G-View: View Management for Graph Databases

Yunjia Zheng
McGill University, Canada
yunjia.zheng@mail.mcgill.ca

Charlotte Sacré
McGill University, Canada
charlotte.sacre@mail.mcgill.ca

Mohanna Shahrad
McGill University, Canada
mohanna.shahrad@mail.mcgill.ca

Owen Lipchitz
McGill University, Canada
owen.lipchitz@mail.mcgill.ca

Yu Ting Gu
McGill University, Canada
yutingxuegu@gmail.com

Bettina Kemme
McGill University, Canada
bettina.kemme@mcgill.ca

ABSTRACT

Graph database systems (GDBS) have become popular for representing real-world entities and their relationships, and offering convenient query languages based on graph pattern matching. As graphs increase in size and complexity, GDBS need to provide the appropriate support for abstraction for which views have demonstrated to be an effective tool, facilitating query writing and improving query execution time via materialization techniques. This paper explores how views can be defined and used in GDBS. We propose view-based extensions to the widely used graph query language Cypher, explore a wide range of possible view types, and outline several implementation strategies for view materialization. Using a set of micro- and macro-benchmarks, we provide insight into how expressive different view types are and how effective the proposed implementation strategies are for different GDBS. Our results show that views can be a powerful tool for GDBS, offering great flexibility in query expression and providing performance improvements if materialized.

PVLDB Reference Format:
Yunjia Zheng, Charlotte Sacré,
Mohanna Shahrad, Owen Lipchitz, Yu Ting Gu, and Bettina Kemme. G-View: View Management for Graph Databases. PVLDB, 18(6): 1730 - 1742, 2025.
doi:10.14778/3725688.3725702

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DISLMcGill/GDB_Views_VLDB2025.

1 INTRODUCTION

Graph databases are used for a wide range of applications where relationships play a major role such as social networks, recommendation engines or knowledge graphs. The property graph model, where nodes and edges can have properties similar to attributes in relational systems, has become the predominant data model of current graph database systems (GDBS). These systems need to support large deployments with growing volumes of data and increasingly complex networks and queries. The use of *views* is an effective mechanism to address such complexity. While views are a well-established and easily comprehensible concept in relational database systems (RDBS), view management in GDBS has only received limited attention. The approaches proposed so far vary widely and many are often developed for specific use cases. For instance, Han and Ives [21] envision views as transformations on the original graph, Kaskade [15] creates materialized views specifically for query optimization and GraphSurge [32] uses views to study changes in the structural properties of graphs over time.

The goal of this paper is to explore the possibilities for generic view management in GDBS so that views for graphs have the same benefits as views in RDBS: streamline query writing by declaring *frequent sub-queries as views and reuse* them in subsequent queries, offer diverse *abstractions* from the underlying graph in order to hide content that is not relevant or for which users might not have authorized access, and enhance *query performance* by using *materialized views*.

In principle, if a graph query language allows graphs as return value, then view definition and usage seem trivial: a view can be defined as the output graph of a query over which new queries can be posed. In fact, the C-CORE [9] language and the recently developed standard GQL [3] have specific keywords for creating such graph-based views. However, in most query languages used by current GDBS a query does not actually return a graph but a table which represents the binding of variables to elements (nodes/edges) in the graph, or collections of nodes and edges. Examples are programmatic APIs such as Gremlin [4] or various declarative query languages such as Neo4j's Cypher [18], Oracle's PGQL [39], and TigerGraph's GSQL [5]. The recently created standard SQL/PGQ [17] also follows this approach. Not returning a graph contrasts with SQL's inherent closed structure, where queries return tables that can be input for subsequent queries. The absence of such a closed nature for these graph query languages complicates view definitions and their usage.

In this paper, we specifically focus on these languages in order to provide solutions that can be integrated in current GDBS. Our proposed approach, named *G-View*, has the following contributions.

Query Language Extensions. We propose a wide range of view types that can return tables, or sets of nodes, edges and/or paths and also subgraphs of the original graph. Furthermore, we allow usage queries to not only retrieve information from a view but also query views together with the underlying graph similar to how base tables and views together can be input of a SQL query in a relational system. We further discuss how these view types can be used for application development. Our query language extensions, presented in Section 4, are based on Cypher.

Implementation Strategies. We present several implementation alternatives for view management. Our proposals do not create a fully materialized view but create meta-information that is linked to the original graph. The first option uses element identifiers that serve as pointers to the elements in the graph that belong to the view. The others encode the necessary information directly into the base graph as metadata. We also analyze whether and how views can be updated

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725702

incrementally when changes occur. We implemented our solutions as a middleware for fast prototyping. This enabled us to use several database systems and understand how implementation alternatives perform with different execution engines. We implemented our ideas on top of Neo4j [31], a native GDBS, Kuzu [33], that has a relational storage and execution engine specifically designed for graph data, and Apache AGE, a graph-based interface on top of PostgreSQL [1]. Section 5 covers the implementation details.

Evaluation. We gain insights into the usefulness of views for GDBS and performance of our view implementations through a set of micro- and macro-benchmarks. Through micro-benchmarks we analyze what kind of views are beneficial for different usages, and how the different implementation strategies differ for our example GDBS. Additionally, two macro-benchmarks, one based on the LDBC benchmark [37], compare view management across a wide range of different views and usage queries. Section 6 covers our evaluation.

2 BACKGROUND

A **property graph** [9, 10, 13] consists of *nodes* connected via directional or non-directional *edges* (*relationships*). We assume each node/edge element to have exactly one *label* as well as a set of *properties* with a name and value. Over the years, many **graph query languages** have been developed. The focus of this paper is declarative query languages. They have two main components ([4, 5, 18, 30, 39]): they can contain graph patterns and SQL-like constructs (projections, selections, joins over graph patterns, aggregations, etc.). In Cypher, graph patterns are written in the form of `MATCH p_var = (n_var1:N_LABEL1)-[:E_LABEL]->(n_var2)`, where expressions in parentheses and square brackets refer to nodes resp. edges and a path is visualized through the undirected "–" and directed "–>" connectors. During query evaluation, qualifying nodes, edges and paths are then bound to the variables. Many languages also allow pipelining a query result into a follow-up query using a `WITH` clause.

In terms of output, the vast majority of query languages of existing GDBS return variable bindings. This can happen either in the form of independent collections of nodes, edges or paths, or in table format. The output can also be restricted to only some attributes or aggregate values. G-CORE, GQL and Apache Spark's dialect of Cypher [2, 3, 9] can return graphs by using a `CONSTRUCT` return clause. But even then, nodes, edges, and paths of the base graph are often intermediate results that are pipelined into the graph construction.

Popular native **graph database systems** adopting the property graph model include Neo4j [7], JanusGraph [6], AgensGraph [36] and Kuzu [33], among others. Our evaluations are based on Neo4j, Kuzu and PostgreSQL, as they have significantly different architectures.

Neo4j stores nodes, relationships, labels and properties as records in separate files. To facilitate quick lookups, unique IDs corresponding to the offset of each entity within the file are maintained for nodes and edges. Neo4j employs iterative query processing where intermediate results between operators are pipelined without full materialization, known as the "Volcano" model [26]. **Kuzu** uses a Cypher dialect and is a schema-based database where the labels and properties need to be pre-defined for all node and edge types. It has a relational query engine and organizes nodes and edges into different tables based on their labels, with indices on user-defined primary keys. Kuzu excels in using various types of joins for graph pattern matching [8]. Additionally, we use **Apache AGE** [1], a Cypher API on top of **PostgreSQL**. Again, a graph is translated into relations (one for each label). PostgreSQL is a traditional row-based RDBS that has been optimized for general table-based query execution.

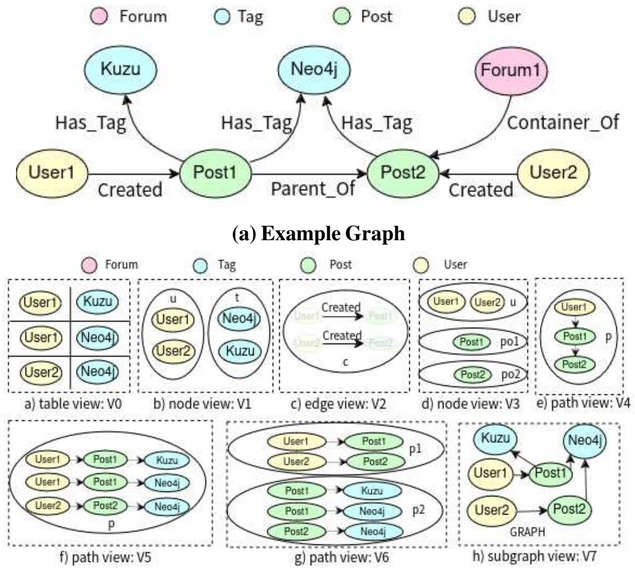
Table 1: Design concern comparison with related work

		G-view	Kaskade [15]	Graphsurge [32]	Han and Yves [21]	Zhuge and Garcia-Molina [40]
View type	Node	✓				✓
	Edge	✓		✓		
	Path	✓				
	Subgraph	✓	✓		✓	
	Table	✓				
	Transform		✓		✓	
Query on view + base graph		✓	X	X	X	✓
Maintenance		✓	X	X	✓	✓
Flexibility		✓	view templates	single WHERE clause	✓[1]	✓
Implementation		middleware	middleware	standalone	middleware	none
language		Cypher	Cypher-SQL hybrid	GVDL	Cypher, Datalog and SQL	OQL

3 RELATED WORK

View management is well understood for relational database systems [19]. When the first semi-structured data models such as XML emerged, view-like components were discussed for indexing and query rewriting [28, 29]. There also exists significant work on views and query rewriting for SPARQL and RDF triplets [22, 24, 25]. When the property graph model, which is significantly different from the previous data models, became popular, a considerable amount of theoretical work analyzed query rewriting. Aspects of views and view maintenance played a significant role [11, 13, 14].

