Exploring Views in Graph Database Systems

Yu Ting Gu

School of Computer Science
McGill University, Montreal

August 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Computer Science

©Yu Ting Gu, 2021

Abstract

Graph-based database systems, which use graph structures for data storage and queries, have been emerging as an interesting alternative to relational database systems for many applications. In relational systems, a view defines a sub-set of the database represented through a SQL query, and can but does not need to be materialized. A view can then later be used in other queries just like a regular table providing data and query simplicity, security, and sometimes increased performance. However, while the concept of views has been an important feature for relational database systems, we are not aware of any graph-based system supporting views. Graph based database systems have very different query languages, deal with both nodes and relationships, do not guarantee a fixed return format, and the storage of data is different from relational databases. Thus, there is no straightforward way of supporting views in a graph-based system.

This thesis investigates views for graph databases and proposes several ways to declare and use views for the popular graph-based database system Neo4j. The first challenge is to define a way for views to be created and used in subsequent queries. For that, we provide a language to define and use views by extending Neo4j's query language Cypher. We then distinguish between views that are materialized and those that are not and implement both approaches. For the latter we use an automatic query re-writing approach that integrates the view declaration query into the query that uses the view. For materialized views, we use internal node and edge identifiers specific to Neo4j, and store them within a middleware. A second challenge for materialized views is an efficient view maintenance scheme in the case of an update to the database. We use different strategies based on the components of a view and any incoming graph modification queries to evaluate whether a view should be re-evaluated or not. We evaluate the system by analyzing the cost of creating a materialized view by comparing the execution times of queries that use our view implementations vs a set of equivalent baseline queries, and by analyzing the effectiveness of our view maintenance. We categorize views according to two parameters and look for trends in performance within each category. The two approaches both return promising results, often performing better than the baseline queries. The materialized approach shows a tendency to benefit views that take a long time to create.

Abrégé

Les systèmes de base de données orientée graphe, qui utilisent des structures de graphes pour le stockage de données et les requêtes, sont devenus une alternative intéressante aux systèmes de bases de données relationnelles pour de nombreuses applications. Alors que le concept de vues a été une caractéristique importante pour les systèmes de bases de données relationnelles, nous ne connaissons, cependant, aucun système orienté graphe pouvant exploiter les vues. Dans les systèmes relationnels, une vue définit un sous-ensemble de la base de données représentée par une requête SQL et peut, mais n'a pas besoin, d'être matérialisée. Une vue peut ensuite être utilisée dans d'autres requêtes, tout comme une table classique offrant simplicité et sécurité des données et des requêtes. Les systèmes de base de données orientée graphe ont, cependant, des langages de requête très différents et traitent à la fois des nœuds et des relations. De plus, ils ne garantissent pas un format de retour fixe et le stockage de données est différent des bases de données relationnelles. Ainsi, il n'y a pas de moyen simple d'exploiter les vues dans un système orienté graphe.

Cette étude aborde ce problème et propose plusieurs façons de déclarer et d'utiliser des vues pour Neo4j, un système de base de données orientée graphe populaire. Le premier défi consiste à définir un moyen de créer et d'utiliser les vues dans les requêtes. Pour cela, nous fournissons un langage pour définir et utiliser des vues en étendant le langage de requête de Neo4j, Cypher. Nous distinguons ensuite les vues matérialisées de celles qui ne le sont pas, puis nous mettons en œuvre les deux approches. Pour ce dernier, nous utilisons une méthode de réécriture automatique des requêtes qui intègre la requête de définition de vue dans la requête qui utilise la vue. Pour les vues matérialisées, nous utilisons des identifiants internes de nœuds et d'arcs spécifiques à Neo4j, puis les stockons dans un système médiateur. Un deuxième défi pour les vues matérialisées est un système de gestion de vues efficace dans le cas d'une mise à jour de la base de données. Nous utilisons différentes stratégies basées sur les composants d'une vue et sur toutes les requêtes entrantes de modification de graphe pour déterminer si une vue doit être réévaluée. Nous évaluons le système en analysant le coût de création d'une vue matérialisée en comparant les temps d'exécution des requêtes qui utilisent nos réalisations de vue par rapport à un ensemble de requêtes de base équivalentes. Nous analysons aussi l'efficacité de notre gestion de vue. Nous catégorisons les vues en fonction de deux paramètres et mesurons les tendances dans chaque catégorie. Toutes deux approches obtiennent des résultats prometteurs qui sont souvent plus performants que les requêtes de base. L'approche matérialisée démontre une tendance à profiter des vues qui prennent du temps à se créer.

Acknowledgements

My research would have been impossible without the aid and support of my supervisor and co-supervisor. First, I would like to thank my supervisor, Professor Bettina Kemme, who sparked my interest in research while I was still an undergrad in 2018. Your expertise in the field has been invaluable to the progression of this thesis, and I am forever grateful for your commitment to always providing critical feedback. I would also like to thank my co-supervisor Professor Jörg Kienzle, who was always there to provide additional insight. To you two: Thank you for supporting me and always guiding me towards a better direction throughout the past two years, thank you for the countless lessons that you have taught me along the way, and thank you for always encouraging me.

I would like to express my appreciation to my family for always being there for me. I would like to thank parents Cherry and David, along with my dog Elsa, for their emotional support throughout this endeavour. I would like to thank Sean MacRae for translating the abstract to French.

Contents

Chapter 1

Introduction

1.1 Graph Database Systems

While relational database systems (RBDMs) have been intensely studied for decades, there has also been continuous growth towards graph database systems. The graph structure provided by these systems gives us in particular the ability to model networks with ease. Whether it is a road network or a social network, as soon as the model contains plenty of relationships, graph databases might be a better option to store this information instead of traditional relational database systems. Even image [6] and video [20] data can be organized efficiently with graph databases. A large contributor to the growth of graph databases is Neo4j [29] [15], a open-sourced graph-based database management system, that models databases as property graphs, where nodes and edges can have properties similar to attributes in the relational model. With Neo4j's graph query language, Cypher, a wide range of queries on nodes and edges can be executed to find specific nodes, edges, or entire paths within the database. In addition, thanks to the graph-like structure, well-known graph algorithms can be leveraged to improve search speeds for graph-specific queries. Companies such as Adobe [1], Airbnb [2], and banks are already using a graph database system.

1.2 Views

The concept of views is widely used in relational database systems. In relational database systems, all data is stored within tables, and related data may be spread across many tables. Often times, users are interested in data that is spread across tables. Thus, complex queries are written to access such related data. The standard query language is SQL, where SELECT statements provide features to join tables according to various conditions. These queries may potentially be difficult to write and very expensive to run. For these purposes, relational database systems offer views. A view is basically a SELECT query that is registered in the database system with a specific name. As SELECT queries always return a table, a view can then be re-used in subsequent queries as if it were a real table. A view may be *materialized*, in which case the result of the query is physically stored within the database, or purely virtual, i.e, *non-materialized*, in which case only the query text is stored and dynamically executed whenever referenced. Views provide security and simplicity, and with the materialized type of views, unnecessary re-evaluation of queries can be avoided, preventing high I/O costs when the underlying view is complicated. However, to support the materialization of such views, maintenance is necessary to ensure that the view does not become outdated upon a write on the database.

While view management is well understood for relational systems, we are not aware of graph databases supporting views. Therefore, this thesis looks at how to define, use, and manage views in a graph-based database system, and evaluates their usefulness. Not only do graph database systems usually store data differently compared to relational systems [28], but the types of queries that can be executed over these graphs are also different. There is no query language standard such as SQL for graph-based systems. Instead, each system uses their own query language or API. Cypher, Neo4j's query language, allows queries with many different return types, be it a set of nodes, a set of edges, or a set of paths, which is a combination of both nodes and edges. As such, it is already difficult to define what a view *should* look like for such a system. That is, we must first properly define what a view looks like for a graph database.

The second question is then how to use a view. In relational database systems a view can be the single input for a SQL query, or it can be joined with other views or tables in the database. Expressing this for graph database views can be challenging, considering the query language is not similar to SQL.

Finally, we must consider the maintenance of these views. While non-materialized views require no maintenance, materialized views can become outdated when a change on the graph occurs, potentially qualifying or disqualifying parts of the graph for views that already exist. Some sort of maintenance algorithm is needed to ensure that all views are up-to-date.

1.3 Contributions

This thesis investigates whether the concept of views developed for relational databases is applicable and beneficial for graph-based database systems. In this regard the thesis makes several contributions. First, we augment a graph database query language with an extension to declare views. While our proposed language extension should apply to any graph query language we present a concrete example by augmenting the syntax and semantics of Neo4j's graph query language Cypher. Second, we provide an implementation of this query extension on top of Neo4j in form of a middleware. We both define solutions for materialized and non materialized views, and compare the performance to a system that does not use views.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. We first provide in **Chapter 2** the necessary background. In order to provide support for views, it is necessary to first define what a view means in a graph database context. To understand this, we must understand what graph databases are made of and what is returned by a query. In particular, we first discuss the high level concepts of both relational and graph databases, including how data is organized and what type of data can be returned by queries from both systems. We introduce the concept of a property graph and graph database, and the syntax and semantics of Cypher, which is Neo4j's graph database query language. Due to the complexity of graphs, there are several different forms of data that a graph query can return. We discuss several of these in order to make sure our extension to provide views can properly handle these cases. We also have a closer look at how views are managed in relational systems, as we are guided by these implementations.

We then present in **Chapter 3** our language extensions to Cypher and their semantics. In particular, we define how to create views and how to use them in subsequent queries. We allow queries to only refer to a view (local) or also to data in the underlying base graph (global).

Chapters 4 and 5 present two design and implementation options. As mentioned before, views in RDBMs can be materialized or non-materialized and we design a solution for both options. In both cases we use a middleware that offers the language extension and communicates with the database to retrieve relevant information when necessary. While the extension is on Neo4j in particular, we do not change Neo4j itself as all the functionality is implemented in the middleware.

The first approach we consider is the non-materialized approach presented in **Chapter 4**. Only the query text of the view is stored, and executed as a sub-query whenever a subsequent query uses the view. In order to do this, a re-write of the query which uses the view is necessary, and we discuss how this rewrite is done.

The second approach, discussed in **Chapter 5**, is a materialized approach which stores query results in the middleware for future reference. We discuss in detail how we store the results and how they are used by queries that use the views. Furthermore, we discuss the maintenance algorithm, which identifies which views require re-evaluations should the underlying database change.

In **Chapter 6** we discuss performance results for the two approaches. We create our own micro-benchmark. We create a set of views over a graph containing social media data, and then define a set of queries that use these views. We evaluate the overhead of creating materialized views and also compare the performance of the queries that use the views for both the materialized and non-materialized options against equivalent queries that run on the base graph. We present an in-depth performance comparison highlighting the advantages and disadvantages of the different implementations. For materialized views, we then verify the correctness of the maintenance algorithm and discuss its effectiveness. Finally we provide a summary and overall guideline for using views.

In **Chapter 7** we discuss related works surrounding graph database systems and views.

In **Chapter 8** we give an overall conclusion along with a few avenues for future work.

Chapter 2

Background

2.1 Relational Database Systems and Views

2.1.1 Relational Data

A relational database separates the data into different tables. A table most commonly groups all data of a certain entity type, e.g. all users. Each row of a table corresponds to a single entity, and each column corresponds to some attribute belonging to the row. Tables may contain a primary key attribute where the value is unique for each row in the table, e.g. *userId* in a *user* table. It may also contain foreign key columns, which are references to primary keys of other tables. Tables are related through these primary and foreign keys, in which case they are often queried together with a join operator.

Working Example Throughout this thesis we use as a working example the "Stack-Overflow" dataset [18]. It originally follows the relational model as presented here. We will later transform it to a graph model to illustrate the differences between relational and graph models. The main entities are *Users*, *Posts*, and *Tags*. They are related to each other. A User can write many Posts but a Post is written by a single User. A Post can have at most one parent Post, but can be parent of many Posts. Thus both of these relationships are one-to-many relationships. Furthermore a Post can have many Tags, and a Tag can be attached to many Posts - thus, a many-to-many relationship. All this is translated as follows into tables:

- a User table which contains the following attributes: userId, displayName, aboutme, reputation, upvotes, downvotes, and view.
- a Post table which contains the following attributes: post body, the number of comments, a postId, and a score, the userId of the user who posted the post, and a timestamp[1].
- a Tag table which contains the tag name, which we may also call the tagId.
- a Post-Parent table, which relates a Post to a parent Post.
- a Post-Tag table that relates a Post to the Tags that are attached to the post.

Tables 2.1 to 2.5 show example instances of these tables. One-to-many relationships can be expressed in two different ways via tables and foreign key relationships. For the Users and their Posts, we have embedded the user reference within the post table as foreign key, but for the Post and its parent, we outsourced this relationship to a separate table. We made the child postId the primary key, ensuring that a child can have at most one parent. For many-to-many relationships we must always use an additional table as shown in the Post-Tag table where every post can have many tags and every tag can have many posts.

2.1.2 Views in Relational Database Systems

A view is a subset of the relational data that is represented as a SQL query that returns a table, and can be used in subsequent SQL queries as an input table. Thus the definition and usage of a view are relatively straightforward in relational systems. As for implementation, a view can be non-materialized or materialized. Non-materialized views are implemented by storing the text of the query together with

[1]Timestamps are not part of the original data, but we include this in the description to highlight the possibility of an attribute which represents a relationship between two tables within the model (User and Post), once the model is translated into a property graph representation.

