outstanding query performance in most queries due to its independent data storage for each version. Next in line is Tp-DB, which exhibits very similar query performance to Sp-DB in most queries, and outperforms Sp-DB in certain queries like Q5 and Q7. Dp-DB consistently shows the highest latency and lowest QPS in every query due to the time required for loading delta. It is evident that as the data size (nodes and edges) increases, the query latency also increases. When the data size exceeds a million, Sp-DB demonstrates clear superiority. However, even at the hundred-thousand level, Sp-DB remains a viable option.

Additionally, considering space cost and query performance, it has been observed that Sp-DB performs well in most cases but

requires significantly more storage space. On the other hand, Dp-DB has relatively low space requirements but may have potentially slower query performance. Tp-DB strikes a balance between these two factors, offering good query performance while maintaining reasonable space costs.

7 RELATED WORK

Dynamic graph version management has become an increasingly important research area in recent years. Numerous new techniques and algorithms have been proposed to effectively handle the continuously changing nature of dynamic graphs.

One widely studied approach is the utilization of temporal graphs to depict the progression of dynamic graphs over time[13, 28]. Temporal graph models have the capability to capture the static structure of a graph along with its temporal dynamics, enabling more precise analysis and prediction of future changes. Several temporal graph models have been proposed, including the snapshot model, sliding window model, and interval-based model.

Another crucial research area is the advancement of efficient algorithms for processing and querying dynamic graphs[1]. Traditional graph algorithms are not well-suited for dynamic graphs as they necessitate frequent updates and modifications. Consequently, numerous new algorithms have been specifically designed for dynamic graphs, including incremental algorithms[18], streaming algorithms[17], and online algorithms[5].

Benchmarking is also a crucial aspect of research in dynamic graph version management. Evaluating diverse algorithms and systems using common benchmarks enables fair comparison and selection of the most effective approaches. Several dynamic graph benchmarks have been proposed, including the LDBC Social Network Benchmark[8], the Graph500 Benchmark[2], and the Twitter Hadoop Benchmark[31].

Overall, while there is still much progress to be made in dynamic graph version management, significant advances have been achieved in recent years through the development of new algorithms, models, and benchmarks. These efforts are crucial for enabling the effective management and analysis of the ever-changing world of dynamic graphs.

8 CONCLUSIONS

Dynamic graph storage is still in an early stage of research. Novel solutions face the additional challenge of comparing performance against other storage policies or schemes, as there is neither a standard way of defining a specific data corpus for dynamic graph storage nor relevant retrieval functionalities.

This paper addresses these shortcomings and provides a blueprint to guide future benchmarking of dynamic graph storage. We describe the main retrieval facilities involved in dynamic graph storage and provide guidelines for selecting relevant and comparable queries. We instantiate these blueprints in a concise benchmark called SciDG, which serves a clean and well-described benchmark corpus, as well as a query test bed comprised of basic, but well-addressed, retrieval queries. Finally, we implement and evaluate three state-of-the-art storage policies. The results clearly highlight weaknesses (especially in scalability) and strengths of current storage policies, guiding future developments.

Currently, our focus is on leveraging the presented blueprints as a basis for designing a suitable dynamic graph version control system for specific scientific scenarios. Additionally, we are working on developing novel algorithms for dynamic graph data mining.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (Grant No.2021YFF0704200) and Informatization Plan of Chinese Academy of Sciences (Grant No.CAS-WX2022GC-02).

REFERENCES

[1] Jain Anuj and Sahni Sartaj K. 2022. Algorithms for optimal min hop and foremost paths in interval temporal graphs. 7 (2022), 1–24.
[2] D. A. Bader, J. Berry, S. Kahan, R. Murphy, and J. Willcock. 2010. Graph 500 Benchmark 1 ("Search"). (2010).
[3] c huan, h liu, m liu, y liu, c he, k chen, j jiang, y wu, and sl song. 2022. TeGraph: A Novel General-Purpose Temporal Graph Computing Engine. (2022), 578–592.
[4] M. Cannataro, P. H. Guzzi, and P. Veltri. 2010. Protein-to-protein interactions: Technologies, databases, and algorithms. *Acm Computing Surveys* 43, 1 (2010), 1–36.
[5] Se-Hang Cheong, Yain-Whar Si, and Raymond K. Wong. 2021. Online Force-Directed Algorithms For Visualization Of Dynamic Graphs. 556 (2021), 223–255.
[6] T. N. Dang, N. Pendar, and Angus Graeme Forbes. 2016. TimeArcs: Visualizing Fluctuations in Dynamic Networks. *Computer Graphics Forum* 35 (2016), 61–69.
[7] Ariel Debrouvier, Matías Perazzo, Eliseo Parodi, Valeria Soliani, and Alejandro Vaisman. 2021. A Model and Query Language for Temporal Graph Databases. *The VLDB Journal* 5 (2021).
[8] O. Erling, A. Averbuch, Josep LluXed, S Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, MD Pham, and P. A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. *ACM* (2015).
[9] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. (2021).
[10] N. Francis, A. Taylor, A. Green, P. Guagliardo, and P. Selmer. 2018. Cypher: An Evolving Query Language for Property Graphs. In *International Conference*.
[11] Dongqi Fu and Jingrui He. 2022. DPPIN: A Biological Repository of Dynamic Protein-Protein Interaction Network Data. In *IEEE International Conference on Big Data, Big Data 2022, Osaka, Japan, December 17-20, 2022*. IEEE, 5269–5277. https://doi.org/10.1109/BigData55660.2022.10020904
[12] Shawn Gu, Meng Jiang, Pietro Hiram Guzzi, and Tijana Milenković. 2022. Modeling multi-scale data via a network of networks. 38 (2022), 2544–2553.
[13] Wentao Hant, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *European Conference on Computer Systems*.
[14] Thomas Hartmann, Francois Fouquet, Assaad Moawad, Romain Rouvoy, and Yves Le Traon. 2019. GreyCat: Efficient what-if analytics for data in motion at scale. *Information Systems* 83, JUL. (2019), 101–117.
[15] P. J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* (2005).
[16] Yifan Hou, Hongzhi Chen, Changji Li, James Cheng, and Ming-Chang Yang. 2019. A Representation Learning Framework for Property Graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/3292500.3330948
[17] Zengfeng Huang and Pan Peng. 2019. Dynamic Graph Stream Algorithms in o(n) Space. abs/1605.00089 (2019), 1965–1987.
[18] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2018. Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs. 29 (2018), 659–672.
[19] S Mehran Kazemi, R. Goel, K. Jain, I. Kobyzev, A. Sethi, P. Forsyth, and P. Poupart. 2019. Representation Learning for Dynamic Graphs: A Survey.
[20] U. Khurana and A. Deshpande. 2013. Efficient Snapshot Retrieval over Historical Graph Data. *IEEE* (2013).
[21] Pradeep Kumar and Hao Howie Huang. 2020. GraphOne. *ACM Transactions on Storage (TOS)* (2020).
[22] Jean-Baptiste Lamy and Rosy Tsopra. 2020. RainBio: Proportional Visualization of Large Sets in Biology. 26 (2020), 3285–3298.
[23] Claudio D. G. Linhares, Bruno A. N. Travençolo, Jose Gustavo de Souza Paiva, and Luis E. C. Rocha. 2017. DyNetVis: a system for visualization of dynamic networks.. In *Selected Areas in Cryptography*. 187–194.
[24] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye. 2022. Clock-G: A temporal graph management system with space-efficient storage technique. In