More recently, a set of more applied papers proposed views in particular contexts. Table 1 compares these works with our G-View proposal in terms of the types of views supported, their implementation and query language, whether they support view maintenance, whether users can use views together with the base graph, and other particular restrictions. Kaskade [15] uses views specifically to optimize graph analytic queries. Views are based on query templates that return graphs at different levels of abstraction, e.g., graphs that replace paths with single edges or aggregate a group of nodes/edges into a super-node/super-edge. Given a set of queries, relevant views are enumerated using a rule-based system and then the best views are selected through a cost-based evaluation. The system exploits Neo4j for storage of graphs, views and query execution. However, the views are not directly exposed to users for usage. Graphsurge [32] targets the management of graph view collections, where individual views can represent varying snapshots of versions of data (e.g., over time) and provide efficient mechanisms based on differential computation to execute usage queries across all views in a collection. The view declaration language has some restrictions and views consist of edges. Han and Yves [21] discuss implementation strategies for views over property graphs. Based on GQL, views are graphs that represent transformations on the original graph (e.g., replacing part of the graph with a subgraph at a different level of abstraction) with the expectation that the view contains much of the original graph unchanged. Usage queries are local on the view graph. The work analyzes several materialization strategies and incremental view updates based on Datalog. Their experiments are based on a custom-based benchmark with 6 views and 18 usage queries and inspired our own micro- and macro-benchmark suites.



(a) Example Graph

(b) Contents of different view types

Figure 1: Base graph and content of the views

The last work [40] in our comparison table is actually quite old but the closest to G-View in terms of how they analyze and use views. Just like us, the work explores possible language extensions for view definition and usage, and they allow usage queries to access views together with the base graph, discussing the challenge of connecting view elements back to the underlying graph. However, their rooted graph data model and their query language differ significantly from current GDBS.

4 VIEW DECLARATION AND USAGE

We want a language that supports diverse abstractions from the underlying graph and convenient view usage. Unlike relational systems, information in graph databases is already represented at different levels of abstractions. *Nodes* and *edges* are basic graph elements while a *path*, being an ordered list of nodes and edges, or *sub-graphs* are at a higher level of abstraction. Each of them has its own distinct characteristics and might be of interest for users. For example, referencing specific paths of the base graph might not be straightforward if views are graphs themselves. With this in mind, we propose a language that enables views to represent nodes, edges, paths, subgraphs or a combination of that. Additionally, we allow views to be tables as this is the typical return value of current query languages. Just as in relational systems, we support usage queries on the view only as well as across views and the base graph.

In the remainder of this section, we propose an extension of the Cypher language to allow the declaration and usage of views. We explain its semantics using examples based on the simple social network shown in Figure 1a with nodes having labels *User*, *Post*, *Forum* and *Tag*. Each node has a property name and its value is shown within the node circle. A user may create a post, a post can have several tags and have parent-child relationships with other posts, and a forum contains posts. Some elements have a property *creationDate*.

4.1 View Declaration

The layout and content of the views for the declaration queries of this section over the graph of Figure 1a are shown in Figure 1b.

4.1.1 *Table Views.*

As Cypher queries usually output a table of variable bindings or pipeline these bindings to the next subquery, it appears natural to capture such a table as *Table View*. Table view V0 below returns all three qualifying pairs of (User, Tag) nodes.

```
CREATE VIEW AS V0
MATCH (u:User)-[:Created]->(:Post)-[:Has_Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
RETURN u,t
```

4.1.2 *Node and Edge Views.*

Often, queries focus on a specific subset of nodes (or edges), such as users of a certain city or users that post in a certain forum. Keeping these elements in a view might facilitate many usage queries. Below view V1 returns only nodes (referred to as *Node View*)[2]. Note that while V1 has the same graph pattern as V0, it keeps user and tag nodes as independent sets. In contrast, V0 keeps track of which user/tag pair is linked through the graph pattern and thus, each user resp. tag can appear several times.

V2 returns a set of edges (*edge view*) and in V3 a post node can appear in two of the returned sets.

```
CREATE VIEW AS V1
MATCH (u:User)-[:Created]->(:Post)-[:Has_Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
RETURN COLLECTSET(u), COLLECTSET(t)
```

```
CREATE VIEW AS V2
MATCH (:User)-[c:Created]->(:Post)-[:Has_Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
RETURN COLLECTSET(c)
```

```
CREATE VIEW AS V3
MATCH (u:User)-[:Created]->(po1:Post)-[:Parent_Of]->(po2:Post)
RETURN COLLECTSET(u), COLLECTSET(po1), COLLECTSET(po2)
```

4.1.3 *Path Views.*

Path views keep track of paths in their entirety. As Cypher supports variables for paths, we can define them similar to node and edge views. Path view V4 has a single path User1->Post1->Post2. V5 and V6 have the same pattern overall but V5 returns a single set of paths, while V6 returns two different sets of paths. In fact, using the syntax of V6, concatenation paths via common variables, the query itself can contain a graph pattern that does not represent anymore a linear path. However, the path variables returned refer to a linear path. So far, our implementation does not support variable-length paths. We will see later that view usage queries often refer to specific nodes or edges within a path through variable assignments which is non-trivial for variable length paths.

```
CREATE VIEW AS V4
MATCH p=(:User)-[:Created]->(:Post)-[:Parent_Of]->(:Post)
RETURN COLLECTSET(p)
```

```
CREATE VIEW AS V5
MATCH p=(:User)-[:Created]->(:Post)-[:Has_tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
RETURN COLLECTSET(p)
```

```
CREATE VIEW AS V6
MATCH p1=(:User)-[:Created]->(po:Post),p2=(po)-[:Has_tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
RETURN COLLECTSET(p1), COLLECTSET(p2)
```

[2] We introduce COLLECTSET for referring to collections without duplicates.

4.1.4 *Subgraph Views.*

As most of the work on graph-based views assume views to be graphs, we also support subgraph views. They use the keyword `CONSTRUCT` as shown for view V7 below. A subgraph view is made up of all nodes and edges that are bound to any variable in the `RETURN` clause or that are part of any path that is returned.

```
CREATE VIEW AS V7
MATCH p=(:User)-[:Created]->(:Post)-[:Has-Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
CONSTRUCT p RETURN GRAPH
```

4.1.5 *Additional features and restrictions.*

As node/edge/path views return sets of entities, table/subgraph views do not. Therefore, a view declaration query can return a mix of node, edge and path sets but this cannot be combined with returning a table or subgraph. Furthermore, we do not allow view declaration queries to return attributes or aggregations as we did not find an intuitive way to reconnect this with the rest of the graph in view usage queries. Unlike other approaches, we so far do not allow a view to have nodes or edges that do not exist in the underlying graph, i.e., our subgraph views are always a true subgraph of the original graph.

4.2 View Usage

A view usage query has *local scope* if it only references view data, and *global scope* if it refers to both view(s) and the underlying graph.

4.2.1 *Local Queries:*

Local queries need to use the keyword `LOCAL`.

Using Node and Edge Views, local usage queries typically return the entire view or include conditions on the nodes/edges. Usage query UQ1 below returns all nodes of view V1, while UQ2 only returns the user nodes. By referring to the variable names that were used during view declaration, the usage query can zoom in on specific sets of nodes/edges. UQ3 retrieves the edges of edge view V2 that fulfill a restriction on one of the properties. Its MATCH clause has to specify the end nodes of the edge in order to follow Cypher syntax. Finally, as shown with UQ4, a usage query can refer to several views. In this case, we need to associate the variables used within the usage query with the appropriate views.

```
UQ1: WITH VIEWS V1 LOCAL MATCH (n) RETURN n

UQ2: WITH VIEWS V1 LOCAL MATCH (n)
     WHERE n in V1.u RETURN n

UQ3: WITH VIEWS V2 LOCAL MATCH ()-[r]->()
     WHERE r.creationDate < 1313000000000 RETURN r

UQ4: WITH VIEWS V1 V3 LOCAL MATCH (n)
     WHERE n in V1.u AND n in V3.u RETURN n
```

Using Path Views, we can return entire paths (that might need to fulfill a certain condition), or individual nodes or edges from the paths. We use the extension `p=*` to refer to all paths in the path view or paths bound to one of the path variables as shown in UQ5 and UQ6. UQ7 returns all the nodes that appear in the path. UQ8 retrieves only the user nodes found in one of the paths of V5 with an additional condition on tag. In this case, the `MATCH` clause has to include the full path pattern of the view declaration query, and specify variable names for the elements of interest. That is, we cannot omit the path pattern and only write `WITH VIEWS V5 LOCAL WHERE t.name = "Neo4j" RETURN u`, because when we create the view we do not keep track of any variable names of elements within the path.