Table 2.1: User table

userId	displayname	aboutme	reputation	upvotes	downvotes	views
1001	foo	baz	130	1213	119	18593
1002	bar	doe	80	404	71	1099

Table 2.2: Post table

postId	body	comments	score	userId	timestamp
10000	loren ipsum	50	3500	1001	2020-08-08
10001	dolor sit amet	10	500	1002	2020-08-09

Table 2.3: Post parent relationship

child	parent
10000	10001

Table 2.4: Post-tag relationship

postId	tagId
10000	databases
10000	neo4j
10001	databases
10001	python

Table 2.5: Tag table

tagId
databases
java
neo4j
python

a view name. When a query uses a view name as one of its input tables, the query is rewritten replacing the view name with the SQL query that defines it. For instance, if a view `v` is created with the view declaration `SELECT a,b FROM table`, and a query `SELECT * FROM v` uses the view, then this query is first rewritten by replacing `v` with a subquery as `SELECT * FROM (SELECT a,b FROM table)`. The query optimizer will then automatically optimize the query and rewrite it to `SELECT a,b FROM table`.

Databases also offer materialized views. Upon declaration of a view, the corresponding query is executed and the result is stored in a view table similar to standard database tables. This prevents re-evaluation of the view declaration query every time a view is used, but also leads to several disadvantages. Materialized views require maintenance in order to prevent views from becoming stale. That is, whenever a table which a view is built from undergoes a change - an update, delete, or insert, this change may affect the view. For instance, a delete on the base table of a row which qualifies for the view requires the corresponding entry in the view to be deleted as well. There are two options for maintenance. The first is to invalidate and re-compute the view: when an underlying table is updated, then the view table is deleted. The query used to create the view is entirely re-executed and the results are stored into a new table. While this is guaranteed to be correct, frequent updates to the database will cause views to be recomputed very often, which is computationally expensive. The second option, which circumvents this downside, is to provide incremental maintenance[9] [23]: whenever one of the underlying tables is updated, the view table is also updated to reflect the change in the underlying table. For incremental view maintenance (IVM), the view does not undergo a full re-computation. Instead, the changes on the base tables are evaluated to determine whether they affect the views, and if so, the changes are applied as well onto the view. However, it can be quite complicated to determine how the view needs to be updated. Thus, incremental view maintenance is often only supported when the view declaration queries are simple. Complex queries such as joins usually require a recomputation upon invalidation.

2.2 Graph Databases

While relational databases are organized based on the relational model of the data, graph databases are often used when the relationships between the data are particularly important.

2.2.1 Property Graphs

We look specifically at graph databases that follow a *property graph* model. A property graph consists of *nodes* and *relationships*. While relational databases represent entities as rows in a table, property graphs represent entities as nodes. These nodes may also have attributes, which are represented by key-value pairs called *properties*. A property of a node is similar to an attribute for relational databases. In addition to properties, nodes can also be optionally tagged with one or more *labels*. A label in a graph database is often used analogous to the table name in a relational database, identifying for each entity what *type* of entity it is. However, labels and properties are not necessarily non-empty. That is, a node or edge may have no label and they may contain null property values. A relationship is an edge between two nodes. In a property graph, relationships may be unidirectional in which case they have a start and an end node, or non-directional, in which case the edge simply connects two nodes. Like nodes, relationships may also contain properties and a label to identify what kind of relationship it is.

A powerful feature of graph databases is a constant time cost for edge traversals [24]. Due to the storage of nodes and relationships internally, graph traversals are typically efficient. This, along with the existence of graph algorithms, such as search or shortest-path, makes it efficient to look for common patterns inside data represented by graph databases.

Translated Working Example To show an example of such a graph, we translate the StackOverflow data presented in the previous section to a property graph. See Figure 2.1 for a visual representation of the graph. We categorize nodes with three possible labels, *User*s, *Post*s, and *Tag*s. As for relationships, a User can post a Post, indicated by a *POSTED* relationship, a Post can contain one or more tags, indicated by a *HAS_TAG* relationship, and a Post can be a child of another Post, indicated by a *PARENT_OF* relationship. While in the relational model the relationships were represented by foreign key references to other tables, all relationships are explicit in graphs. As with the attributes in the relational database, each type of node and the *POSTED* relationship also contains different attributes. In fact for *User*, *Tag*, *PARENT_OF*, and *HAS_TAG* the attributes are the same as the ones in the corresponding tables. However, we split the *Post* table into a Post node, and maintain the poster information (originally embedded in the Post table as a foreign key) as a relationship from the User node. In our example all relationships are directed.

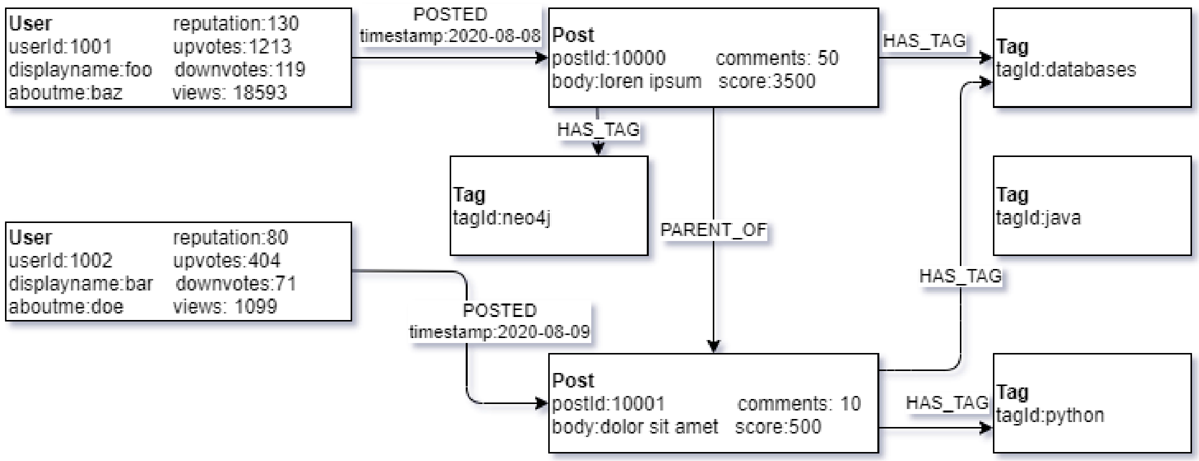

Figure 2.1: StackOverflow data translated into a property graph representation

2.2.2 Cypher Query Language

While SQL is a standard language for relational databases that basically all relational systems have adopted, there is no such standard for graph databases. Thus, in this thesis we take Cypher [14][15], the query language for Neo4j, as an example. A Cypher query can be used to fetch information from the graph database. Queries can have different return types: from a single attribute of a node to a set of nodes to a set of paths within the graph. This makes it more complex than SQL SELECT queries that always return a table. Cypher queries are visually intuitive, and use patterns to match onto the graph. In a query we may specify particular types of nodes or relationships that we wish to match on.

We now introduce three types of queries. Note that this categorization is not part of Cypher's official definitions but will be useful later when we discuss view details.

Node Queries We call a query a node query if it only returns nodes. The most basic form of a node query returns all nodes:

```
MATCH (n) RETURN n
```

A set of parentheses indicates a node within the graph. We do not necessarily need to enclose a variable within the parentheses, but in order to return something a variable must be included in the match pattern. Therefore, we use `n` to refer to the node. In this query, we search for *all* nodes in the graph, meaning that this query will return Users, Posts, and Tags. A node is returned in its entirety, including all its properties and labels.

Cypher allows us to indicate conditions that a node must fulfill to be returned. The condition can be the specification of a label or a restriction on attributes and their values. For instance, we may include a label condition on `n` to match only on nodes labeled with User with the following query:

```
MATCH (n:User) RETURN n
```

By appending a colon along with the label to the node variable, we add a condition to the query. Referring to SQL, this query is similar in nature to `SELECT * FROM Users`. A node in the pattern may also contain *only* a label, but like with the previous example, we require a variable in order to return something.

Aside from the label condition, we may also include conditions on the attributes of the nodes. Like in SQL, attribute conditions are specified with a `WHERE` keyword, and can be used to access and compare attributes. Similarly to SQL, where attributes are referred to with `tableName.attribute`, we refer to attributes with `node.attribute`. The following query filters Users with a condition to only return those whose reputation property is above a certain value.

```
MATCH (n:User) WHERE n.reputation > 120 RETURN n
```

Within the conditions that may be specified, the `IN` operator indicates membership. Given a list or set S and an element e, the `IN` operator will evaluate to `true` if S contains e and they share the same datatype, otherwise it will evaluate to `false`. For example, consider the following query, which will only return User nodes which contain a userId attribute whose value is between 1 and 5:

```
MATCH (n:User)
WHERE n.userId IN [1,2,3,4,5]
RETURN n
```

Node queries with paths

The next type of query is one which includes a path in its pattern matching.

```
MATCH (n:User)-[:POSTED]->(p:Post) RETURN n
```

We see that as with nodes, edges can also have label or attribute conditions. This query matches on patterns where there exists a directed edge from any User node to any Post node, where said edge is labeled as `POSTED`. In SQL, such semantics often lead to joins between tables, as we see in the following SQL query which returns the same information from the relational model:

```
SELECT n.userId, n.aboutme, ..., n.views
FROM User n, Post p
WHERE n.userId == p.userId
```

A relationship in Cypher is indicated by square brackets surrounded by a dash on both sides: `-[]-`. The relationship may be unidirectional (`-[]->` or `<-[]-`), bidirectional (`<-[]->`), or nondirectional (`-[]-`) (functionally the same as bidirectional). Like with nodes and parentheses, a relationship does not need to have a variable nor a label assigned to it within the pattern; it may contain either, none, or both. In fact, had we used `-[]-` in this example, the result would have been identical, since 1) all User-Post relationships are already POSTED relationships according to the model and 2) there is only one type of User-Post or Post-User relationship. For the same reasons, it would also be correct to omit `p:Post` in the second pair of parentheses. We can omit `p` because we have no attribute condition for it nor do we return it.

Note while the query contains a path constraint it still has as a return set only nodes. This time, it only returns Users that have posted at least one Post. We may also choose to return multiple nodes (per pattern match). For example, if we are interested in returning both `n` and `p`, we may do so by specifying `RETURN n,p`.

Relationship queries

A query with a path pattern can also return relationship information. In fact, in order to return a relationship, a path must be specified. We can then return a relationship if we assign a variable to it. For example, using a simple query we can return all `POSTED` relationships:

```
MATCH (n:User)-[r:POSTED]->(p:Post)
RETURN r
```

Path queries

Finally we look at the same query but we change the return type from sets of nodes or edges to a set of paths. This can be done by assigning a variable to the entire pattern:

```
MATCH p = (n:User)-[r:POSTED]-() RETURN p
```

This returns a set of paths: for each path which satisfies the pattern, we get a User node, a POSTED relationship, and a Post node.

Other types of queries

A query can also return both nodes and relationships. For example, in the above query we could return `n,s`. Furthermore, we may also choose to return a specific attribute of a node or relationship. An example of returning an attribute of a node is:

```
MATCH (n:User) RETURN n.userId
```

An example of returning an attribute of a relationship would be:

```
MATCH (n:User)-[r:POSTED]-(p:Post) RETURN r.timestamp
```

While queries that return attributes, and queries that return both nodes and relationships are allowed in Cypher, our language extension to define views introduced in Chapter 3 does not allow these forms of queries as input.

Graph Changes

Finally we look at Cypher statements that update the graph. We limit updates to three different types: additions of nodes or edges, deletions of nodes or edges, and modifications of node or edge properties. A node or edge may be added at any time with the `CREATE` keyword. A node may be deleted only if it has no edges with any other nodes[2], while an edge may be deleted at any time with the `DELETE` statement. Finally, a node or edge property can be changed at any time with the `SET` keyword to set a property, or the `REMOVE` keyword to remove a property. Syntactically, these keywords must come after the `MATCH` pattern and conditions, and replaces the `RETURN` statement:

```
MATCH (n:User)-[:POSTED]-(p:Post)
WHERE n.userId=100
SET n.reputation=150
```

We note that studies have been done and inconsistencies have been found regarding Cypher's lack of atomicity in graph update statements [16]. However, these cases are very specific and we go with general semantics and do not consider queries with unpredictable behavior.

Unique identifiers in Neo4j

Neo4j has a particular way of storing nodes and relationships. Each node is assigned an identifier unique in regard to all other nodes, and each relationship is assigned an identifier unique in regard to all other relationships. That is, a node with a User label cannot have the same identifier as a node with a Post label, and a relationship with a POSTED label cannot have the same identifer as a relationship with the PARENT_OF label. However, any node can have the same identifier as any relationship. Neo4j also has an internal index on these identifiers, which is used whenever the function `ID(n)`, which returns the identifier of `n`, is called from a query. This makes queries on identifiers particularly efficient.

[2]Cypher also supports a `DETACH DELETE` which automatically deletes edges attached to a node to be deleted, but our current implementation does not support this statement.

2.2.3 Additional Cypher Functions and Clauses

Cypher provides many more language features. We do not support them within queries that define views, but we use several of them in our implementation to support non-materialized views. In particular, Cypher works with the graphs and query return values in various formats, and distinguishes between rows and lists. In most parts of the thesis we ignore the subtle differences but when we rewrite queries for non-materialized views we have to use some of Cypher's special functions to transform return values in the formats we need and to pipeline results across several queries. We shortly outline these functions here.

Functions on path variables

Cypher provides `NODES` and `RELATIONSHIPS` functions. Both of these functions map from a given path variable to a list of nodes and relationships respectively which belong to the path variable. When more than one path is returned, the list returned contains all nodes (resp., relationships) from all paths.