2022 IEEE 38th International Conference on Data Engineering (ICDE). 2263–2276. https://doi.org/10.1109/ICDE53745.2022.00215

[25] Carolina E. S. Mattsson and Frank W. Takes. 2021. Trajectories through temporal networks. *Applied Network Science* 6, 1 (2021), 1–31.

[26] M. Meimaris and G. Papastefanatos. [n. d.]. The EvoGen Benchmark Suite for Evolving RDF Data. ([n. d.]).

[27] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage* 11 (2015).

[28] Othon Michail and Paul G. Spirakis. 2016. Traveling Salesman Problems in Temporal Graphs. *Theoretical Computer Science* 634, C (2016), 1–23.

[29] Paul Murray, Fintan McGee, and Angus Graeme Forbes. 2017. A taxonomy of visualization tasks for the analysis of biological pathway data. *BMC Bioinformatics* 18 (2017), 21:1–21:13.

[30] Arvind Murugan, Kabir Husain, Michael J Rust, Chelsea Hepler, Joseph Bass, Julian M J Pietsch, Peter S Swain, Siddhartha G Jena, Jared E Toettcher, Arup K Chakraborty, Kayla G Sprenger, T Mora, A M Walczak, O Rivoire, Shenshen Wang, Kevin B Wood, Antun Skanata, Edo Kussell, Rama Ranganathan, Hong-Yan Shih, and Nigel Goldenfeld. 2021. Roadmap on biology in time varying environments. *Phys Biol* 18 (2021 May 17 2021). https://doi.org/10.1088/1478-3975/abde8d

[31] Usman Naseem, Imran Razzak, Matloob Khushi, Peter W. Eklund, and Jinman Kim. 2021. COVIDSenti: A Large-Scale Benchmark Twitter Data Set for COVID-19 Sentiment Analysis. 8 (2021), 1003–1015.

[32] F. Petrizzelli, P Hiram Guzzi, and T. Mazza. 2022. Beyond COVID-19 Pandemic: Topology-aware optimisation of vaccination strategy for minimising virus spreading. *arXiv e-prints* (2022).

[33] Christopher Rost, Kevin Gomez, Matthias Taeschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2022. Distributed temporal graph analytics with GRADOOP. *VLDB journal: The international journal of very large data bases* 2 (2022), 31.

[34] P. J. Russell, P. E. Hertz, and B. Mcmillan. 2015. Biology: The Dynamic Science. *Biology the Dynamic Science* (2015).

[35] B. Salzberg and V. J. Tsotras. 1999. Comparison of Access Methods for Time-Evolving Data. *Comput. Surveys* 31, 2 (1999), 158–221.

[36] Robert Soulé and bugra gedik. 2016. RailwayDB: adaptive storage of interaction graphs. 25 (2016), 151.0–169.0.

[37] Benjamin A. Steer, Felix Cuadrado, and Richard G. Clegg. 2017. Raphtory: Decentralised Streaming for Temporal Graphs: Poster. In *the 11th ACM International Conference*.

[38] J. Webber. 2012. A programmatic introduction to Neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*.

[39] Jae wook Ahn, Catherine Plaisant, and Ben Shneiderman. 2014. A task taxonomy for network evolution analysis. 20 (2014), 365–376.

[40] H. Zhuang, Y. Sun, J. Tang, J. Zhang, and X. Sun. 2013. Influence Maximization in Dynamic Social Networks. In *2013 IEEE International Conference on Data Mining (ICDM)*. IEEE Computer Society, Los Alamitos, CA, USA, 1313–1318. https://doi.org/10.1109/ICDM.2013.145