Currently, it is not possible to retrieve a sub-path within a path of a view. Extending the language syntax to allow pattern matching on paths within a view would be straightforward. But this would mean we treat each of the paths as a mini-graph. Recall that the view contains sets of paths and there could be many such paths (in the hundreds and thousands). Performing the pattern matching on each of them can be very expensive. Thus, the last query UQ9 below will be empty as all the paths in V4 have a parentship of posts.

```
UQ5: WITH VIEWS V4 LOCAL MATCH p=* RETURN p

UQ6: WITH VIEWS V6 LOCAL MATCH p=*
     WHERE p in V6.p1 RETURN p

UQ7: WITH VIEWS V4 LOCAL MATCH (n) RETURN n

UQ8: WITH VIEWS V5 LOCAL MATCH p=(u:User)-[]->()-[]->(t)
     WHERE t.name = "Neo4j" RETURN u,t

UQ9: WITH VIEWS V4 LOCAL
     MATCH p=(u:User)-[cr:Created]->(po:Post) RETURN p
```

Using Subgraph Views We do not support a view usage query that returns the full subgraph captured in a subgraph view as Cypher does not support graphs as return value. Instead we can query that subgraph as if it were a standard graph. For instance, below query UQ10 returns all nodes in V7. UQ11 has the same pattern as the empty usage query UQ9 on paths above, but this time, it will return the sub-paths found because subgraph views manage nodes and edges as a graph, allowing for new graph patterns in the query.

While this might be advantageous in some situations, if one wants to find the exact paths defined in the view declaration query, path views can be more convenient (they allow for the p* expression) and possibly faster because with a subgraph view the query engine has to again execute the path pattern.

```
UQ10: WITH VIEWS V7 LOCAL MATCH (n) RETURN n

UQ11: WITH VIEWS V7 LOCAL
      MATCH p=(u:User)-[cr:Created]->(po:Post) RETURN p
```

Using Table Views A usage query must refer to the columns of the table view. To do so, we use the variable names used during view declaration. In query UQ12 below the keyword `rec` specifies that `u1` and `t1` are aliases to the `u` and `t` columns of V0.

```
UQ12: WITH VIEWS V0 LOCAL MATCH (u1:User),(t1:Tag)
      WHERE rec(u1,t1) IN V0.(u,t) AND t1.name = "Neo4j"
      RETURN u1,t1
```

Note that this usage query provides the same result as the path usage query UQ8. However, V0, in contrast to V5, cannot be used to retrieve any information regarding posts because posts are not in V0's result. But if we let V0 return all nodes and edges along the path, then it becomes as expressive as V5. Furthermore, if the declaration query contains several path patterns, then a table view can contain elements from the different paths. Thus, comparing table views with path views, table views might be more compact if we know that we do not need all elements anymore in later usage queries. Furthermore, usage might be easier as `rec()` only needs to refer to the elements of the table that are of interest for the query while the path query needs to write out the entire path pattern.

We would like to note that the same result as UQ8 resp. UQ12 cannot be obtained by using node view V1 as in the node view the users and tags are no more associated with each other. Thus, if the relationships expressed in a view declaration query are relevant in later queries, node views might not be appropriate.

4.2.2 Global Queries: Global queries are used when information from the base graph is also needed. A view usage query is like a standard query on the original graph but will have some nodes/edges/-paths/tuples matched in the query to be part of a view. Any variable of the query that is not associated with a view is matched purely on the base graph. UQ13 below looks for forums that were created before a certain date and that contain the selected child posts from view V3. Only the posts refer to a view, therefore the Container_Of edges and Forum nodes pertain to the base graph.

```
UQ13: WITH VIEWS V3 GLOBAL
      MATCH (f:Forum)-[:Container_Of]->(po:Post)
      WHERE po IN V3.po2 AND f.creationDate<X RETURN f
```

If we want to retrieve similar information using path view V4, we cannot simply take UQ13 and replace node view V3 with path view V4, because we did not keep track of the variable name po2 during view creation time as already discussed for local views. Instead, we do have to repeat the entire path pattern of V4 as depicted in UQ14 below in order to bind the child post to the variable po2. In contrast, if we want to do the match with any post in the path view, then we can use query UQ15 below without specifying the pattern of the path view as the path view keeps track of all nodes.

```
UQ14: WITH VIEWS V4 GLOBAL
      MATCH p=()-[]->()-[]->(po2),(f:Forum)-[]->(po2)
      WHERE p IN V4 AND f.creationDate<X RETURN f

UQ15: WITH VIEWS V4 GLOBAL
      MATCH (f:Forum)-[:Container_Of]->(po)
      WHERE po IN V4 AND f.creationDate<X RETURN f
```

We can get the same information using a sub-graph view by replacing V4 with V7 in queries UQ14 and UQ15. In the first case, it will give us the forums of the child posts while the latter also returns the forums of parent nodes of the view.

Generally, usage queries need to be global when the search goes beyond the data returned by the view declaration queries. They must be global even if the patterns in the view declaration query captured all the information but it was not returned. For instance, as table view V0 only returned users and tags but not the posts, UQ16 below must be global because it refers to posts.

```
UQ16: WITH VIEWS V0 GLOBAL
      MATCH (u1:User)-[]->(po:Post)-[]->(t1:Tag)
      WHERE rec(u1,t1) IN V0.(u,t) RETURN po
```

4.3 Discussion

4.3.1 Language Expressiveness. Our proposal represents a fairly straightforward extension of the Cypher language, and the semantics for pattern matching and pipelining remain the same while offering a wide range of options to declare and use views.

View types Table views return what Cypher queries typically return while our other view types contain sub-components of the original graph at different levels of abstraction. Table and path views capture the relationships as expressed in the graph pattern of the declaration query, while the other view types are set based.

View Scope View usage queries with LOCAL scope can only access the elements that are contained in the view while GLOBAL usage queries also access the base graph.

Searching within views Our language proposal enables searching within views at different levels of granularity. For instance, we can consider all nodes in the view or restrict the search to a more specific "location" with the "n in V.varname" notation. We expect table views to typically be always used with specific matching of columns.

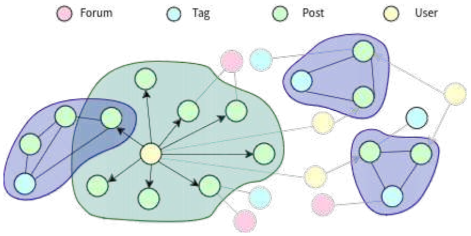


Figure 2: Universal and seeded queries in real-world scenarios

4.3.2 Use Cases. In order to illustrate the potential of views, we want to outline some view scenarios for our social network example of Figure 1a. We observe two main forms of graph search queries in the literature, illustrated in Figure 2. First, *universal queries* involve general graph patterns, often with conditions on properties, that target recurring patterns across the graph, e.g., parent-child posts with a common tag (purple in Figure 2). We can express this as a view with pattern MATCH p1=(po1:Post)->(po2:Post), p2=(po1)->(t:Tag), p3=(po2)->(t) that returns a table with all node variables, all paths or everything as a subgraph. This view can serve as input for and simplify a wide range of usage queries, e.g., understanding which tags trigger conversations (count posts in the view with a given tag name), or performing sentiment analysis on a tag basis (check the posts around a topic that are supportive vs. hateful).

Another scenario is for access control. For instance, in order to allow moderators to check for offensive content, one can create a node view with all posts and allow moderators to only pose local queries on this node view – hiding sensitive information about users.

A second type of graph search queries are *seeded queries* which start the search from one specific node (usually with a high degree of edges and referred to as a *hub*), such as a famous person. This is shown in Figure 2 in green. Extracting popular nodes and their neighbors as views can be useful for simplifying and speeding up many different queries as such popular nodes are often of specific interest. Hub presence and degree centrality are crucial in airline scheduling [16], disease propagation [23] and user engagement analysis [27]. Our simple example in Figure 2 could represent all posts of user 'Mads Mikkelsen'. Then queries interested in posts of this celebrity can use this view locally while users interested in who responds to such posts could use a global usage query. In fact, the LDBC benchmark [37, 38] in its interactive complex query set has many queries that start by searching for people connected to or acquainted with a specific person.

5 IMPLEMENTATION STRATEGIES

Using materialized views, a view usage query runs over the materialized result of the view declaration query. With non-materialized views, the database system rewrites the view usage query and combines it with the view declaration query to run on the original data. What we present is a hybrid, where a view declaration query is executed resulting in materialized "meta-information". A view usage query is then rewritten based on the meta-information and runs on the original base graph. The goal is that the rewritten view usage query executes faster

Table 2: Identifier Tables

View	Table Type	Set
V1	Node Table	{(u,{ID(User1),ID(User2)}), (t,{ID(Neo4j),ID(Kuzu)})}
V4	Path Table	{(p,{[ID(User1,Post1),ID(Post1,Post2)]})}
	Node Table	{ID(User1),ID(Post1),ID(Post2)}
	Edge Table	{ID(User1,Post1),ID(Post1,Post2)}

than if the user would have to write a baseline query that combines the semantics of the usage and declaration queries.