COLLECT. Any result a Cypher query returns as we have seen in the previous section (nodes, edges, attributes, paths) is represented as a collection of rows. This is very similar to relational systems where we get a table-like representation where each row represents a record that matches the pattern. For example, looking at `COLLECT(ID(n))`, with `n` referring to nodes, `ID(n)` returns the identifiers of these nodes (in row format) and `COLLECT` then returns a list that contains all these identifiers.

UNWIND. `UNWIND` transforms any list into rows. For example, consider the following `UNWIND` usage:

```
MATCH p = (:User)-[]-(:Post)
UNWIND NODES(p) as n
UNWIND RELATIONSHIPS(p) as r
```

We first use the functions `NODES` and `RELATIONSHIPS` to get a list of nodes and relationships of all qualifying paths, and then transform these lists into rows. In Chapter

4 we see how this becomes helpful in rewriting queries.

WITH... `WITH` pipelines results from one query to the next. Instead of returning from the first query, `WITH` can be used to bring any result from the first query to the next. `WITH` can be combined with all previous functions, so graph data can be transformed or modified before pipelining it to the next query, much like SQL's `WITH` clause. We can also use this to specify additional constraints, such as a uniqueness constraint onto data.

DISTINCT Using the `DISTINCT` keyword, we ensure the node and relationship sets do not contain duplicates. Assume `n` to be a set of nodes and `r` to be a set of relationships that were determined by some Cypher `MATCH`. Then we can pipeline them in the following way:

```
WITH DISTINCT n, r
MATCH (a)-[rel]-(b)
WHERE a IN n AND rel IN r
RETURN ...
```

With `WITH`, we pipeline `DISTINCT n,r` to the next query, which can use the membership conditions with `n` and `r`.

Chapter 3

Language Extensions for View Creation and Usage

In a relational database, a view becomes conceptually equivalent to a table and can be re-used in queries or even to define further views. Views can be used in combination with original tables in the database, or with other views. We want a similar concept for graph databases. However, this is more challenging, as graph query results can have various types as we have seen in the previous chapter. The query language we propose is an extension of Cypher, with added features used to specify whether the query is meant to declare a view (view declaration query), or use a view (view use query). In this chapter we discuss the syntax for the language and the semantics for creating and using views. We use a stepwise approach first introducing simple views, which do not re-use previously defined views in their own definition, and only later in the chapter we extend to view declaration queries that use existing views in their definition. We define three categories that a view can belong to; a view is one of a node view, a relationship view, or a path view, depending on the result of the view declaration query. We then introduce the language extensions to use an already defined view in a view use query. Finally we introduce the concept of *scope* with respect to queries using views, and its semantics.

3.1 Creation of Simple Views

We start by discussing how a simple graph view is created. We define a *simple* view to be one where the view declaration query is entirely on the base graph. We will look later at *complex* views where the corresponding query refers not only to the base graph, but also to other views that have already been defined.

A view creation command looks as follows:

⟨*viewInit*⟩ ::= 'CREATE VIEW AS' ⟨*viewContents*⟩

⟨*viewContents*⟩ ::= ⟨*name*⟩ ⟨*scope*⟩? ⟨*query*⟩

We indicate a view creation with the `CREATE VIEW AS` keyword. A view creation must contain a name to associate the view with. This is followed by *scope* (in the following example, the `GLOBAL` keyword), which we ignore for now but we discuss in detail later, and finally the Cypher query that defines the result to which the view refers to. We refer to this part as the view declaration query. For example, suppose we want to define a view that refers to all User nodes that have a reputation greater than 1000. We do so with the following:

```
CREATE VIEW AS rep1000users
GLOBAL
MATCH (n:User)
WHERE n.reputation > 1000
RETURN n
```

3.2 Types of Views

In a relational system a view is always represented by a table, whether it is virtual or not. In a graph database system we can always think of a view as sets of elements, but we make the distinction of what kind of elements are in each set. Here we categorize views with a type depending on the type of data that is returned by the query. As we have already indicated before not all query constructs allowed in Cypher can be used for view declaration queries. Instead only queries that return nodes, or only return relationships, or only return paths are supported. Thus, we distinguish between three different types: node views, relationship views, and path views:

- A view is a **node view** if the query that defines the view only returns nodes.
- A view is a **relationship view** if the corresponding query only returns relationships (i.e, edges).
- A view is a **path view** if the corresponding query returns paths (which contain both nodes and relationships).

Furthermore, we specify restrictions on how these queries must be structured.

Node Views

A node view creation must end with a return statement which only refers to node variables in the query. The query itself may contain paths and relationships, but the return type must be only nodes. Using the working example from Chapter 2, we define a node view which returns all User nodes that have POSTED a Post which contains a "neo4j" Tag:

```
CREATE VIEW AS usersThatPostedAboutNeo4j
GLOBAL
MATCH (n:User)-[:POSTED]-(:Post)-[:HAS_TAG]-(t:Tag)
WHERE t.tagId = 'neo4j'
RETURN n
```

If we also wanted to return the corresponding posts, then we would have to provide a variable for this and return it (i.e, `..-[p:Post]-.....RETURN n,p`).

Relationship Views

As with node views, a relationship view's return statement must refer only to relationships. In the following example, we store all `POSTED` relationships in a view:

```
CREATE VIEW AS postRelationships
GLOBAL
MATCH ()-[r:POSTED]-()
RETURN r
```

Path Views

Finally, a path view may only return path variables. Take the previous view creation, which was a relationship view. Let us now store the entire path in the view - that is, for every User-Post relationship, we store the User node, the Post node, and the POSTED relationship:

```
CREATE VIEW AS postPaths
GLOBAL
MATCH p = ()-[:POSTED]-()
RETURN p
```

3.3 Simple View Usage

We now look at how a view is used. We start with simple usage, i.e., with queries that only refer to a single view and nothing else. We will later see complex usage, where a query refers to also data in the base graph, or to multiple views. View usage syntax is as follows:

⟨*viewUse*⟩ ::= 'WITH VIEWS' ⟨*name*⟩+ ⟨*scope*⟩? ⟨*query*⟩

where `name` refers to the view to be used and `query` refers to the Cypher query to be executed over the view. We refer to this as a view use query. The query itself is almost identical to a standard Cypher query, with small restrictions that we discuss shortly. We ignore "scope" for now. An example for a simple view usage is as follows:

```
WITH VIEWS rep1000users
LOCAL
MATCH (n)
```

```
    WHERE n.name = "Smith"
    RETURN n
```

In this example, we return all nodes in the view `rep1000users` defined in the previous section, where users have the name Smith. Since we only refer to one view, it is obvious that we only wish to return nodes from that view.

3.4 Complex View Usage and Scope

With simple view creations, we use view declaration queries that only refer to the base graph. With simple view usages, we refer to a single view, and restrict the query to search only within the data contained in that view. However, a query should also be able to refer to several views at once, or to a view and data from the underlying base graph, just like how a SQL query on relational data can refer to one and more views and original tables in the database. We refer to these queries as complex queries and we need the concept of scope to make clear which parts of the query belong to which data.

Scope can take two possible arguments: `LOCAL` and `GLOBAL`. `LOCAL` is used when we only care about data in the view(s) that are referenced in the query but we do not need any data from the underlying base graph for the query. That is, any node or relationship mentioned in the query must be in one of the views. In contrast, `GLOBAL` is used when any part of the query can refer to the original graph. Furthermore, we impose two restrictions on the query:

- a `LOCAL` query must be followed by a membership condition for each node or relationship in the query. That is, for each node and relationship we must indicate to which view it belongs. However, if only one view is referred to, then these conditions need not be specified. We have seen examples of this when we looked at simple view usage.
- a `GLOBAL` query refers also to the underlying graph, along with one or more existing views. It must use a membership condition for all nodes or relation-

ships that are intended to belong to a particular view. Nodes and edges that do not have such a membership condition refer to the base graph.

Indicating view membership We use the membership condition with the `IN` keyword discussed in Chapter 2 to indicate whether a node/relationship in the graph should belong to a view. The syntax is as follows: any variable in the query may be used as the left-hand side of the `IN` operator with the right-hand side being the name of the view to which it is intended to belong to.

3.4.1 Example Queries

For an example, consider the following `GLOBAL` view usage, given the same `rep1000users` view defined earlier:

```
WITH VIEWS rep1000users
GLOBAL
MATCH (n)-[:POSTED]-(p)
WHERE n IN rep1000users
RETURN n
```

In this example, the `POSTED` relationship and the nodes `p` (from Posts) refer to elements of the underlying base graph, but `n` must also be in the specific view. That is, this query selects those users of the view that have posted at least one post. As such, this query represents a join between information in the base graph and the view.

Now consider the following example, which uses two arbitrary views, `v1` and `v2`, along with the base graph to return all relationships that exist in the base graph between nodes in `v1` that are Users and nodes in `v2`:

```
WITH VIEWS v1 v2
GLOBAL
MATCH (n:User)-[r]-(m)
WHERE n IN v1 AND m IN v2
RETURN r
```

This time, both n and m are part of existing views, and the GLOBAL keyword indicates that r may be a relationship that belongs to the original graph; that is, it does not necessarily need to belong to either v1 nor v2. Additionally, we impose a label condition onto n such that n must now also be a User node. That is important, as the view v1 may possibly contain nodes that are not User nodes, but also other types of nodes (e.g. Tags or Posts).

The LOCAL keyword restricts all nodes and relationships inside the query to the view(s) specified at the start. As such, all of these nodes or relationships must be part of at least one membership condition, since LOCAL queries do not refer to the base graph. For example, we look at the following view usage which uses two views v1 and v2. In the query, we look for all node-to-node relationships that either are contained entirely in v1 or contained entirely in v2 and return the pair of nodes for each match.

```
WITH VIEWS v1 v2
LOCAL
MATCH (n)-[r]-(m)
WHERE (n IN v1 AND r IN v1 AND m IN v1)
OR (n IN v2 AND r IN v2 AND m IN v2)
RETURN n,m
```

3.5 Complex View Creation

View creation becomes equally more expressive with scope. We can create views that do not only refer to the base graph, but also to other views. Consider the following GLOBAL node view creation, given an arbitrary view exView:

```
CREATE VIEW AS exampleInit WITH VIEWS exView
GLOBAL
MATCH (n)-[]-(p)
WHERE n IN exView
RETURN p
```

In this example, we wish to use `exView` and join it with a part of the existing graph database. In order to restrict `n` to belong to `exView`, we use the membership condition. Since only `n` is subject to the condition, the unnamed relationship and the node `p` may refer to data in the original graph. Semantically, this is a view for a query which returns all nodes which have some relationship with any node who belongs to the view `exView`.

Furthermore, we can also create views that refer to other views but not the base table. Consider as an example the following `LOCAL` view creation, which returns all nodes of `exView` into a new view:

```
CREATE VIEW AS exampleInit WITH VIEWS exView
LOCAL
MATCH (n:Post)
WHERE n.score > 700
RETURN n
```

Because only one view (`exView`) is specified, then all nodes (and edges, if the query includes any) are by default associated with it. However, if another view, `exView2` were to be included alongside `exView`, then it is necessary to specify to which view each of the nodes and relationships belongs to:

```
CREATE VIEW AS exampleInit WITH VIEWS exView exView2
LOCAL MATCH (n)
WHERE n IN exView AND n IN exView2
RETURN n
```

The requirement for the membership condition may be many-to-many with regards to the number of views being referred to. In this example, we specify that `n` must belong to both `exView` and `exView2`, meaning that we return nodes that belong to both views.

Chapter 4

Non-materialized Views

In Chapter 3, we proposed and described a language extension for Cypher to be able to define and use views in graph database systems. In this section, we discuss the implementation of this language extension in form of non-materialized views.

We maintain a middleware system which accepts Cypher queries and our language extensions as input, rewrites queries as necessary, and communicates with the Neo4j database. The middleware also keeps some meta-information.

Figure 4 shows the overall architecture. Parts of the figure also refer to materialized views and we explain them in the next chapter. We can see that the middleware has a parser. Normal queries are simply forwarded to Neo4j. When view extensions are used, they are forwarded to the query rewriter. In the context of non-materialized views, all view management tasks are implemented within this query rewriter.

4.1 View Creation

For non-materialized views, upon receiving a view declaration query, we simply have to store the query text. Therefore we maintain a query table within the middleware which keeps track of all views and their declaration queries. When a view is instantiated, this information is stored inside the table.

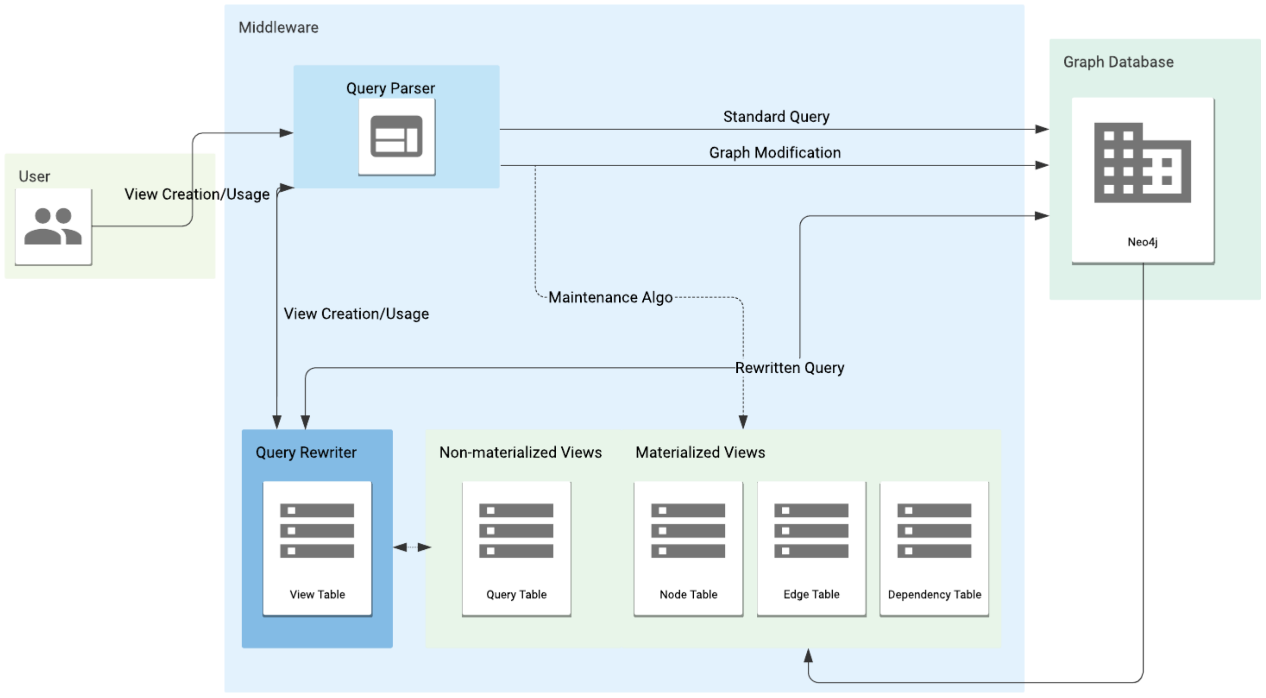

Figure 4.1: Overall Architecture. Arrows indicate communication from one component to another component.

4.2 View Usage

When receiving a view use query, the middleware has to rewrite the query so that the view declaration query is included in the rewritten query. When we re-write queries for non-materialized views, we do so by inserting sub-queries into the view use query. The sub-query, when executed, can be considered a temporary "on-the-fly" materialization of the view. We must be able to handle the rewriting of any view use query, whether it uses a node, relationship, or path view. In particular, path queries are represented in a different way within Neo4j, so we will make proper adjustments for path views. Thus, we look at the rewriting process with these concerns in mind.

To re-write view use queries we translate any of the queries following the syntax in Chapter 3 back into a query that is fully understandable by Cypher and does not include any of our language extensions. We do this in the following way: for each view referred to in the view use query, we execute a variation of the original view declaration query as a sub-query and hold the results in a set S. Recall from Chapter 2 the built-in index on node and relationship identifiers within Neo4j. We take advantage of this index while re-writing the query, such that each sub-query only returns the identifiers of the resulting nodes and edges rather than the entire result. Then, we replace all membership conditions "n IN viewName" in the view use query with ID(n) IN S.

We now illustrate this process through some examples. We first begin with a simple example using only node views. Let us have two node views, view1 and view2 with their declarations as follows:

```
CREATE VIEW as view1
MATCH (n:User) WHERE n.upvotes > 250
RETURN n

CREATE VIEW as view2
MATCH (n:User) WHERE n.reputation < 1000
RETURN n
```

Using only one view

Suppose we wish to return all nodes from view1 with an additional condition on reputation being lower than 100. We first execute the view declaration query for view1 and replace the return statement with the function, COLLECT. Recall from Chapter 2 that COLLECT is a function on any "row-like" data which aggregates all results pointed to by the original return variable into a set.

```
MATCH (n:User) WHERE n.upvotes > 250
WITH COLLECT(ID(n)) as view1
```

The WITH clause saves the set we call view1 and allows us to pipeline it to the next part of the query. The next step is to re-write the view use query. Let the following be the view use query:

```
WITH VIEWS view1 LOCAL MATCH (n:User)
```

```
WHERE n.reputation < 100 RETURN n
```

We remove the language extensions we introduced to use a view, i.e. everything in the query up to the `MATCH`, and add the membership conditions, completing the rewrite.

```
MATCH (n:User) WHERE n.reputation < 100
WHERE ID(n) IN view1
RETURN n
```

Multiple views in the same view use query

Now consider a view use query that refers to multiple views. In this case, we use one `WITH` clause per sub-query. Additionally, each time the `WITH` clause is used it must include not only the set returned by the current sub-query, but all sets from previous sub-queries. Thus given n views referred to in the view use query, we use n `WITH` clauses, and the i^{th} `WITH` clause contains i sets to be pipelined to the next. As for the view use query itself, the same rules apply as before, with no modifications.

As an example, we join `view1` and `view2` using the following view use query: `MATCH (n:User) WHERE n IN view1 AND n IN view2`. Given this view use query, we first create a sub-query to replace the first view:

```
MATCH (n:User)
WHERE n.upvotes > 250
WITH COLLECT(ID(n)) as view1
```

As we have another view to use, we use another sub-query. This time, in addition to the result set generated from the current sub-query, we also pipeline the previous results:

```
MATCH (n:User)
WHERE n.upvotes < 1000
WITH view1, COLLECT(ID(n)) as view2
```

Finally we reach the view use query itself, and our last step is to rewrite it with the appropriate set conditions:

```
MATCH (n:User)
WHERE ID(n) IN view1 AND ID(n) IN view2
RETURN n
```

While our examples only show view use queries that use node views, the same principles hold for relationship views.

Using a path view

The re-writing follows a different rule when path views are involved. We explain the steps first along an example. For that, consider the following path view declaration query:

```
CREATE VIEW AS pathView
MATCH p = (n:User)-[:POSTED]-(:Post) WHERE n.reputation < 1000
RETURN p
```

The result of this query are all the nodes and edges in the matching paths. Now assume the simple use query `MATCH (n:User) WHERE n IN pathView` that wants to have only the user nodes in the view. To allow this, we break down the path variable of the view declaration query into nodes of the path and relationships of the path. Recall that there exist two functions to group nodes and relationships for a given path: the `NODES` and `RELATIONSHIPS` functions. Using these functions we can rewrite the view use query to embed a sub-query for the view as follows:

```
MATCH p = (n:User)-[:POSTED]-(:Post) WHERE n.reputation < 1000
UNWIND NODES(p) as pathViewN
UNWIND RELATIONSHIPS(p) as pathViewR
WITH DISTINCT pathViewN, pathViewR
WITH COLLECT(ID(pathViewN)) as
pathViewNid, COLLECT(ID(pathViewR)) as pathViewRid
MATCH (n:User) WHERE n.upvotes < 100 AND ID(n) IN pathViewNid
RETURN n
```

We use the extra functions and clauses of Neo4j that were introduced in Chapter 2 to transform the path data of the view declaration query into two sets that can be fed to the view use query. The `COLLECT` and `ID` functions only work on row-like data hence the need to `UNWIND` in a previous step. In summary, the exact steps taken to rewrite a view use query that uses a path view are as follows:

1. Execute original view declaration query as a sub-query.
2. Unwind both the nodes and relationships of the resulting path variable.
3. Ensure the sets do not contain duplicate values (for space and performance optimizations).
4. Get all identifiers and collect them into a set for both the node and relationship sets.
5. Execute view use query, rewriting the membership condition.

LOCAL vs GLOBAL

In Chapter 3 we impose several restrictions to queries. `LOCAL` must include at least one membership condition for all nodes and relationships that are referenced as part of the view use query, and `GLOBAL` queries assume any node or relationship without such a condition may belong to the underlying graph. The resulting rules that we described above hold for both `LOCAL` and `GLOBAL` view use queries.

4.3 View Maintenance

Non-materialized views do not require view maintenance. Every view-use query executes the view declaration query on the latest instance of the graph. Thus, Cypher modification statements are simply forwarded to Neo4j for execution without any extra actions at the middleware.

Chapter 5

Materialized Views

We implement materialized views by executing the view declaration query once it is submitted and storing the result in the middleware. We will see that we actually do not store the full result, but only the identifiers of the nodes and edges in the result set. When a query uses the view it has to be again re-written to take advantage of the information stored in the middleware. Finally, when updates change the graph, we have to be able to determine whether that change might affect the result of a view and if yes, take appropriate actions. In the following we describe all that in more detail. Figure 4.1 shows the extra data-structures we need for materialized views.

5.1 View Creation

When a materialized view is declared, the underlying query is first executed, and the result materialized. However, we do not actually execute the original view declaration query, but instead rewrite it to only return the node and edge identifiers of the nodes and edges in the result set. That is, instead of a full materialization, we actually build something like an "index" that is then used when a query uses the view. These identifiers are stored in the middleware. For that, we maintain two hashtables: a table which maps each view to a set of node identifiers referencing the nodes that are in the view's result set, which we call the *node table*, and a corresponding table which maps each view to a set of relationship identifiers, which we call the *edge table*. Recall that we have categorized views as node views, relationship views and path views. For node views, there is no entry in the edge table, for relationship views, there is no entry in the node table. Since a path view may contain both nodes and relationships, path views will have entries in both tables.

Working Example Throughout this section we work with an example to help illustrate the data structures mentioned. Node and relationship views follow the same re-writing order. For example, a view:

```
CREATE VIEW as myView
MATCH (n:Label1)-[:REL1]-(m:Label2)-[:REL2]-(p:Label3)
WHERE {n-condition} AND {m-condition} AND {p-condition}
RETURN p
```

is rewritten to

```
MATCH (n:Label1)-[:REL1]-(m:Label2)-[:REL2]-(p:Label3)
WHERE {n-condition} AND {m-condition} AND {p-condition}
RETURN ID(p)
```

and the result is stored in the node table. A path view is handled slightly differently: a path view's query is not rewritten. Instead, the path itself is returned to the middleware and the middleware itself unwinds each path to extract the unique identifiers for the nodes and edges. These are then added to the respective tables.

5.2 View Usage

When a view is used in a query, the view use query is first parsed to find which views are referenced, and where the `IN` keyword appears. Suppose that we have the node and edge tables shown in Tables 5.1 and 5.2, and the following view use query:

```
WITH VIEWS view1 view2
GLOBAL MATCH (n:Label1) WHERE n IN view1 AND n IN view2
RETURN n
```

Table 5.1: Node Table

view	identifiers
view1	[1,3,5,7,9]
view2	[1,2,3,4,5]

Table 5.2: Edge Table

view	identifiers
view1	[1,2,3,4]
view2	[3,4,5,6]

For all instances of "`<variable> IN <viewName>`", we replace it with "`ID(<variable>) IN <identifierSet>`". Therefore the re-written use query that is forwarded to Neo4j is as follows:

```
MATCH (n:Label1) WHERE ID(n) IN [1,3,5,7,9] AND ID(n) IN [1,2,3,4,5]
RETURN n
```

Now suppose a view use query that matches on a relationship of view2 as well:

```
    WITH VIEWS view1 view2
    LOCAL MATCH (n)-[r]-(m)
    WHERE n IN view1 AND m IN view2 AND r IN view2
    RETURN n
```

Like with the previous query, we perform the same re-writing, but we re-write the membership for the relationship `r` as well:

```
    MATCH (n)-[r]-(m)
    WHERE ID(n) IN [1,3,5,7,9] AND ID(m) IN [1,2,3,4,5]
    AND ID(r) IN [1,2,3,4]
    RETURN n
```

As Neo4j manages indices for node and edge identifiers, finding the matching nodes during the execution of the rewritten view use query should be very fast. From there, the conditions in the `MATCH` phase of the use query are only executed on the relevant nodes and edges and not all nodes or edges of the base graph. Overall, the rewrite process is much simpler than with non-materialized views. However, the middleware has to store extra information.

5.3 Maintenance

When the base graph is updated, a view result might become outdated. We approach view maintenance with reevaluation rather than incremental updates. That is, the challenge is to detect with as much precision as possible whenever a view requires reevaluation. Upon a graph change, the middleware must decide for each view whether or not it should be re-evaluated. In this section we introduce the building blocks for the maintenance algorithm. The basic idea is to keep track of the relevant information in the view declaration query. We then discuss the different types of graph changes and how they affect views in different ways.

Recall that a graph change can be of three different types:

1. A node or relationship creation. A node or relationship can be created along with attributes.
2. A node or relationship deletion. A relationship can be deleted at any time while a node can only be deleted if it has no relationships with other nodes.
3. A node or relationship attribute modification: an attribute may be set to a value or removed to contain no value.

Given this, we ask ourselves how different kinds of views are affected by each of the above changes. We describe our solution in two steps. We first outline a simple solution that is based solely on the labels of nodes and relationships. At a high level, if a graph modification adds/deletes/updates a node/relationship with a label that is also referenced in the view declaration, then the view is invalidated. This is conceptually similar to a view invalidation in a relational system, where a modification on a table invalidates all views that reference that table - a very common approach in existing systems. However, this simple solution might lead to many unnecessary invalidations, or false positives. We have a false positive if we decide to invalidate a view, but the graph modification did not change the result of the view declaration query. Therefore we refine the solution to also consider the attribute conditions defined in the view declaration queries in order to reduce the number of unnecessary invalidations.

Our solution stores additional meta-information in the middleware in a dependency table, which is a hashtable where the key is a label (either label of a node or a relationship), and the value will guide us to which views to invalidate. Upon a modification, this dependency table is consulted to determine the set of views that need to be invalidated. For simplicity of description we maintain a single dependency table that keeps track of both nodes and relationships. This requires nodes and relationships to have disjoint labels. We can simply remove this restriction by having separate dependency tables for relationships and nodes.

5.3.1 Invalidation based on Labels

In this section, our invalidation mechanism only looks at nodes and relationship labels.

5.3.1.1 Dependency Table

Any Cypher query must contain at least a node. Whether the node is returned or not does not matter, but a pattern to query the graph cannot exist without a starting point for the query. Likewise, any Cypher query that contains a path must obviously also contain a relationship.