We present two approaches for view management. In the ID-based approach, we keep track of the IDs of the elements that constitute the view. In the property-based approach the nodes and edges of the original base graph are extended with special properties that capture view containment. All our implementation is in a middleware layer – we do not change the underlying database software. Instead, we use the standard Cypher-based APIs of the GDBS to execute view declaration and rewritten usage queries. Therefore, the two approaches are fully portable to various systems. As the different GDBS we used have interface variations, we had to adjust our solution slightly for each system supported.

5.1 ID-based strategy

Identifiers are widely used in GDBS to identify and reference nodes, edges or even properties. Therefore, we propose a strategy to use identifiers as an index for the result set of a view. The middleware keeps track of all necessary meta information about the views – leaving the original graph untouched[3].

5.1.1 View Declaration. Upon view declaration, the query is first executed. For node/edge/path views, the identifiers of all returned nodes resp. edges are then stored in a node resp. edge table. Were they assigned to (a) variable(s), we also keep track of this. Additionally, for path views the middleware keeps a path table where each qualifying path is recorded as a list of edge identifiers sorted by their appearance along the path. Subgraph views only maintain one node table and one edge table for all the node and edge identifiers within the subgraph, respectively. Table views contain a record table that has similar format as the table returned by the original view declaration query, but only with the identifiers instead of the complete elements. As examples, Table 2 shows the meta-information the middleware maintains for views V1 and V4. V1 maintains the identifiers of the two user and two tag nodes and the variables u resp. t to which they were assigned during declaration. V4 has a single path identified by its edges and keeps also track of the individual nodes/edges in extra tables.

Note that declaration queries are rewritten so that they retrieve only the identifiers and not the full elements with properties whenever possible. Otherwise, identifier extraction is done in the middleware.

5.1.2 View Usage. A usage query is extended so that the search within the graph is restricted to the components that are part of the view. Consider a scenario when we have the following global view use query, trying to expand from paths in V4 to tags in V1 as follows:

```
UQ19: WITH VIEWS V4 V1 GLOBAL
      MATCH p=()-[]->()-[]->(po2), (po2)-[]->(t)
      WHERE p IN V4 AND t In V1 AND <other condition>
      RETURN po2
```

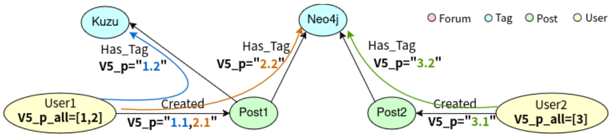


Figure 3: Property-based implementation for path view V5

The middleware rewrites the usage query including conditions, ensuring that only elements that are part of the corresponding views are considered during the execution as shown below in Neo4j's Cypher notation. For checking path membership, we transform the list of path sequences of the path table into a list of arrays (named edges) and use the UNWIND operator to unfold the list (line 4) ensuring that each path is matched individually. As UQ19 has not defined variable names for the edges in the path but we need them in order to check containment, our rewriting algorithm adds them as needed. For checking membership of nodes/edges, a search on the identifiers is conducted (line 5) by providing the identifiers in form of a list. As UQ19 asks for "t in V1" all node identifiers are in the search list. If the usage query had "t in V1.t", then the search would be restricted to the tag nodes.

```
1 UNWIND $edges AS edges
2 MATCH ()-[e1]-()-[e2]-(po2),
3       (po2)-[]->(tag)
4 WHERE ID(e1) = edges[0] AND ID(e2) = edges[1]
5 AND ID(tag) IN [ID(User1),ID(User2),ID(Neo4j),ID(Kuzu)]
6 AND <other condition> RETURN po2
```

When using table views, the records in a table view are unwrapped using the UNWIND operator similar to path views. Each row in the table is considered individually when looking at membership matches for the columns in the table. For subgraph views, we check the membership for each node and edge in the usage query.

5.2 Property-based strategy

The second implementation approach augments the graph with additional meta-information to indicate which elements of the graph are part of which views. Usage queries are then rewritten to check for the appropriate meta-information. The idea is to attach special properties to nodes and edges to identify view inclusion. There are several possible implementation alternatives and we present below the one that was most efficient within Neo4j. At the end of this section, we discuss alternative strategies that we have explored.

5.2.1 View Declaration. We attach the information for a view named Vname to the graph in the following way.

Node/edge sets: Assume Vname returns COLLECTSET(u) where u is a node resp. edge variable u. Then each node resp. edge in this set will get a property Vname which is a list, and one of the list values is u. For instance, for V1 each qualifying node has either property V1=[u] or V1=[t]. In contrast, for V3, a post could be both parent and child. In this case, its property would be V4=[po1,po2].

Subgraph views If Vname is a subgraph view, then all nodes and edges in this subgraph view get a property Vname. Its value has no importance as we do not keep track of the the variables from which the subgraph was constructed.

[3] While our current implementation does not provide fault-tolerance this could be easily added by forcing the middleware to persist the information it maintains.

Table 3: View Properties for table view V0

Name	Properties
User1	`V0_r="1.u,2.u";V0_all=[1,2]`
User2	`V0_r="3.u";     V0_all=[3]`
Kuzu	`V0_r="1.t"`
Neo4j	`V0_r="2.t,3.t"`

Path sets If Vname returns COLLECTSET(p) of path variable p and let $\{p_1,p_2,...\}$ be the set of paths returned, then we give each path within p a monotonically increasing *sequence number* s, and within a path, each edge receives additionally a *rank* r indicating its position within the path. From there, we create properties as follows.

- If an edge is involved in at least one of the paths bound to p, it gets a string-based property Vname_p. If the edge has rank r in the path with sequence number s, then Vname_p contains the *path-rank* identifier s.r. We implement Vname_p as a string because Neo4j provides efficient sub-string operators.
- If a node is the first node in any path returned by p, it gets a property Vname_p_all. The value is the list of the sequence numbers of the paths for which it is the first node. This will allow us to quickly find the start points of all paths without doing a string comparison.
- Each node/edge that is part of any path returned by p gets a property Vname. Its value has no importance. It allows us to quickly find all elements of the view (similar to the node and edge tables for path views in the ID-based implementation).

Figure 3 shows the view properties of all nodes and edges involved in path view V5 omitting Vname. There is one path variable p and three valid paths. User1 is start node of the first two paths and User2 of the third path. The Create edge between User1 and Post1 is the first edge of the first two paths and the Has_Tag edge to Kuzu is the second edge of the first path.

Table views Table views are handled similar to path views. If Vname returns a table with variables $\{v_1,v2,...\}$ bound to nodes or edges, then each row in the table gets a monotonically increasing sequence number, and $j.v_i$ is the *row-variable* identifier for v_i in row j.

- Each node/edge in the view gets a property Vname_r listing the row-variable identifiers to which it is bound.
- Each element that is bound to the first variable v_1 at least once gets a property Vname_all with the value being the list of row numbers for which this element is bound to v_1.

Table 3 shows the view properties of all nodes of table view V0. There are three rows in the table and User1 is the first element for the first two rows, and User2 for the third.

5.2.2 *View Usage.*

Below shows the rewrite of the global usage query UQ19 of the previous section. We reconstruct each of the paths in V4, starting with their first nodes, and adding variable names for edges as needed. We then check whether the relevant tag is in V1. Similar to the ID-based strategy, if UQ19 had specified "tag In V3.t", then the condition would be rewritten as "tag.V3 CONTAINS t".

```
MATCH (user) WHERE user.V4_p_all IS NOT NULL
WITH user UNWIND user.V4_p_all AS seqNum
MATCH p=()-[e1]->()-[e2]->(po2),(po2)-[]->(tag)
WHERE e1.V4_p CONTAINS seqNum+".1"
AND e2.V4_p CONTAINS seqNum+".2"
AND tag.V1 IS NOT NULL AND <other condition>
RETURN po2
```

For the usage of table views we use similar unwind strategies for each row and then extract the elements in the individual columns.

5.2.3 *Implementation Alternatives.*

Adding additional properties to existing graph elements is only one option. Alternatively, view management information can be stored in new nodes and edges. For example, one could create a special view node that has as label the view name and then connect the view node with all relevant nodes of the view through edges. Additional information can then be encoded as properties of these edges (e.g., indicating that the connected element is in a certain position of a returned path). However, we believe that this would create many new edges in the system. Also, it is not clear how to connect edges in the view to the view node.

Another option could be to create copies of qualifying elements for view management purposes. This would probably resemble the most the materialized views in a relational database system. However, in order to support global queries, one has to relate these copies back to original elements, which can quickly become complex. Nevertheless, we did explore this approach partially for path views. Here, for each edge of each qualifying path of view Vname, the edge is mapped to a new edge between the same endpoints with a Vname label. Sequence numbers and ranks are then encoded as properties of the new instead of the original edge. This approach has shown benefits in Kuzu and for Apache AGE because these GDBS have a table for all edges with the same label. A compact edge table containing only the edges relevant for a usage query can be significantly faster than looking for property values in the potentially large original edge tables.

All these strategies change the base graph. As such, standard queries on the base graph have to be rewritten so that none of the meta-information is returned.

5.3 View maintenance

When the underlying graph changes, views might be affected. In this section, we look at the insertion/deletion of individual elements, i.e., nodes or edges. If several elements are inserted or deleted within a query, below actions are performed for each of these element.