The dependency table is initially empty. Whenever a view is created, the dependency table is updated on-the-fly as follows. For each node and relationship label in the view declaration query, we look in the table to find an entry corresponding to the label, and create the entry if it does not already exist. Furthermore, for nodes and relationships in a view declaration which do not have a corresponding label, we create an entry with key node* (and respectively edge*). All values corresponding to that entry will be view names that depend on it. If any key already exists in the table, then there is no need to re-create it, however we do update the value to include the newly created view name.

Assume now as an example a graph that contains nodes with labels `Label1` and `Label2` and relationships with label `REL1`, and some nodes with no labels. We start with a simple node view as the only view in the system:

```
CREATE VIEW as V1
MATCH (n:Label1)
WHERE n.id < 50
RETURN n
```

Assuming that this is the only view in the system, this translates to two updates to the table: an entry for the view itself will be created and the entry for `Label1` will be created. For the entry corresponding to the view itself, it will become relevant if other views use that view in their declaration. Because if view `V1` needs to be invalidated, then the dependant views need to be invalidated as well. For the entry for `Label1` we keep track of the fact that `V1` depends on nodes with label `Label1`. That is, `V1` is added to the dependent sent of the entry with key `Label1`, resulting in the table shown in Table 5.3.

Table 5.3: Dependency table after `V1` is created

Label (Key)	**Dependents**
:Label1	`V1`
`V1`	none

Eventually, each entry of the table can contain many dependents. If we create another view that also uses `Label1`, then that view will also be added to the dependent list of the entry corresponding to `Label1`. If we create a view that refers to `V1`, then that view is added to the dependent list of the entry with key `V1`.

5.3.1.2 Deletes, Inserts, and Updates

What we invalidate upon a graph change depends on the type of graph change. For now, when we only consider label conditions, the way we handle deletions, insertions, and updates will be very similar.

For deletions and updates we are concerned only about the node or relationship that is being deleted. For example, with the dependency of Table 5.3 above, suppose we receive the following deletion: "`MATCH (n:Label2)-[]-(m:Label1) DELETE n`", which only deletes nodes with `Label2`. We look in the dependency table and there is no entry for `Label2`. Thus no view is invalidated. Even though a node with condition `Label1` is along the path in the query, the view `V1` is unaffected since the deletion will only potentially affect views that involve `Label2` nodes. That is, given a deletion graph change, we only look for matches in the dependency table for the node or relationship being deleted.

We go through the identical process for updates on nodes or relationships. Suppose the graph change were "`MATCH (n:Label2)-[]-(m:Label1) SET m.value = 30`". We determine that `m` refers to a node with `Label1` and find `V1` as a dependent of `Label1` in the dependency table. Thus, we need to reevaluate `V1`.

Insertions are a different case due to the structure of an insert statement. Also, there are three types of insertions that we must consider. First is the creation of a new node with no relationship: in this case, there is no "`MATCH`" statement in the graph change query, and we purely look at the newly created node and any label condition it may contain. For example, we have the following insertion which creates a node with label condition `Label1`:

```
CREATE (n:Label1)
```

Since the newly created node contains the label `Label1`, we search the dependency table and find the corresponding match: this node may qualify for `V1`, so we must re-evaluate. Note that we might invalidate views unnecessarily, namely when the view is only interested in nodes that have relationships. A newly created node does not have a relationship, thus will not qualify. However, we do not keep track of this.

The second type of insertion we may have to deal with is an insertion of a relationship between two existing nodes, and the third type of insertion is an insertion of an entirely new path. That is, some or all nodes and relationships along the path are newly created upon insertion. For these types of insertion, to refer to these existing nodes, a `MATCH` statement must precede the `CREATE` keyword.

Consider the following insertion queries: the first inserts a relationship between all pairs of nodes `n` and `m` for which `n` has label `Label1` and `m` has label `Label2` and there already exists a `REL1` relationship between the pair, and the second creates a single new node and for all nodes with label condition `Label1`, creates a new relationship between that node and the newly created node.

```
MATCH (n:Label1)-[:REL1]-(m:Label2)
CREATE (n)-[:REL2]-(m)

MATCH (n:Label1)
CREATE (n)-[:REL2]-(m:Label2)
```

We must be careful here, as for both queries, if we only consider the path indicated in the `CREATE` statement, we do not know whether `n` and `m` already belong to the existing graph (the second type), or if they are also newly created nodes (in which case this would be the third type of insertion!). However we can make certain observations: in the first query, we know that the only real change is an insertion of relationship `REL2`. `n` and `m` already exist in the graph so no entry in the dependency table corresponding to `Label1` or `Label2` will be affected. For the second query, `m` is a new node, and all entries corresponding to `Label2` will be affected. In general, we look for the set difference between the nodes and relationships referred to in the `CREATE` statement and those referred to in the `MATCH` statement. All nodes or relationships belonging to this difference must be checked in the dependency table. For example, in the first query we find `n` and `m` in both statements, but the `REL2` relationship is found in the `CREATE` statement, but not the `MATCH` statement. Therefore we look up entries in the table corresponding to `REL2`. In this particular case `REL2` does not even have an entry in the table. In the second query, this is the case with both the `REL2` relationship and the node `m`, with condition `Label2`. Therefore we look up entries in the table corresponding to `REL2` and `Label2`, and all matching entries have their dependents invalidated.

Furthermore, for each view listed, independently of the type of graph change, once a view **V** is invalidated because of a node or relationship dependency, we also invalidate all dependent views. That is, we look at the entry for view **V** in the table and also invalidate all its dependents. We do this second step recursively until there are no further dependents, in which case we will have found the original user-defined view(s) that are affected by the graph change.

There are several special cases. First, it is possible to receive a graph change that refers to nodes without a label condition. In that case, we cannot be sure which nodes will be affected. Therefore, we must reevaluate all views as all views refer to at least one node. When a graph change refers to a relationship without a label condition, we must invalidate all views that refer to at least one relationship. Furthermore, it is possible that the view itself contains nodes (resp. relationships) not specified with label conditions. Those views must be listed under the entry node* (resp. edge*). Whenever any change to a node (resp edge) occurs, all dependents of node* (resp. edge*) must be invalidated.

5.3.2 Invalidation based on Conditions

If we only consider labels, then a view may be flagged too quickly since a graph might not have many different node and relationship labels. In this case many queries that change the graph are likely to already contain many of these labels. To avoid too many false positives, we can use the property or attribute condition contained in the views as well. For example, a graph change that modifies one property of a node may not affect a view if the view has a condition on a different property of the node.

5.3.2.1 Dependency Table

To accommodate conditions, we change the structure of the dependency table. The value of an entry consists now of two lists: a dependency list and a conditions list. The dependency list contains, as before, the views affected. The conditions list is the same size as the dependents list and contains for each view in the dependents list all conditions the view has on nodes/edges with this specific label. If a view refers to a node with the specific label without condition, then we mark it with a special "none" condition. Looking now at view `V1` of the previous section, the new dependency table looks as in Table 5.4. The dependents and conditions list for label `Label1` each have one entry: for the dependents list it is `V1` and for the conditions list it is the condition `V1` has for nodes with `Label1`, namely that `id < 50`.

Overall, we may have views that contain conditions on all attributes, views that contain conditions on some attributes, or views that contain no attribute conditions at all.

Table 5.4: Table Structure

Label (Key)	**Dependents List**	**Conditions List**
V1	none	
:Label1	(1) V1	(1) id < 50

To better understand the structure of the dependency table, let us look at a variety of views. We create `V2`, a view involving two nodes and a relationship where an attribute condition exists for all nodes involved, and `V3`, a view involving two nodes and a relationship where there are two attribute conditions for one node, and no conditions for the other. View `V4` contains no label or attribute conditions at all, which will fall under the node* entry. Finally, `V5` is a more complex view with more conditions. Their view declaration queries are as follows:

```
CREATE VIEW as V2
MATCH (n:Label1)-[:REL1]-(m:Label1)
WHERE n.id > 30 AND m.id > 30
RETURN m

CREATE VIEW as V3
MATCH (n:Label1)-[:REL1]-(m:Label2)
WHERE n.id > 30 AND n.name = 'foo'
RETURN n

CREATE VIEW as V4
MATCH (n)
RETURN n

CREATE VIEW as V5
MATCH (n:Label1)-[:REL1]-(m:Label2)-[:REL1]-(p:Label1)
WHERE  n.name = 'bar' AND m.id = 35 AND m.name = 'foo'
      AND p.id < 10 AND p.name = 'baz'
RETURN n
```

Table 5.5 shows the dependency table after the creation of all of these views. For V2, we have three total graph elements to consider: two nodes and one relationship. The two nodes in this case have the exact same label and attribute conditions: "Label1" as the label condition and "id > 30" as the attribute condition. As a result, they belong to the same entry for Label1 in the dependency table and create only one element in the two lists., i.e., V2 and id > 30. Additionally, an entry for the relationship REL1 will also be created with REL1 as key with V2 and "none" in the dependency and condition lists. For V3, we have to add elements for Label1, Label2, and REL2, creating the entry for Label2 on the fly. The entry for Label1 contains a specific condition that is added to the condition list. For Label2 there is no condition, so "none" is added to the condition list. The value for the entry REL1 is also updated, and as the condition is the same as an existing one ("none"), we add V3 to the element in the dependent list that refers to the "none" condition. We then have for V4 a node with no label nor attribute conditions, which must be placed under the node* entry, as it returns all nodes no matter what label. Finally, V5 adds two elements in the dependency and condition list under the entry for Label1. Although n and p both have label Label1, they must be distinct entries under that label, as their conditions are not the same. This view also includes a node with label Label2, so we add V5 and "name = 'foo' & id = 35" as elements to Label2's dependency and condition list. Finally, for the entry REL1 we again find an existing element with the same conditions in the condition list, therefore we do not create a new element and instead add V5 to the corresponding element in the dependent list.

We also see with V3 and V5 that if there are several conditions on a node connected with an AND, we create an entry that covers all these conditions, concatenated with AND. Note that when conditions are connected with an OR, in contrast, we treat them as if there are no conditions. That is, we replace the actual conditions with "none". We will see later that if there is only one condition or all conditions are connected with ANDs, there is a reasonable potential that a graph change does not affect the view. Thus, keeping this extra information is worthwhile. However, once there are OR conditions, invalidations are much more likely, and thus, we consider it not worth keeping track of the information.

Table 5.5: Example dependency table after creating V2, V3, V4, and V5

Label (Key)	**Dependents List**	**Conditions List**
:Label1	(1) V1 (2) V2 (3) V3 (4) V5 (5) V5	(1) id < 50 (2) id > 30 (3) id > 30 & name = 'foo' (4) name = 'bar' (5) id < 10 & name = 'baz'
:Label2	(1) V5	(1) id = 35 & name = 'foo'
:REL1	(1) V2, V3, V5	(1) none
V1	none	
V2	none	
V3	none	
V4	none	
V5	none	
node*	(1) V4	(1) none

5.3.2.2 Inserts, deletions, updates

Given the dependency table shown in Table 5.5, we now look at examples of graph changes and how each view may be affected, and the process used to detect it. We will first start with all cases for deletions, then insertions, and then updates.

1. Node deletion with no conditions

In this simple case, we delete a node with no conditions at all:

```
MATCH (n)
DELETE n
```

This is the special case where all views must be reevaluated, as each view depends on at least one node, regardless of their label condition. In this case, we do not even need to look at the dependency table as long as we have somewhere a list of all defined views.

2. Node deletion with only a label condition

For this, we consider the following node deletion where the deletion query contains only a label condition and no attribute conditions.

```
MATCH (n:Label2)
DELETE n
```

Following the same steps as in the previous section we look at the entry for `Label2` in the dependency table. This is the special case where all entries corresponding to `Label2` must be reevaluated, regardless of the attribute conditions as the delete affects all nodes of `Label2`. In this case, all elements in the dependency list under `Label2` will be marked, leading to the invalidation of `V3` and `V5`. Furthermore, `V4` is also invalidated since the graph change affects nodes, and `V4` considers nodes without consideration of labels. We detect `V4` by looking at the entry node* in the dependency table. `V1` and `V2` are not invalidated because they do not reference `Label2`.

3. Node deletion with one attribute and label condition

Let us now have a look at deletions for nodes with a specific label and a single attribute condition:

```
MATCH (a:Label1)
WHERE a.id = 1
DELETE a
```

The idea here is that we now have a closer look at the conditions list for entry `Label1` and only invalidate when the condition in the delete and the condition of the view overlap, meaning the set of nodes deleted possibly intersects with the result set returned by the condition found in the view declaration. If they do not intersect, then there is no need to invalidate the view as its result will not change. For each element in the conditions list, we first look whether it is on the same condition as the delete statement, i.e., on `id`. If yes, then we check whether the conditions overlap.

In this case, there are two elements where the condition in the delete and view have common attributes. For those we check whether the conditions overlap and if yes we invalidate the corresponding views. The first is the one with dependent `V1` and condition `id < 50`, because if a node with "`id = 1`" exists, then it is in the result set of `V1` and the delete statement will delete it. Note that it might be possible that no node with `id = 1` exists, and then `V1` would not be affected. But by only looking at the conditions we cannot see that, and therefore we talk about a possible overlap and invalidate. `V1` is marked for invalidation, as well as all its dependents (in this particular case, none). The second element that is affected is the final element in the lists with dependent `V5` and conditions `id < 10 & name = 'baz'`, because if a node with "`id = 1`" exists, then it could also be in the result set of `V5`. We invalidate `V5` and its dependents (none) due to the possibility of overlap within the graph database.

In addition to this, the fourth element in the condition list indicates `name = 'bar'`, that is, it is on a different attribute and does not have a condition on `id`. In this case, the result sets might also overlap, and we need to invalidate. Note that if the condition were "none" (no example shown) then we would also need to invalidate because of overlap.

Furthermore, we have the node* entry in the table: this entry is affected as long as the graph change affects any node. This special case is checked separately, and since the graph change does affect nodes, `V4` is also marked as invalidated as a consequence.

Note that V2 is not affected by this graph change, which is reflected in this process as well. As the second element in the condition list for `Label1` was the condition "`id > 30`", it is guaranteed to have no overlap with the condition in the delete, and thus `V2` is not re-evaluated.

A special case is an element in the conditions list with several conditions concatenated with an `AND`, in our example the third and fifth condition. As long as one of the conditions creates an empty intersection with the conditions in the delete, invalidation is not needed. The third element in the condition list does not cause invalidation as `id > 30` does not intersect with `id > 1`. We don't care about the other condition (`name = 'foo'`) as it is sufficient to find one condition in the `AND` clause that is not fulfilled. On the other hand, the fifth element has potential overlap: as `id < 10` overlaps with `id = 1`, the only way to avoid invalidation is if there is an empty intersection on the `name` attribute. However, the delete statement contains no conditions on it, so a node which has `id = 1` and `name = 'baz'` might be affected. Thus `V5` must be invalidated.

4. Node deletion with label condition and multiple attribute conditions

We move into cases where more than one attribute condition exists within the deletion query - these can either be joined with AND clauses or OR clauses. For the former, consider the following example:

```
MATCH (n:Label1)
WHERE n.id < 10 AND n.name = 'foo'
DELETE n
```

With AND clauses, we bundle the conditions for each node together. In this example, we have nodes with `Label1` with both "`id < 10`" and "`name = 'foo'`". We now search the dependency table for `Label1`, and look in the condition list for conditions that overlap with the conditions in the change. As with the case with a node deletion with a single attribute condition, we consider a change condition to overlap with the view condition if the sets produced by the conditions are potentially non-disjoint. Only then invalidation is needed. For example, consider the third element in the condition list. This contains the conditions "`id > 30`" and "`name = 'foo'`". Although the condition on the attribute `name` matches with the one in the change, the condition on `id` does not: `id > 30` and `id < 10` guarantees to produce disjoint sets. Therefore invalidation is not triggered due to this condition.

However, it is again possible that the graph change contains conditions on attributes that a view does not contain conditions on. For instance, we look at both the first and second conditions for entry `Label1`. The first has a potential overlap, and thus triggers invalidation. The second does not overlap.

From here, we can derive the general formula to determine overlap. Let C_{gc} be the conditions in the delete statement for a node with label L (all concatenated with `AND`), and let C_{vc} be the conditions listed in one of the elements in the condition list for entry L (again only with ANDs). If C_{gc} and C_{vc} have a condition on at least one common attribute (e.g. id) and the conditions don't overlap, then we know there is no overlap overall and no invalidation is needed. Otherwise, that is, when they overlap on all attributes they have in common, or if they don't have an attribute in common at all, then they potentially overlap, and the corresponding view and dependents must be invalidated.

Handling OR clauses is more difficult. A graph change that contains OR clauses on its attribute conditions can be split and treated as separate graph changes - an extra graph change per OR clause. For instance, the graph change

```
MATCH (n:Label1)
WHERE n.id < 10 OR n.name = 'foo'
DELETE n
```

can be separated into the following two graph changes, and then processed individually.

```
MATCH (n:Label1)
WHERE n.id < 10
DELETE n

MATCH (n:Label1)
WHERE n.name = 'foo'
DELETE n
```

As a result any condition in the condition list that contains an attribute condition on only name or only id will lead to invalidation through one of the two deletes, which is the correct behavior. The only type of view that would not be affected would be one which contains an attribute condition on both of these attributes, concatenated with AND, and whose conditions do not overlap with either of these.

As shown in the example a created view might contain multiple conditions on multiple nodes or relationships. In our examples, this is the case for V2, V3, and V5 whereby V5 has even two entries for a single node label. In those cases, as soon as a graph change leads to an invalidation due to one of the conditions, the view must be reevaluated.

For instance, given the graph change:

```
MATCH (n:Label1)
WHERE n.name = 'bar' AND id = 35
DELETE n
```

Looking at the entry for `Label1` the fourth condition `name = 'bar` leads to invalidation but the fifth condition does not. Still, `V5` needs to be reevaluated.

5. Relationship deletions

We treat relationships identically to nodes when considering their entries in the dependency table and graph changes. Each relationship gets a corresponding entry in the table, and that entry also has dependency and condition lists. There are, however, additional aspects that can be considered. For example the left-hand and right-hand sides (that is, the nodes to the left and right) of a relationship give context to when the relationship should be under consideration for invalidation. However due to the complexity of these cases, we forego considering these possibilities.

6. Node insertions without label

If a node without a label is inserted then we only look under the node* entry as we compare conditions. The steps that we take as we make these comparisons are identical to the ones already described in the previous cases.

7. Node insertions with labels

Consider the following node insertion with a label which also includes certain attributes on the node:

```
CREATE (n:Label1 {id:3, name:'foo'} )
```

For these insertions, we may have one of two possibilities:

1. The insertion includes an attribute for which a view contains an attribute condition on.

2. The insertion does not include any attributes for which a view contains an attribute condition on.

For the first possibility, this node insertion may affect the view in question. Thus we must check within the dependency table and compare the inserted attributes with the conditions. We use the same deciding criteria as described before: as long as there is one attribute where the conditions do not overlap, then we do not need to invalidate the associated view. Using the same example dependency table (Table 5.4), this case occurs for the second and third elements because the condition `id > 30` does not intersect with `id = 3`, and for the fourth and fifth elements because of the name attribute . On the other hand, we see that this can overlap with the condition `id < 50` of the first element, thus view `V1` becomes invalidated. Finally, following the same rule described in the previous section with basic insertions, all dependents under the `node*` entry must also be reevaluated, since it is a node that is being inserted into the graph. Thus, view `V4` is invalidated.

For the second possibility, there are two sub-cases of possibilities. First, the view may have zero attribute conditions - in this case we invalidate it because the new node might now belong to the result set of the views. Second, the view may contain other attribute conditions that are not on the same attributes that were included during the insertion - in this case, we may safely assume that the view in question does not require reevaluation, as a node that does not have an attribute that is part of a condition in a view can not fulfill the condition.

8. Relationship and path insertions

Recall from Section 5.3.1.2 that a path insertion can include already existing nodes or relationships within the `CREATE` block. Therefore, we use again the same process to find the nodes and relationships that are truly inserted and do not already exist: we take the set of variables specified in the `CREATE` statement and the set of variables in the `MATCH` statement; their set difference will be all newly inserted nodes and relationships. From here, we can treat relationship insertions in the exact same way as node insertions considering attribute overlap, as we have just discussed in the previous subsection: the same possibilities can occur for both the node and relationships, and we deal with them in the exact same way.

9. Node and relationship updates

With updates we are now concerned with the new values for attributes that are updated. As a reminder, there are two parts to an update query that we must consider, shown below in the example update:

```
MATCH (n:Label1)
WHERE n.id = 1
SET n.name = 'foo'
```

The first part to the update is the `MATCH` query, where we identify the graph portion which contains the node (or relationship) that we wish to update. For simplification we assume that nodes and relationships have their label conditions specified in the query. The second part is the update itself, which follows the `SET` keyword. We are interested in anything that is mentioned in the second part, but we need the first part in order to associate all variables with the proper entries in the dependency table. We continue with two phases of checking: after we find the entries for a label, we look for any conditions in the `MATCH` part of the graph change and compare to the conditions in the condition list: the overlap considerations follow the exact same steps as discussed before. This phase looks for all elements which may possibly be affected by the change. For all elements with possible overlap, we perform the second phase which checks whether the updated attribute will actually affect the condition in the condition list. For example, "`id = 1`" is the condition in the `MATCH` part of the above change, which overlaps with the first condition `id < 50`. However setting "`name = 'foo'`" will not affect the condition because it does not care about the particular value of the name. However, we also have the fifth element with condition `id < 10` and `name = 'baz'`. Again, `id = 1` and `id < 10` overlap and the view has now also a condition on the attribute to be updated. In this case we must invalidate. There might have been a node with `id = 1` and `name = 'baz'` in the view and after the change, it should no longer be in the result set. Note that if the view's condition were `id = 10` and `name = 'foo'`, then invalidation is also needed because that node might now be added to the result of the view. Therefore, our criteria becomes the following: given that the check passes the first phase, as long as the attribute which is updated is part of the condition for an element in the entry of the dependency table, it leads to invalidation of the dependent views.

Chapter 6

Evaluation

In this chapter we present a performance analysis of the view management solutions proposed in the previous chapters. The objectives of the tests are:

1. To evaluate the feasibility of our solution in a realistic environment. The use of our system should be practical in a real-world scenario, and we test to ensure that our system provides an overall benefit compared to not using views.

2. To compare the non-materialized and the materialized view methods. In particular we are interested in looking at common use cases where these methods will excel or under-perform.

3. To identify what variables can speed up or slow down performance, for all three steps of a given approach (view creation, usage, and maintenance).

In the following we first present the setup, and then discuss the performance of view creation, view usage, and view maintenance.

6.1 Test Setup

While there exist popular benchmarks for relational database systems such as the TPC-H benchmark [22], we are not aware of a standard benchmark for graph database systems. In order to fairly evaluate our system with sufficient breadth, we create our own graph database and a set of queries that are meant to encapsulate a wide range of possible use cases on this graph database. We use the StackOverflow database mentioned throughout the thesis as our working example, and we have created a set of view declarations and view use queries, of varying complexity. The details of these queries are provided in the following sections. Furthermore we create 3 instances of the database and vary their sizes as follows:

1. The large database contains 11.4 million nodes and 24 million relationships.
2. The medium database contains 9 million nodes and 17.7 million relationships.
3. The small database contains 5.6 million nodes and 10.2 million relationships.

The complete database is from the data described in [18]. We scaled this down to our three databases which are smaller by truncating the csv files which contain the data.

For the hardware, all tests are run on a machine equipped with an AMD 1700 processor with 16GB of RAM. As for the software, the databases are running on Neo4j 4.0.4 with Java version 11.0.7 for the middleware, and with Windows 10 as the operating system. All values in this chapter are taken as an average of 5 runs after an initial warm-up run. These tests are also run in a random order, in order to prevent cached values, as such caching would unlikely occur in real-life environments. Finally, we note that there are no indexes on the database for any properties, except for the natural index on node and relationship identifiers.

6.2 View Creation

In this section we look at view creation performance. For that purpose we have designed a set of 20 views. The complexity of the view declaration queries ranges from simple `MATCH (n)` with a single condition, to queries with multiple conditions on one or more attributes. Some are node queries, and others are path queries. The details are summarized in Table 6.1, listing views V1, V2, V3, etc., together with their view declaration queries.

6.2.1 Materialized Views

When using the materialized view approach, the middleware executes the view declaration query but only in a way that the node and edge identifiers of the result set are returned, and stores these identifiers within the middleware.

6.2.1.1 View Creation time

The overhead for this materialization can be measured by two performance metrics that quantitatively reflect the complexity of the queries: the time needed to execute the view declaration query and the size of the result set that must be stored in the middleware. We have measured the time taken to materialize each view in each of the three differently-sized databases, shown in Table 6.2. The values reflected in this table correspond to the execution time of the queries corresponding to each view. We also measured the time needed to store all identifiers within the tables in the middleware, but it was insignificant.

Table 6.1: View Declaration Queries

	Declaration	Details
V1	MATCH (n:Post) WHERE n.score >350 RETURN n	Query involves a node with a filter on a single attribute.
V2	MATCH (n:Post) WHERE n.score <800 AND n.score >350 RETURN n	Query involves a node with a range filter on a single attribute.
V3	MATCH (n:User) WHERE n.upvotes>1000 RETURN n	Query involves a node with a filter on a different attribute.
V4	MATCH (n:User) WHERE n.reputation >90000 RETURN n	Query involves a node with a filter on a different attribute.
V5	MATCH (n:Tag) WHERE n.tagId = ’java’ OR n.tagId = ’html’ RETURN n	Query involves a node with an OR clause on a single attribute.
V6	MATCH p = (n:User)-[:POSTED]-(po:Post) WHERE n.reputation <500 RETURN p	Query involving a path with a single condition on a single attribute.
V7	MATCH (n:Post)-[:PARENT_OF]-(m:Post) WHERE m.score >100 AND m.score <600 RETURN n	Query involves a path with a range filter on a single attribute
V8	MATCH (n:User)-[:POSTED]-(po:Post)-[:PARENT_OF]-(po2:Post) WHERE n.upvotes >800 AND po.comments >10 RETURN po2	Query involving a path with conditions on several attributes.
V9	MATCH (n:User)-[:POSTED]-(p:Post) WHERE n.userId = 19 RETURN p	One-hop query centered around a single node.
V10	MATCH (betterPost:Post)-[:PARENT_OF]-(worstPost:Post) WHERE worstPost.score <10 AND betterPost.score >worstPost.score * 10 RETURN betterPost	Query involving a path with complex condition on an attribute.
V11	MATCH (n:User)-[:POSTED]-(p:Post) WHERE n.upvotes >1000 OR p.score >350 RETURN p	Query involves a path with more than one condition on different attributes.
V12	MATCH (p1:Post)-[:HAS_TAG]-(t:Tag) WITH p1, COUNT(t) as numberOfTags WHERE numberOfTags >20 RETURN p1	Query involving aggregate functions and pipelined results, with a condition on the aggregation.
V13	MATCH (n:User)-[:POSTED]-(p:Post)-[:HAS_TAG]-(t:Tag) WITH n,t, COUNT(*) as numberOfPosts WITH n, COLLECT(t) as tags, COLLECT(numberOfPosts) as counts, MAX(numberOfPosts) as highestTagCount WITH n,highestTagCount, [i IN range(0, size(counts)-1) | CASE WHEN counts[i] = highestTagCount THEN tags[i] ELSE NULL END] AS finalVal RETURN n,finalVal	Query involving many functions and aggregations.
V14	MATCH (n:User)-[:POSTED]-(p:Post)-[:PARENT_OF]-(p2:Post)-[:POSTED]-(m:User) WHERE n.userId<50 RETURN m	Query with a 3-hop path.
V15	MATCH (p:Post)-[:HAS_TAG]-(t:Tag) WHERE t.tagId=’html’ RETURN p	Query centered around a single node (tag whose tagId=’html’).
V16	MATCH p =(n:User)-[:POSTED]-(p:Post) WHERE n.reputation<50000 RETURN p	Similar to V6, but with a higher selectivity.
V17	MATCH (n:User)-[:POSTED]-(p:Post)-[:PARENT_OF]-(p2:Post)-[:POSTED]-(m:User) WHERE n.userId<m.userId AND n.reputation>m.reputation RETURN m	Similar to V14 but with a more complex set of conditions.
V18	MATCH (n:User)-[:POSTED]-(p:Post)-[:PARENT_OF]-(p2:Post) WHERE n.upvotes>0 AND p.comments>10 AND p2.comments<10 RETURN p2	Two-hop query with high selectivity at each node.
V19	MATCH (n:Post)-[:PARENT_OF]-(m:Post) WHERE n.score =15 AND m.score = 50 RETURN n	One-hop query with very low selectivity at both nodes.
V20	MATCH (n:Post)-[:HAS_TAG]-(t) WHERE n.postId = ’1065111’ RETURN t	Similar to V15, but centered on a Post node instead.