We first determine the views that are guaranteed to *not be affected by the modification*. This can be done if the view declaration query has no negation and only contains labeled elements as is the case for all views in Section 4. Then, if these labels are different from the label of the inserted/deleted element then this element cannot affect the view's result, and no further action is required. For instance any insert/delete of forum nodes or has_container edges will not affect any of the views in Section 4.

For all other views, they are either *updatable* and then we can perform incremental view maintenance or we invalidate and fully reevaluate them when a graph update occurs.

For updatable views, we execute a "seeded" delta query. This is the view declaration query with an additional condition that ensures that a row/node/edge/path entity is part of the result only because of the existence of the inserted/deleted element. More precisely, the additional condition requires that the inserted/deleted element appears at least once in the graph pattern of the query. The result of the delta query is then added to resp. deleted from the view. If executing the seeded delta query is faster than the original view declaration query, then incremental view update is faster than reevaluation.

So far our delta queries are only possible for a restricted set of views where we can add/remove delta results. In particular we support view declaration queries where the MATCH clause only contains fixed-length path patterns, and if there are several patterns, they have common variables so that they represent one connected graph pattern. For instance, V6 defines two paths with common variable po, and thus, in every assignment of elements to the variables, the two paths together build a connected graph. We do not support path patterns without common variables as they lead to a Cartesian product in Neo4j. We also currently do not support aggregation or existence functions in the query nor pipelines using WITH or UNWIND. On the other hand, we support directional and non-directional edges, unlabeled elements and conditions on properties (including numeric functions). All views in Section 4 and a vast majority of views in Section 6 are updatable.

In the following we describe our maintenance algorithm for the ID-based implementation. When a declaration query is executed, an entity (node/edge/path) can be bound to a variable multiple times (e.g., a node can be part of several matching paths). As such, when we delete an edge, we cannot simply delete all entities that are returned by the delta query from the view because they might remain in the view. In order to handle this we keep counters. First, when we execute the view declaration query we keep track of the number of times each entity is returned by adding appropriate counters for each entity in the node, path and edge tables. For instance, in Neo4j, for V1 we use COLLECT(u) to retrieve the users, which returns User1 twice for our example graph as two paths match the path pattern, and we keep track of this in the node table. From there, we look at insertions and deletions of nodes, and then of edges.

Insert or delete a node When a node is added or deleted it cannot have any edges. Thus, if a view has a path pattern with edges, the delta query is guaranteed to return an empty result. Therefore, the only meaningful views to check are those with a single node pattern that return a single node variable, possibly with property conditions on that node and/or a label condition (e.g., MATCH (n) WHERE n.location=... RETURN n). In this case, the delta query is simply the view declaration query with the extra condition that n must be the inserted/deleted node. If the delta query returns the node, it is added resp. removed from the view.

Add an edge Adding an edge has no influence on views that match on nodes only. But if the query contains a graph pattern with at least one edge, the query could now match on new paths that contain this edge which might require adding new entities to the view. Therefore, we execute the declaration query with the additional condition that at least one of the edges in the graph pattern must be the inserted edge. This might require a slight adjustment of the declaration query in order to give variable names to all relevant edges in the query. We can ignore edges with a label different to the label of the inserted edge. As an example, the following delta query is executed for V1 upon inserting an edge e with identifier ID(e) and label Has_Tag.

```
MATCH (u:User)-[:Created]->(:Post)-[e1:Has_Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
AND ID(e1) = ID(e)
RETURN COLLECTSET(u),COLLECTSET(t)
```

We do not perform the identifier comparison on the Created edge because it has a different label, and thus, would never match. If there are edges with no labels (e.g., if the :Created condition is missing in V1), then we would have to check this edge, too, as below:

```
MATCH (u:User)-[e1]->(:Post)-[e2:Has_Tag]->(t:Tag)
WHERE t.name = "Neo4j" OR t.name = "Kuzu"
AND (ID(e1) = ID(e) OR ID(e2) = ID(e))
RETURN COLLECTSET(u),COLLECTSET(t)
```

This query matches on paths that did not exist before the edge was created. In case a table is returned like in V0, we simply add every matching variable assignment as row to the view table. If nodes are returned as in the above example and a node already exists in the view, its counter is increased according to how often this node was in a match. If it does not yet exist, we insert it and set the counter accordingly. Similar holds for edges and paths. Note that even returned paths can already exist in the view. For instance, when we add an edge from Post2 to Kuzu in our example graph, the returned path User2->Post2 already exists in P1 of V6. We also keep the counters for the nodes and edges of path and subgraph views as there can be several paths that allow them to be part of the view.

Delete an edge We run a similar delta query before edge deletion. In case of a table view, we simply remove for each record in the result of the delta query one record in the record table with the same element identifiers. For node, edge and path views, we decrease the counters of the matching entities within the view (removing the entity if the counter is 0), again according to the number of times the entities appear in the return of the delta query. For instance, for node view V1, the deletion of the edge from Post1 to Kuzu causes the counter of User1 to decrease from 2 to 1.

Our approach has some similarities with incremental view maintenance in relational systems [12, 20] which relies on keeping track of cardinalities of entities, and deltas for individual tables in the view declaration query. Having one connected fixed-length graph pattern somewhat resembles SQL queries with simple joins where we can count the number of matches.

6 DEPLOYMENT AND SYSTEM EVALUATION

This section illustrates view behavior through a set of micro- and macro-benchmarks. We first do a detailed analysis using Neo4j only, and then perform further analysis with Kuzu and PostgreSQL as well as compare with a recent related view management system [21].

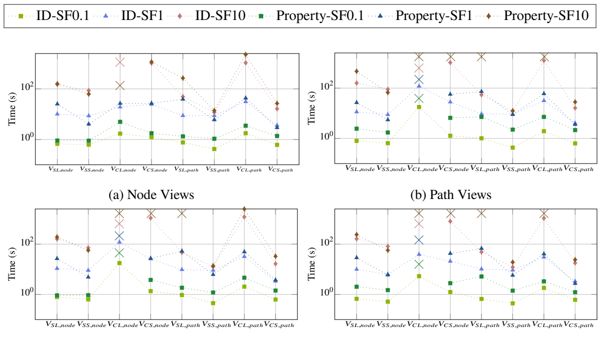
6.1 Experimental setup

Experiments used a machine with an Intel Xeon E3-1220 V5 processor, 32 GB of RAM, using Neo4j v.5.2.0, Kuzu v.22.1, PostgreSQL 16.4 and Java v.19.0.2. The results show the average of 5 runs within a warmed-up system. Note that I/O overhead was not a major factor.

Datasets Most experiments use the LDBC benchmark data [37, 38] at scale factors (SF) 0.1, 1 and 10. The SF1 graph has 3.18M nodes and 17.2M edges. In the following, if not explicitly specified, experiments run on the SF0.1 dataset. Some further experiments were done with StackOverflow data [34] containing 5.6M nodes and 9M edges. These graphs represent characteristics from real-world social networks. In general, the LDBC graphs are denser.

Workload Description Our query workloads consider both universal and seeded queries as described in Section 4.3.2.

Our **micro-benchmarks** consist of universal queries over the LDBC dataset and aim to understand when views are beneficial, and the differences in view types, and implementations. We create view categories along three dimensions. First, a declaration query can have either a *simple (S)* one-edge or a *complex (C)* three-edge graph matching pattern. Second, for node and table views, only some of



(a) Node Views

(b) Path Views

(c) Subgraph Views

(d) Table Views

Figure 4: Declaration time for various SF (in seconds)

Table 4: Declaration time for SF0.1 (in seconds)

	ID-based				Baseline	Property-based			
	Node	Path	Subgraph	Table		Node	Path	Subgraph	Table
V_{SS_node}	0.590	0.642	0.623	0.616	0.594	1.143	1.686	1.19	1.072
V_{SL_node}	0.697	0.748	0.730	0.718	0.677	1.473	2.498	2.158	1.627
V_{CS_node}	1.331	1.396	1.373	1.312	1.239	2.117	-	9.321	-
V_{CL_node}	4.978	17.807	17.679	5.105	2.149	66.398	-	142.919	-

Table 5: Usage time (in seconds) retrieving nodes

	ID-based				Baseline	Property-based			
	Node	Path	Subgraph	Table		Node	Path	Subgraph	Table
V_{SS_node}	0.054	0.083	0.086	0.056	0.594	0.180	0.192	0.183	0.188
V_{SL_node}	0.116	0.156	0.153	0.119	0.667	0.259	0.263	0.261	0.256
V_{CS_node}	0.051	0.128	0.130	0.050	1.239	0.187	-	0.191	-
V_{CL_node}	0.094	0.217	0.227	0.095	2.149	0.264	-	0.262	-

the nodes in the path pattern (e.g., only posts) or all the nodes are returned (notations of *_node* resp. *_path*). Third, we create views with *small (S)* or *large (L)* result sizes by having conditions on nodes with varying selectivity. Combining these three dimensions (complexity, returned variables and size) results in 8 different categories. For each we create 4 different view types (node, path, subgraph and table views) for a total of 32 view versions. We omit edge views as they are similar to node views. There are between 150 and 7000 different nodes and between 700 and 44,000 different paths in the result.