Table 6.2: Time taken (ms) to execute view declaration queries

View	Small	Medium	Large
V1	2376	3916	7088
V2	8358	9370	11895
V3	37	149	487
V4	39	102	536
V5	81	89	147
V6	63	785	34144
V7	14136	133694	247722
V8	161064	343122	1160022
V9	32	93	442
V10	54541	290013	1225093
V11	5837	647623	1151968
V12	6868	11317	15674
V13	3645	249845	624538
V14	146	1429	10356
V15	3004	8427	20088
V16	2005	5178	10573
V17	32552	101127	202168
V18	4123	579645	1346992
V19	2548	14311	65218
V20	3000	5004	8238

We can notice immediately that some queries take much less time than others to execute, even when they appear to be equally complex. For some of these, there is no clear reason why. For instance, V6 is a one-hop path query, but executes almost instantly on even the largest database while V7, another query that only contains a one-hop path, takes much longer. On the other hand, we can explain other results, such as V8 taking an order of magnitude longer than V7, independent of the database size. The reason for this is likely due to the length of the path involved in each query; the longer the path, the more "branching out" must be done for each potential match on the pattern. Thus we expect a view involving a two-hop path (in the case of V8) to take significantly longer than one with a one-hop path (V7). With all of these different types of queries, we are confident that we encompass many possible use-cases for the system.

Additionally, we notice that the creation time increases as the size of the database increases. In general, this increase tends to be exponential, which makes sense especially for the views for which the query matches onto a path.

6.2.1.2 View Size

Table 6.3 shows the size of the result set returned by the view declaration queries. We can see that there is a big difference in result sizes. The smallest result sets are those that first pre-select a single node as the start base of the query (V9, V20) or those which simply have very restrictive selections (V5). Otherwise, it is not straightforward to determine from the query itself whether its result set is small or large.

6.2.1.3 Categorization

In order to determine whether view creation time and/or view size affect the performance of a query which uses a view, we create four categories of view declaration queries. We set a cut-off threshold over which we consider a query to be fast or slow to materialize, and we set a cut-off threshold to decide whether a result set size is small or large. For the large database, we select a cut-off of 100,000 records and 100,000 milliseconds. That is, any view that takes longer than 100,000 milliseconds to create is considered a slow view, otherwise it is fast. And any view that returns more than 100,000 records is a large view, otherwise it is small. Depending on the cut-off that we choose, a view may be categorized differently, thus it is likely that these cut-offs should be different for different database sizes. However if we scale down the same views appropriately and take smaller cut-offs for smaller databases (such that the views remain in the same category), then we should expect the relative performance to remain consistent. Table 6.4 shows the category of each view on the large database. Most small views have fast execution times and many large views take long to execute, but there are views where this correlation does not hold.

Table 6.3: View Sizes for V1-V20 in each database

View	Large	Medium	Small
V1	14847	13661	11388
V2	10733	9799	8012
V3	6570	5139	2341
V4	29799	17503	4929
V5	2	2	2
V6	44036	36213	8605
V7	169677	150683	118846
V8	31176	24240	11465
V9	14	14	13
V10	2776312	2210293	1406036
V11	2152189	1674982	766993
V12	10736017	8026777	4856056
V13	3698662	2467985	793076
V14	6778	5784	4256
V15	443357	106750	58015
V16	4408322	1909170	730811
V17	1677752	715684	214768
V18	95511	38221	16752
V19	101	48	37
V20	3	3	3

Table 6.4: Categorization of the view declaration queries on the large database.

Small view, Slow time	Small view, Fast time	Large view, Slow time	Large view, Fast time
V8, V18	V1, V2, V3, V4 V5, V6, V9, V14 V19, V20	V7, V10, V11, V13 V17	V12, V15, V16

6.2.2 Non-Materialized Views

We have no measurements for the non-materialized method since a non-materialized method has no such view creation time, since there is nothing to persist. Therefore we may effectively treat the 'creation time' of these views as zero since there is no work to be done at this step, apart from storing the query text in the middleware.

6.3 View Usage

In this section we look at the performance of when views are used in further queries. We discuss the performance of the materialized views versus the non-materialized views, and compare them with a baseline which we introduce in Section 6.3.1.
We have a total of three sets of experiments; the first aims to determine whether the category of the used views affects performance. For this, we write queries that use the views of the 4 categories we defined, i.e., using views that we define as small or large, and slow to materialize or fast to materialize. For these queries, we also aim to determine whether performance is consistent across queries that use the same view. The second set of experiments targets special case queries that display behavior that can be explained by different factors, and for this we write a few special queries for which we expect special or interesting results. Finally, our third set of experiments is to ensure that we can expect consistent results as the size of the database changes. For this we dig deeper into a subset of the queries used for the

previous two.
Overall, we also compare the view-based approaches to the baseline that does not use views.

6.3.1 Introducing a baseline

The baseline to which we compare the materialized and non-materialized approaches does not use any views, but represents a user who writes a query directly on the base graph that has the same result as a query which uses a view - we call these equivalent queries "baseline queries". Note that they are different from the queries that are created by our rewrite mechanism for non-materialized views. Recall from Chapter 3 that our middleware automatically rewrites queries that use views. For instance, assume the following view use query:

```
WITH VIEWS V1 V8
LOCAL MATCH (n) WHERE n IN V1 AND n IN V8
RETURN n
```

The rewritten non-materialized query would be the following, after following the steps outlined in Chapter 4:

```
MATCH (n:Post)
WHERE n.score > 350
WITH COLLECT(ID(n)) as V1

MATCH (n:User)-[:POSTED]-(po:Post)-[:PARENT_OF]-(po2:Post)
WHERE n.upvotes > 800 AND po.comments > 10
WITH V1, COLLECT(ID(po2)) as V8

MATCH (n) WHERE ID(n) IN V1 AND ID(n) IN V8
RETURN n
```

This, however, may not be the most optimal, as we perform two separate queries for the `:Post` nodes that are common to both nodes, plus a final query to join the results of the previous two. A more natural query, which returns the same values while both reducing the number of sub-queries needed to execute and allowing us to check each condition in fewer steps would be the following (recall that V8 returns po2, so the condition `score > 350` must be on po2 and not po):

```
MATCH (n:User)-[:POSTED]-(po:Post)-[:PARENT_OF]-(po2:Post)
WHERE n.upvotes > 800 AND po.comments > 10 AND po2.score > 350
RETURN n
```

The query is equivalent because `n IN V1` and `n IN V8` correspond to `:Post` labeled nodes, so we may join all conditions regarding that particular node together. Additionally, if the two views both contain paths then we may also join these paths together into a single path, again allowing us to reduce the number of sub-queries, though this may increase the run-time if the resulting path is many hops long. We believe these queries are written in the same way as how a developer would have written it, which would be different from the non-materialized method that follows a systematic and automated approach to rewrite queries.

6.3.2 View Use Queries

We have two types of view use queries; categorized use queries and special use queries. They are detailed in Table 6.5. For the categorized queries, recall that we created four categories of view declaration queries, based on their result size and creation time. To find trends from these two characteristics, we write 16 queries which use views in each of these categories, and denote them as $U^i_{V_a,...,V_z}$, where $V_a,...,V_z$ corresponds to the view(s) used, and i indicates that it is the i^{th} use query which uses views $V_a,...,V_z$. For example, $U^1_{V_{15}}$ and $U^2_{V_{15}}$ are two different queries that both use V15. We categorize these use queries by the creation time and size of the underlying views. We look for trends within each category when we compare the performance of these queries between the materialized, non-materialized, and baseline methods. Table 6.6 summarizes the creation times and size of the views for views involved in the use queries that we evaluate in the next section.

In addition to the categorized queries, we have two additional special queries for which we expect interesting results, due to the complexity of the use query itself. We call these two queries U_S^1 and U_S^2.

Note that for the categorized queries, our focus is not the complexity of the use queries, but rather the complexity of the underlying views used, as the former is much more difficult to quantify. Thus, for these queries it does not matter that many of them are simple and only recall the view that is used. On the other hand, U_S^1 and U_S^2 are chosen due to the complexity of the use query itself, thus we expect special behavior that cannot be explained solely through categorization.

Table 6.5: Use Query Details.

Use Query	Query Body
$U^1_{V_1,V_8}$	WITH VIEWS V1 V8 LOCAL MATCH (n) WHERE n IN V1 AND n IN V8 RETURN n
$U^1_{V_4}$	WITH VIEWS V4 LOCAL MATCH (n) RETURN n
$U^1_{V_5}$	WITH VIEWS V5 LOCAL MATCH (n) RETURN n
$U^2_{V_5}$	WITH VIEWS V5 GLOBAL MATCH (n)-[:HAS_TAG]-(p:Post) WHERE n IN V5 RETURN p
$U^1_{V_6}$	WITH VIEWS V6 LOCAL MATCH (n) RETURN n
$U^1_{V_7}$	WITH VIEWS V7 LOCAL MATCH (n) RETURN n
$U^1_{V_8}$	WITH VIEWS V8 LOCAL MATCH (n) RETURN n
$U^1_{V_{15}}$	WITH VIEWS V15 LOCAL MATCH (n) RETURN n
$U^2_{V_{15}}$	WITH VIEWS V15 GLOBAL MATCH (n)-[:POSTED]-(m:User) WHERE n IN V15 RETURN m
$U^1_{V_{16}}$	WITH VIEWS V16 LOCAL MATCH (n) RETURN n
$U^1_{V_{17}}$	WITH VIEWS V17 LOCAL MATCH (n) RETURN n
$U^1_{V_{18}}$	WITH VIEWS V18 LOCAL MATCH (n) RETURN n
$U^1_{V_{18},V_{19}}$	WITH VIEWS V18 V19 LOCAL MATCH (n) WHERE n IN V18 AND n IN V19 RETURN n
$U^1_{V_{19}}$	WITH VIEWS V19 LOCAL MATCH (n) RETURN n
$U^1_{V_{20}}$	WITH VIEWS V20 LOCAL MATCH (n) RETURN n
$U^2_{V_{20}}$	WITH VIEWS V20 GLOBAL MATCH (n)-[:HAS_TAG]-(p:Post) WHERE n IN V20 RETURN p

U_S^1	WITH VIEWS V8 V6 GLOBAL MATCH (p1:Post)-[:HAS_TAG]-(t:Tag)-[:HAS_TAG]-(p2:Post) WHERE p1 IN V8 AND p2 IN V6 RETURN t
U_S^2	WITH VIEWS V7 V3 GLOBAL MATCH (n:User)-[:POSTED]-(p:Post) WHERE n IN V3 AND p IN V7 AND n.reputation >2*p.score OR n.downvotes <p.score RETURN n

Table 6.6: Creation time, size, and category of the views used in our view use queries

View	Creation Time (ms)	Size	Type
V1	7088	14847	Fast/Small
V4	536	29799	Fast/Small
V5	148	2	Fast/Small
V6	34144	44036	Fast/Small
V7	247722	169677	Slow/Large
V8	1160022	31176	Slow/Small
V15	20088	443357	Fast/Large
V16	10573	4408322	Fast/Large
V17	202168	1677752	Slow/Large
V18	1346992	95511	Slow/Small
V19	65218	101	Fast/Small
V20	8238	3	Fast/Small

6.3.3 Expectations

Recall that for the materialized method, we retrieve all identifiers from the middleware and rewrite the incoming query to refer to those identifiers. In contrast, the non-materialized method re-executes each view referred to as a sub-query. Therefore, we expect the materialized method to outperform the non-materialized method as the non-materialized method has to execute an additional sub-query, while the materialized method can rely on Neo4j's indices on node and relationship identifiers for fast referral to relevant nodes and relationships. Also, we expect the non-materialized method to take at least as long as the creation times of the underlying views as seen in Table 6.2.