The first **macro-benchmark** is based on a subset of queries from the LDBC benchmark that are characterised as interactive short (IS), interactive complex (IC) and business intelligence (BI) queries, all *seeded queries*. We rewrote them into semantically equivalent queries using views. We create two view variations for each IC/BI query. A *declaration-heavy* view covers nearly the entire benchmark query. Thus, a usage query is often local performing minor tasks (e.g., extracting attributes). A *usage-heavy* view contains only one part of the benchmark query and the usage query is thus likely global. The more general usage-heavy view can then be potentially exploited for various usage queries. Finally, we also created macro-benchmarks with universal queries on the LDBC and StackOverflow benchmarks.

All benchmark queries can be found in our code repository.

6.2 View Declaration

In this section we evaluate the declaration times for our different views in Neo4j using both the ID and the property-based implementation approach. Figure 4 shows the declaration times for all 32 views in our micro-benchmark for all three scale factors (0.1, 1, 10). Table 4 zooms in on the declaration times for SF0.1 and *_node* views for better readability, as in most cases relative execution times at larger scales and for *_path* views are similar. The table also has the execution time for a *baseline query* that contains the same core query (graph pattern and conditions) and returns all corresponding nodes. Overall, we view it as acceptable that a declaration query be more expensive than the baseline query as the overhead occurs only once.

Property-based Approach Property-based is generally slower than ID-based because persisting the view information within the GDBS has more overhead than materializing IDs in the middleware. Also, execution times are considerably larger than for the baseline. For small views, node views are the fastest. Table views update the same nodes but this is more complex because a node can occur in many rows of the table. Subgraph and path views add properties to both nodes and edges, but this is more complex for path views similar to what we observe in table views. For large views, all scale factors had memory issues leading to declaration failure ('-'/'x' in the table/figure). The reason is a heap-size problem within the database engine to keep track of all the paths/table rows.

ID-based Approach For most view versions, declaration is only slightly longer than the baseline query because the actual query takes most of the time and view management overhead is quite small. Only the declaration for V_{CL_node} takes significantly longer. For path and subgraph views all nodes and edges are returned with all their properties (because we use `CONSTRUCT path RETURN GRAPH`), leading to a lot of data transfer and processing. Node views and table views only retrieve ids and are thus much faster, but they are still more expensive than the baseline because node views perform duplicate elimination and table views create large record tables.

There are views in the ID-based approach that time out at larger SFs, as seen, e.g., for V_{CL_node} in Figure 4. This happens at the time results are transferred and processed at the middleware.

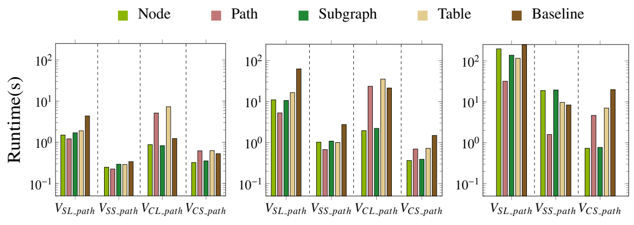
6.3 Usage Behavior

Views are often defined to support frequent queries and the question arises which views are useful for what kind of queries. If we only want to return nodes then local usage queries on node views will be enough, but path, subgraph and table views might also be used although they might be more expensive. When a query is complex it might still be able to use a node view but the usage would become global while it might remain local for the other view types. In here, we explore these different aspects, and compare again with baseline queries that retrieve the same information without using views.

6.3.1 Retrieving nodes. Table 5 shows the runtime for finding a subset of nodes within _node views, e.g., all comments in V_{CS_node}[4].

Baseline vs Views View usage is significantly faster than the baseline. The property-based approach is at least 2x faster for simple views and 10x faster for complex views. The ID-based approach is even better and between 4x to 20x faster than the baseline. Benefits

[4] The results are for SF 0.1. Higher scale factors show similar results.



(a) SF0.1

(b) SF1

(c) SF10

Figure 5: Runtime retrieving path

are higher for complex views as the usage queries do not need to execute the complex graph pattern. Additionally, if the return set is small, the usage query only has to retrieve few nodes from the base graph. The ID-based approach is between 1.2 and 3.6x better than the property-based because Neo4j provides very fast lookup of nodes by identifier. Although we created indices over the view properties for the property-based approach, their lookup was less efficient.

View types For the ID-based approach and a given category, path and subgraph views are slower than node and table views because their node tables are larger (containing node information of all nodes in the path), leading to longer lookup times. This is not true for the property-based approach where the usage queries are nearly the same within a given category, independently of the view type.

View size Executions take longer with larger views and the difference is more pronounced for the ID-based approach. There, every qualifying identifier in the node table causes an extra lookup. In contrast, for the property-based approach, the search time might be more influenced by the number of nodes that need to be searched instead of the nodes in the view. This effect is not visible in our experiments as our queries look for `Post` or `Comment` nodes, which have roughly the same cardinality.

6.3.2 Retrieving paths. We expect path views to be the most useful when a query wants to retrieve exactly the paths stored in the view but other view types can also be used. In case of node views, we need a global usage query that looks for the graph pattern in the base graph and requires all nodes to be part of the node view. As our table views only contain the nodes, we require a global query to also retrieve the edges for each row in the table. We use the views from the `_path` category where node and table views maintain all the nodes that are part of the graph pattern. Figure 5 illustrates the runtime across all scale factors. We only analyze the ID-based approach as it has shown so much better performance in our previous experiments. As the queries using V_{CL} as well as the corresponding baseline query took very long for SF10, we omit them from the figure.

The behavior varies considerably between view categories. Surprisingly, *path views*, while being the best option for the simple view category, behave worse than other view types for the complex one. Cypher's `UNWIND` operator, that retrieves the paths from the base graph given their edge identifiers, turns out to be expensive. Furthermore, Neo4j's engine first looks for all edges of a given source node and then checks edge identifiers as a condition. It would be faster if Neo4j did first the lookup of only the relevant edges. Still, using path views is faster than the baseline query most of the times, and in particular for large graphs as they facilitate a more focused search. *Table views* also use `UNWIND` for each row in the view. In our case, they perform worse than the path views as they maintain more node variables than the path has edges, leading to more lookups. Generally, *node and subgraph views* behave similarly. The rewritten usage query restricts the retrieval from the base graph by requiring the nodes (and also the edges for subgraph views), to be in the view. In fact, this reduced search space allows them to be always faster than the baseline query and the path views for complex views.

Table 6: LDBC benchmark queries (in seconds)

	Declaration heavy			Baseline	Usage Heavy		
	Type	Creation	Usage		Type	Creation	Usage
IC2	GRAPH	0.734	0.826	0.806	NODE	0.215	0.689
IC5	PATH	3.399	0.389	1.848	NODE	0.374	1.597
IC6	TABLE	3.022	0.478	1.225	NODE	0.962	0.354
BI5	GRAPH	0.766	0.477	0.864	PATH	0.512	0.466
BI17	TABLE	12.423	0.329	6.244	TABLE	0.749	86.614

	Type	Creation	Usage	Baseline
IS1	NODE	0.296	0.219	0.310
IS2	NODE	0.324	0.209	0.513
IS3	PATH	0.309	0.315	0.590
IS4	NODE	0.179	0.187	0.181
IS5	NODE	0.307	0.091	0.293
IS6	TABLE	0.453	0.142	0.454
IS7	TABLE	0.339	0.200	0.504

Table 7: Runtimes (in s) for incremental view maintenance

	IC2	BI5	IS1	IS2	IS3	IS4	IS5	IS6	IS7
insertion	0.296	0.421	0.279	0.641	0.287	0.259	0.724	0.584	0.498
deletion	0.275	0.224	0.263	0.287	0.277	0.542	0.329	0.280	0.273
insertion overhead	0.078	0.168	0.085	0.101	0.058	0.060	0.103	0.195	0.127
deletion overhead	0.086	0.173	0.109	0.127	0.071	0.082	0.112	0.210	0.142

Generally, at larger scales and assuming the right view choice, view usage becomes increasingly beneficial in absolute terms because if we can avoid doing path traversal for a certain percentage of nodes, the benefits overall increase with the number of qualifying paths.

6.4 Macro-benchmark with universal queries

In order to cover a wider spectrum of queries than is possible with micro-benchmarks, we had various students generate a large and diverse set of universal view declaration and relevant usage queries over the StackOverflow and LDBC graphs.

For StackOverflow, we generated 34 views with 70 usage queries where around half were node views and half more complex views. Some of the StackOverflow queries were presented in [35]. For the ID-based approach in Neo4j, view declaration was only 1.3x slower than the baseline queries, showing an acceptable overhead for our hybrid view materialization. The view usage queries then only took 6.67% of the execution time of the baseline queries.

For the LDBC graph, we generated 38 views with 57 usage queries where around 1/3 were node views and the others were more complex views. Here, view declaration took 2.2 times longer than the baseline but the usage queries only took 3.1% of the time of the baseline queries.

The results mostly reflect the performance shown in our micro-benchmarks. Overall, view usage performance was excellent but in few cases, using views was not beneficial.

6.5 Experiments with LDBC queries

As a second macro-benchmark, we took the 7 short (IS1-7) and 3 of the complex queries (IC2,5,6) from the LDBC SNB interactive benchmark, as well as 2 queries (BI5,17) from the LDBC BI benchmark and split them into pairs of view declarations and corresponding usage queries. These queries are seeded queries where graph patterns start from a specific node that is identified as query input.

View Content While the simple IS queries themselves represent the views (just returning full elements instead of attributes), we developed two sets of splits for each of the ID and BI queries. A *declaration-heavy view* encodes the majority of query logic, and thus usage queries are more likely to be local with additional conditions. By contrast, *usage-heavy views* contain a smaller part of the query logic and then global usage queries need to cover the remaining parts. Usage-heavy views have the potential to serve more than one usage query but they might then have longer execution times.

View Type For each of the views, we chose the type that keeps the minimum information needed and is the most promising in terms of run-time for usage as explored in the previous sections. For example, the usage-heavy view for IC2 only needs the `friend` nodes and thus, a node view is enough. On the other hand, when node views are not sufficient to preserve the relationship between entities, then table, subgraph or path views are more appropriate. We found edge views less useful because searches on edges were less frequent.

Performance Analysis Table 6 presents the creation and usage times as well as the runtime of the baseline query. We show the view types that had the fastest runtime. Most view usages are significantly faster than the baseline but some outliers exist. The experiments in this section confirm the overall observations and usefulness of views. Similar to the previous experiments, usage-heavy views have an advantage when the additional search is simple and short, such as for views V_{IC2} and V_{IC6}. On the contrary, when the view is small like V_{IC5}, a local usage of a declaration-heavy view will be more beneficial. Table views were used more often than one would expect with our micro-benchmarks. The reason is that in this case, they captured exactly what was needed, making search faster.

6.6 View Maintenance

Graph modifications require to update or reevaluate affected views. Table 7 gives an intuition of the overhead. The first two lines indicate the runtime of an insert/delete that affects a particular view but without the view maintenance overhead. The last two lines indicate the additional costs for updating the view. From the LDBC benchmark, all the *IS* views are updatable. The more complex queries contain nesting, negation and aggregation. Thus, with our current view maintenance algorithm, only the usage-heavy versions of *IC*2 and *BI*5 are updatable. IS4 matches on nodes, and thus the corresponding graph updates are node insertion and deletion while the rest are adding or deleting an edge. Comparing the results from Table 7 with full recreation of Table 6, shows that updating views incrementally incurs significantly less overhead for all cases. While we only show results for SF0.1, these times do not increase significantly for larger scale factors. Therefore, the benefits of incremental view maintenance are even larger for larger databases. We also ran experiments on our micro-benchmark queries, which all allow for incremental view updates, and the results were similar.

6.7 Querying graphs over relational engines

We run similar micro-benchmarks to the ones we used for Neo4j, this time using Kuzu and Apache AGE on top of PostgreSQL. We performed slight adjustments to the setup[5]. Table 8a shows declaration and usage times for Kuzu and Table 8b for Apache AGE for the node and path views as well as execution times for the baseline queries. As path views had varied results, we added a medium complexity level where the path has two edges.

Table 8: Runtime (seconds) in the two relational engines

	Baseline	Declaration		Usage	
		ID	prop.	ID	Prop
V_{SS_node}	0.025	0.131	0.121	0.197	**0.006**
V_{CS_node}	0.328	**0.259**	6.138	0.743	**0.028**
V_{SL_node}	0.080	0.135	0.396	0.219	**0.033**
V_{CL_node}	2.306	**0.496**	22.258	**1.065**	**0.058**
V_{SS_path}	0.080	0.347	0.253	5.666	**0.070**
V_{MS_path}	0.141	1.264	1.121	3.256	**0.110**
V_{CS_path}	0.223	0.390	0.529	0.389	0.406
V_{SL_path}	0.644	2.623	4.309	407.7	**0.529**
V_{ML_path}	4.853	12.67	135.2	149.0	**1.069**
V_{CL_path}	0.911	2.289	10.86	58.02	0.938

(a) Kuzu

	Baseline	Declaration		Usage	
		ID	prop.	ID	Prop
V_{SS_node}	0.243	**0.201**	0.352	**0.228**	**0.050**
V_{CS_node}	3.771	3.861	20.86	**0.231**	**0.071**
V_{SL_node}	0.266	0.437	1.575	0.269	**0.132**
V_{CL_node}	12.32	12.58	55.61	**0.277**	**0.174**
V_{SS_path}	0.090	0.316	0.878	0.382	0.142
V_{MS_path}	0.401	0.414	1.084	0.493	**0.235**
V_{CS_path}	1.028	**0.498**	2.387	**0.742**	**0.361**
V_{SL_path}	0.305	2.640	13.13	11.36	1.059
V_{ML_path}	3.407	**1.783**	258.8	6.130	**1.486**
V_{CL_path}	3.864	**1.775**	289.7	5.157	**2.959**

(b) Apache AGE/PostgreSQL

Kuzu: The *ID-based approach* always executes declaration queries faster than the property-based approach, but its usage times are much worse and even worse than the baseline query in most cases. Kuzu works poorly with the ID-based approach because of limited index support for IDs. Kuzu searched for elements by checking for each record in the table whether the ID of the record is in the ID list of the view which was inefficient when the list was large.

For the *property-based* approach, the declaration queries are considerably more expensive than the baseline, similar to what we saw in Neo4j. For usage queries, they are 2.5 to 40x faster on node views than the respective baseline query. Looking up a view property is well supported by Kuzu and thus allows for significant performance benefits. Using path views outperforms the baseline for simple and medium views (between 1.14x to 4.5x faster), but not for complex views. Recall that for Kuzu we created extra edges for each path with appropriate labels so that Kuzu could keep them in special tables. With 1- and 2-edge path patterns, Kuzu first looks up the edges and then combines them to paths. For the 3-edge path, it does this only for the first two edges and then performs a join. Despite the smaller edge tables, this was still inefficient. The lookup for sequence numbers and ranks turned out to be inefficient.

Furthermore, we ran the same marco-benchmarks on universal queries from Section 5.4 on Kuzu using the property-based approach. View declaration queries overall took nearly 9x longer than the baseline queries for StackOverflow and 3x for the LDBC data, but view usage queries were overall 5x faster than the baseline for StackOverflow and 3x faster for the LDBC data. These results validate our micro-benchmark findings.

Apache AGE/PostgreSQL The *ID-based approach*, similar to Kuzu, is faster in declaration than the property-based approach while always slower for usage for all view types. ID-based usage, in contrast to Kuzu, is often faster or takes similar time as the baseline for both node and path views although they are also sometimes slower.

Similar to Kuzu, using node views in the *property-based approach* is always significantly faster than the baseline (2x to 53x faster) because of the fast view property lookup. Interestingly, using path views is faster than the baseline for medium and complex views (up to 2.8x) but up to 3.5x slower for simple paths. The short paths have a large

[5] We experienced memory problems with Kuzu with large intermediate or end results, even with the baseline queries. Using the WITH clause in our rewritten usage queries aggravated the problem. As such, we changed V_{SL_node} and V_{CL_node} to return less nodes. With Apache AGE, the overall performance degrades dramatically for both view declaration and baseline queries with large graphs and large return sets. As such we generated views with generally smaller result sets.

result set and unwinding the identifiers in Apache AGE appears to be costly while direct joins are more efficient.

We would like to note that Kuzu, having a specialized relational engine, executed queries significantly faster than Neo4j or Apache AGE. The latter, based on the general-purpose relational database system PostgreSQL, had the overall slowest execution times.

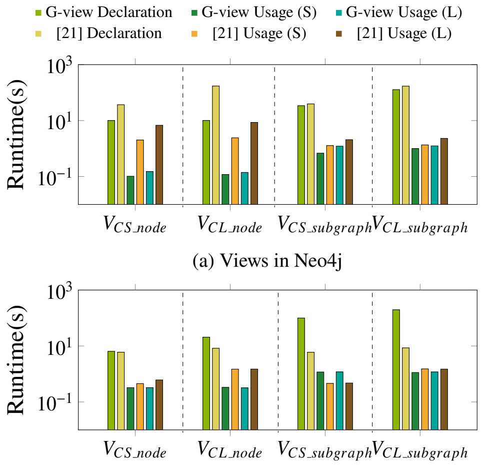
6.8 Comparison with existing implementations

In recent work, Han and Yves [21] proposed views as transformations over the original graph. As such, they do support local usage over subgraphs. Furthermore, they provide implementations on top of Neo4j and PostgreSQL. Therefore a partial comparison is possible. We consider a user that is interested in a *complex* graph pattern that has either a *small amount* (CS) or a *large amount* (CL) of matches. In Scenario I the user wants to find specific *nodes* in this complex pattern. In Scenario II, the user looks for *patterns* and/or new conditions within the complex pattern. The usage queries can cover a *small (S)* or *large (L)* portion of the view. Using the approach of [21] we always create a subgraph view. For G-View, we use a node view for the first scenario and a subgraph view for the second.

Neo4j For G-View we use the ID-based implementation and for [21] the *Overlay* implementation, where they attach properties to nodes and edges (in spirit similar to our property-based approach). These are the implementations with the best performance for each of the systems. Figure 6(a) shows the performance for declaration and usage. Declaration is always faster with G-View as managing IDs is more light-weight than adding properties to nodes/edges in [21]. In scenario I, when G-View can exploit a node view for usage, it is up to 20x faster than [21], and that independently of the size of the view and the result set of the usage query. In scenario II, where both solutions use subgraphs, G-View is still faster by up to a factor of 2.

Apache AGE / PostgreSQL For G-View we use the property-based option and for [21] the implementation based on *Subgraph Substitution Relations (SSRs)*, that keep track of transformations in a rule-based manner. Again, these are the best-performing options for both systems. Figure 6 (b) shows the results. In scenario I, G-View again always performs better than [21] as it can use the simpler node views (achieving 1.4x to 4.5x speedup). In scenario II, where both use subgraphs, G-View is better than [21] for large views but worse when the view is small. The results are the same independently of the result size of the usage query.

Overall, the performance of the approaches varies depending on view type, usage queries and internal implementation. Note also that all usage queries in this experiment set outperform the baselines (not shown in the figures). G-View can use different view types depending on the need and thus, has more potential for optimization.



(a) Views in Neo4j

(b) Views in Apache AGE/PostgreSQL

Figure 6: G-View vs. Han and Yves [21]

6.9 Lessons Learned

Our results show that our middleware-based view management can significantly reduce runtime for frequent queries compared to baseline queries. For Neo4j, an ID-based implementation performs better than a property-based implementation and the inverse holds true for relational execution engines. This shows that view management needs to be adjusted to the underlying system. Furthermore, our analysis shows that the performance of different view types can differ depending on the complexity of the views, the size of the result and the further usage of the views. It is not always obvious which view types are the best but node views are generally good for node lookups and can also be used for global usage queries with new graph patterns. Path views are efficient when usage queries want to access specific path patterns that are simple, but might have a performance penalty if complex. Generally, our strategy to use `UNWIND` did not always create the best execution plans in the database engine.

In terms of convenience and the ease of crafting a usage query each view type has its advantages and limits. Node and table views allow referring back to nodes (and edges) based on variable names used during view declaration, which can be useful and also reduce overhead. As to finding paths, global usage is necessary for node views but local usage could be enough for other view types. Table views allow to keep track only of those elements in a path pattern that are of interest for a user. Usage queries for path and subgraph views, even if local, must rewrite the actual path pattern if the usage query has additional conditions or only partial results should be returned. Our experiments show that global queries do not necessarily have bad performance. Thus, one might want to consider to keep views simpler so that they can serve a wider range of usage queries.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed graph-based language extensions to declare and use various types of views that differ in how they capture subcomponents of the underlying graph. Our views can be used together with the underlying graph similar to view usage in relational systems. We presented various implementation strategies to maintain view information and for incremental view maintenance. We developed micro- and macro-benchmarks to understand how views could help query execution. We demonstrated that with careful implementation and usage choices, views can offer significant advantages.

In the future, we are interested in looking how such view management strategies could be implemented within the query engine to further optimize creation and utilization, in particular in order to quickly find relevant paths, and choose appropriate execution plans. The property-based approach could also potentially better separate view data from the actual data of the underlying graph.

REFERENCES

[1] [n.d.]. Apache AGE. https://age.apache.org/. (Accessed December. 19, 2024).

[2] [n.d.]. Cypher – the SQL for Graphs – Is Now Available for Apache Spark. https://neo4j.com/blog/cypher-for-apache-spark/. (Accessed July.14, 2024).

[3] [n.d.]. GQL Standard. https://www.gqlstandards.org/. (Accessed July. 14, 2024).

[4] [n.d.]. Gremlin Query Language. https://tinkerpop.apache.org/gremlin.html. (Accessed Mar. 14, 2024).

[5] [n.d.]. GSQL Query Language. https://www.tigergraph.com/gsql/. (Accessed Mar. 14, 2024).

[6] [n.d.]. JanusGraph. https://janusgraph.org/. (Accessed April. 25, 2024).

[7] [n.d.]. Neo4j. https://neo4j.com/. (Accessed July.14, 2024).

[8] [n.d.]. Transforming your data to graphs - Part 1. https://blog.kuzudb.com/post/transforming-your-data-to-graphs-1/. (Accessed Jun.23, 2024).

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 1421–1432.

[10] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2, Article 198 (jun 2023), 25 pages. https://doi.org/10.1145/3589778

[11] Thomas Beyhl and Holger Giese. 2016. Incremental View Maintenance for Deductive Graph Databases Using Generalized Discrimination Networks. In *Proceedings Second Graphs as Models Workshop, GaM@ETAPS 2016, Eindhoven, The Netherlands, April 2-3, 2016 (EPTCS)*, Vol. 231. 57–71. https://doi.org/10.4204/EPTCS.231.5

[12] José A. Blakeley, Neil Coburn, and Per-Åke Larson. 1989. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.* 14, 3 (Sept. 1989), 369–400. https://doi.org/10.1145/68012.68015

[13] Angela Bonifati and Stefania Dumbrava. 2019. Graph Queries: From Theory to Practice. *SIGMOD Rec.* 47, 4 (may 2019), 5–16. https://doi.org/10.1145/3335409.3335411

[14] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[15] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph views for efficient graph analytics. In *Proceedings of the 36th International Conference on Data Engineering (ICDE)*. 193–204.

[16] Nigel Dennis. 1994. Scheduling strategies for airline hub operations. *Journal of Air Transport Management* 1, 3 (1994), 131–144. https://doi.org/10.1016/0969-6997(94)90034-5

[17] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 2246–2258.

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 1433–1445.

[19] Ashish Gupta and Inderpal Singh Mumick. 1999. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press. https://doi.org/10.7551/mitpress/4472.003.0016 arXiv:https://direct.mit.edu/book/chapter-pdf/2305627/9780262287500_cak.pdf

[20] Ashish Gupta and Inderpal Singh Mumick. 1999. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press. https://doi.org/10.7551/mitpress/4472.003.0016 arXiv:https://direct.mit.edu/book/chapter-pdf/2305627/9780262287500_cak.pdf

[21] Soonbo Han and Zachary G. Ives. 2024. Implementation Strategies for Views over Property Graphs. *Proc. ACM Manag. Data* 2, 3 (2024), 146. https://doi.org/10.1145/3654949

[22] Edward Hung, Yu Deng, and Venkatramanan S Subrahmanian. 2005. RDF aggregate queries and views. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. 717–728.

[23] Elizabeth Hunter, John Kelleher, and Brian Mac Namee. 2019. Degree centrality and the probability of an infectious disease outbreak in towns within a region. In *33rd Annual European Simulation and Modelling Conference 2019, ESM 2019 (33rd Annual European Simulation and Modelling Conference 2019, ESM 2019)*, Pilar Fuster-Parra and Oscar Valero Sierra (Eds.). EUROSIS, 195–202. https://doi.org/10.21427/bbp3-hr31 Publisher Copyright: Copyright © 2019 EUROSIS-ETI.; 33rd Annual European Simulation and Modelling Conference, ESM 2019 ; Conference date: 28-10-2019 Through 30-10-2019.

[24] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2016. Optimizing aggregate SPARQL queries using materialized RDF views. In *Proceedings of the 15th International Semantic Web Conference (ISWC)*. 341–359.

[25] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. 2011. Rewriting queries on SPARQL views. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*. 655–664.

[26] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proc. VLDB Endow.* 12, 4 (dec 2018), 390–403. https://doi.org/10.14778/3297753.3297759

[27] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 469–478. https://doi.org/10.1145/2505515.2505561

[28] Tova Milo and Dan Suciu. 1999. Index Structures for Path Expressions. In *Database Theory — ICDT'99*, Catriel Beeri and Peter Buneman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 277–295.

[29] Yannis Papakonstantinou and Vasilis Vassalos. 1999. Query rewriting for semistructured data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/304182.304222

[30] E. Prud'hommeaux and A. Seaborne. [n.d.]. SPARQL query language for RDF. W3C Recommendation. https://www.w3.org/TR/rdf-sparql-query/. (Accessed Mar. 14, 2024).

[31] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. O'Reilly Media, Inc.

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

[33] Semih Salihoglu. 2023. Kùzu: A Database Management System For "Beyond Relational" Workloads. *SIGMOD Rec.* 52, 3 (nov 2023), 39–40.

[34] Bryce Merkl Sasaki. [n.d.]. Import 10M Stack Overflow Questions into Neo4j In Just 3 Minutes. https://neo4j.com/blog/import-10m-stack-overflow-questions/. (Accessed Mar. 14, 2024).

[35] M. Shahrad, Y. Gu, Y. Zheng, and B. Kemme. 2024. Towards View Management in Graph Databases. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE Computer Society, Los Alamitos, CA, USA, 355–359. https://doi.org/10.1109/ICDEW61823.2024.00053

[36] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1887–1901. https://doi.org/10.1145/2723372.2723732

[37] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (dec 2022), 877–890. https://doi.org/10.14778/3574245.3574270

[38] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proceedings of the VLDB Endowment* 16, 4 (2022), 877–890.

[39] Oskar Van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[40] Yue Zhuge and Hector Garcia-Molina. 1998. Graph structured views and their incremental maintenance. In *Proceedings 14th International Conference on Data Engineering*. 116–125.