We also expect the relative performance of the materialized views to be better if the underlying view(s) used take longer to create. That is, a query which uses a view with a large time in Table 6.6 will benefit from materialization more than a query which uses a view with a short time. This is because the materialization will avoid the long re-execution of the underlying query associated with the view. Similarly, we would like to find out if the size of the result set of the underlying view affects the performance of a query which uses the view.

Finally, we are interested in the consistency of the performance when using the same view. If a query uses a view V and benefits from materialization, will any query that also uses V also benefit, regardless of its complexity? We have no hypothesis for this, but our evaluation will attempt to locate this trend as well, if it exists.

6.3.4 Performance Results

For the categorized queries, we run all of them on the large database and record their results in Table 6.7 below. We separate these into individual tables for each category, and show the run-time of the queries in milliseconds for the materialized, non-materialized, and baseline approaches. Some of these queries did not terminate, in which case we write "[too high]" to indicate it. Note we categorize $U^1_{V_1,V8}$ as a query which uses a small view with a slow creation time even though it uses V_1 because it still uses V_8. All values are in milliseconds.

Table 6.7: Performance on the large database for each of the three approaches.

Small view, slow creation time	Materialized	Non-materialized	Baseline
$U^1_{V_1,V_8}$	3270	186325	1284654
$U^1_{V_8}$	8388	205039	205370
$U^1_{V_{18},V_{19}}$	1958	34335	659781
$U^1_{V_{19}}$	17	11811	95846

Small view, fast creation time	Materialized	Non-materialized	Baseline
$U^1_{V_4}$	16392	2701	2409
$U^1_{V_5}$	4061	227	163
$U^2_{V_5}$	1093283	1108458	1113844
$U^1_{V_6}$	52059	47390	44786
$U^1_{V_{18}}$	27281	195381	1684268
$U^1_{V_{20}}$	13	9121	9540
$U^2_{V_{20}}$	49972	24006	215

Large view, slow creation time	Materialized	Non-materialized	Baseline
$U^1_{V_7}$	112674	219116	203660
$U^1_{V_{17}}$	36569	[too high]	6679643

Large view, fast creation time	Materialized	Non-materialized	Baseline
$U^1_{V_{15}}$	84105	103167	106144
$U^1_{V_{16}}$	40946621	[too high]	[too high]
$U^2_{V_{15}}$	1213140	20323	59894

In order to understand the impact of database size a bit better, we have run a subset of the use queries ($U^1_{V_1,V_8}$, $U^1_{V_5}$, $U^1_{V_8}$, $U^2_{V_{15}}$, $U^1_{V_{18},V_{19}}$, and U^2_S) on all three database sizes. The results are shown in Figure 6.1.

6.3.5 Analysis for Materialized Approach

6.3.5.1 Categorized Queries

We see immediately that queries which use views that take a long time to materialize benefit significantly with respect to both the non-materialized approach and the baseline queries. On the other hand, queries that use views which are fast to materialize do not seem to have a trend in either direction; for some, materialization benefits the performance and for others it is worse. This result does not seem to depend on the size of the view either, as we see materialization can either benefit fast large views ($U^1_{V_{15}}$, $U^1_{V_{16}}$) or hinder them ($U^2_{V_{15}}$). Similarly, we see that materialization can benefit small fast views ($U^1_{V_{18}}$, $U^1_{V_{20}}$) and also hinder them as well ($U^1_{V_4}$, $U^1_{V_5}$). This means that for fast views, there must be more variables that contribute to the effectivity of materialization.

6.3.5.2 Special Queries

Upon executing U^1_S we find that the query fails to terminate. The execution time is too long even on the smallest database, and does not complete with any of the three methods. Upon investigating this, we find an interesting detail about the execution plan for this query. We find that in Neo4j, subsequent NodeByIdSeek searches after the first one become less efficient when the query involves paths. The execution planned by Neo4j for the queries U^1_S and U^2_S can be seen in Figure 6.2. Recall that query U^1_S contains two membership conditions, `p1 IN V8` and `p2 IN V6`. These are rewritten into `ID(p1) IN {V8}` and `ID(p2) IN {V6}`, where `{V8}` and `{V6}` indicate the sets returned by the views `V8` and `V6`. Additionally, the use query contains a two-hop path (p1→t→p2). When we check the execution plan for this query, we do find two NodeByIdSeek steps; one for each view. However Neo4j decides that instead of exploring all two-hop paths between `p1` and `p2`, it is more efficient to ex-

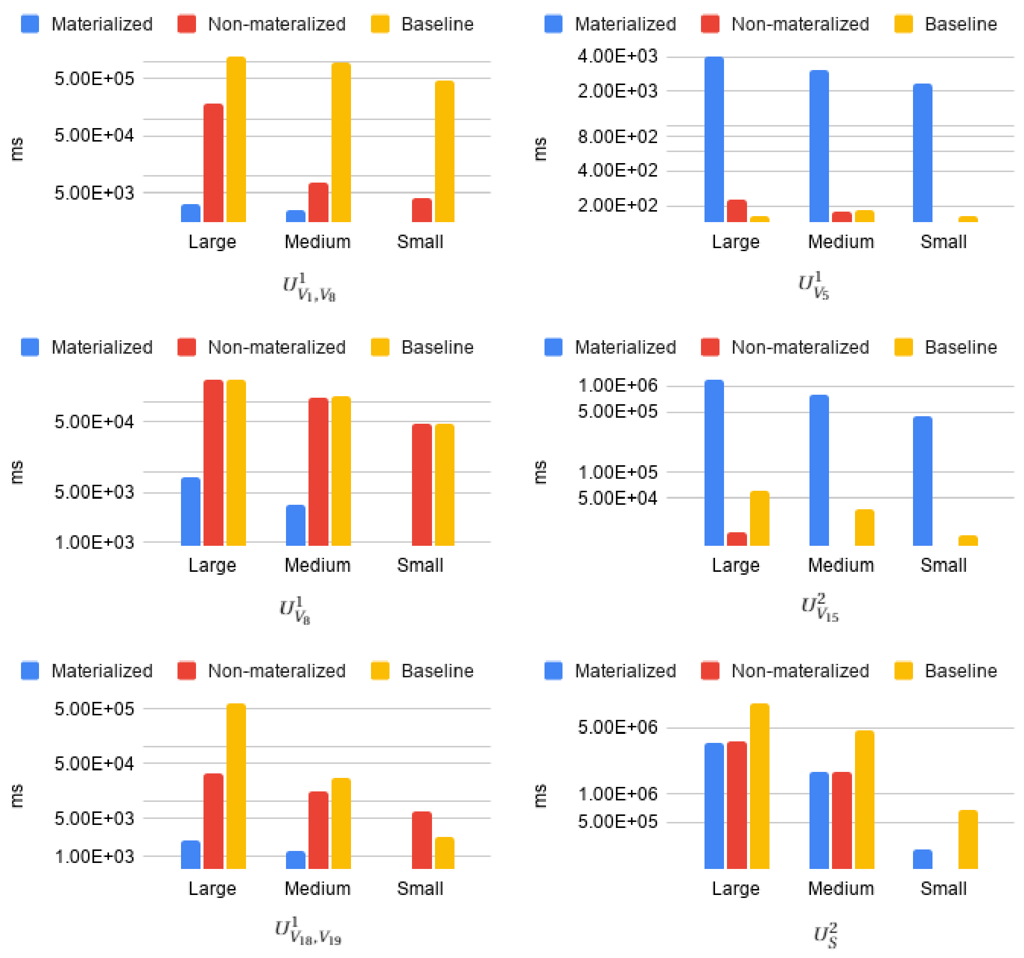

Figure 6.1: Execution time comparison for the three methods for several queries on the three different databases (log-scale). Any method which does not have a corresponding bar in the chart indicates that it is too small to be visible.

plore all sets of one-hop paths between `p1` and `t1`, all sets of one-hop paths between `p2` and `t2`, and to join the latter two results when `t1` equals `t2`. The main cost of this is the expensive join, since we expect a lot of results for both one-hop matches. With the NodeByIdSeek search only filtering out results on the first step, then as long as `V6` and `V8` are relatively large, we still incur the large cost of the join. Though the text-based method would suffer for the same reasons, this is a case where we expect that materialized views do not provide a significant benefit.

As for U_S^2, the materialized approach does not outperform the others. It is a query which only contains a one-hop path,and also has an interesting detail in its execution plan: there is only one NodeByIdSeek despite there being two views used. Since the query only involves a single hop, then a second NodeByIdSeek would provide zero additional benefit, since there is no way to avoid searching for all single-hops from either `n` or `p`. As a result, view use queries that contain only one hop but reference several views benefit less from materialization, compared to those that either only use one view or those that do not contain paths in the query.

6.3.5.3 Queries across database size

As expected, in most cases execution time for use queries increases with database size as we can see in Figure 6.1. In terms of consistency, we observe that in general queries do seem to perform consistently across the three databases. For instance, the materialized approach is always better for $U^1_{V_1,V_8}$, $U^1_{V_8}$, and $U^1_{V_{18},V_{19}}$ on all three databases, while it is worse for $U^1_{V_5}$ and $U^2_{V_{15}}$. There is an exception which is U_S^2 where the relative performance differs slightly; the materialized approach is equal to the non-materialized approach for the large and medium databases, but is worse for the small database.

6.3.5.4 Queries with the same underlying views

We refer back to Table 6.7 which also shows us the performance of queries that use the same views. We already observed that for views that are slow to create, queries are always faster with materialization. We see this for $U^1_{V_1,V_8}$ and $U^1_{V_8}$, which both use V8, and $U^1_{V_{18},V_{19}}$ and $U^1_{V_{19}}$, which both use V19. On the other hand, for fast views

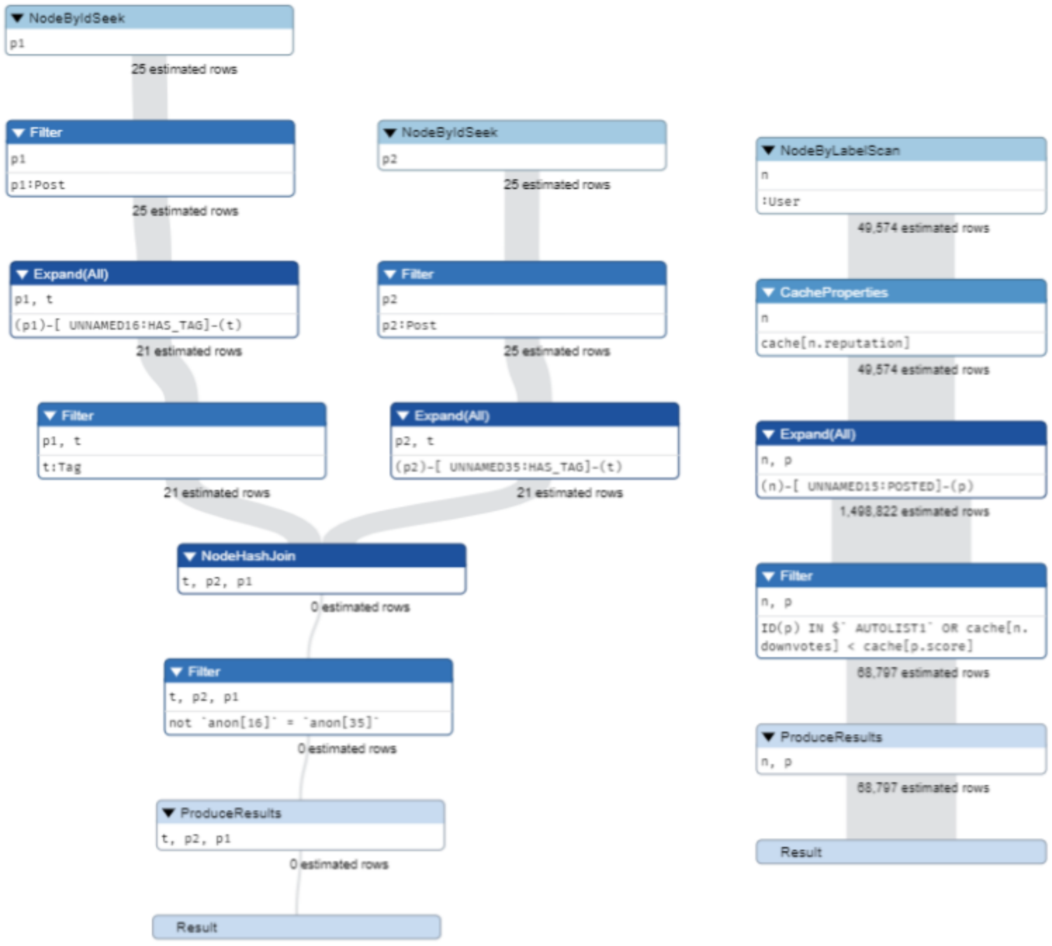

Figure 6.2: Execution plans for U_S^1 (left) and U_S^2 (right)

we see that some use queries perform differently even if they use the same view. For instance, the materialized approach for $U_{V_5}^1$ is worse by an order of magnitude compared to the other approaches, while performance for $U_{V_5}^2$ is roughly the same for all approaches. Similarly, $U_{V_{15}}^1$ is better with materialization but $U_{V_{15}}^2$ is much worse, and we see the same difference between $U_{V_{20}}^1$ and $U_{V_{20}}^2$.

6.3.6 Analysis of Non-Materialized Approach

Since using non-materialized views requires the execution of a sub-query for each view used, we should be able to estimate the execution time for each query. In particular, we expect the time taken for each of these queries to be **at least** (at minimum) the sum of the time taken to create the views that the query uses, from Table 6.2. We can see this in Figure 6.3, which shows the expected minimum time for query execution versus the actual time for the three databases.

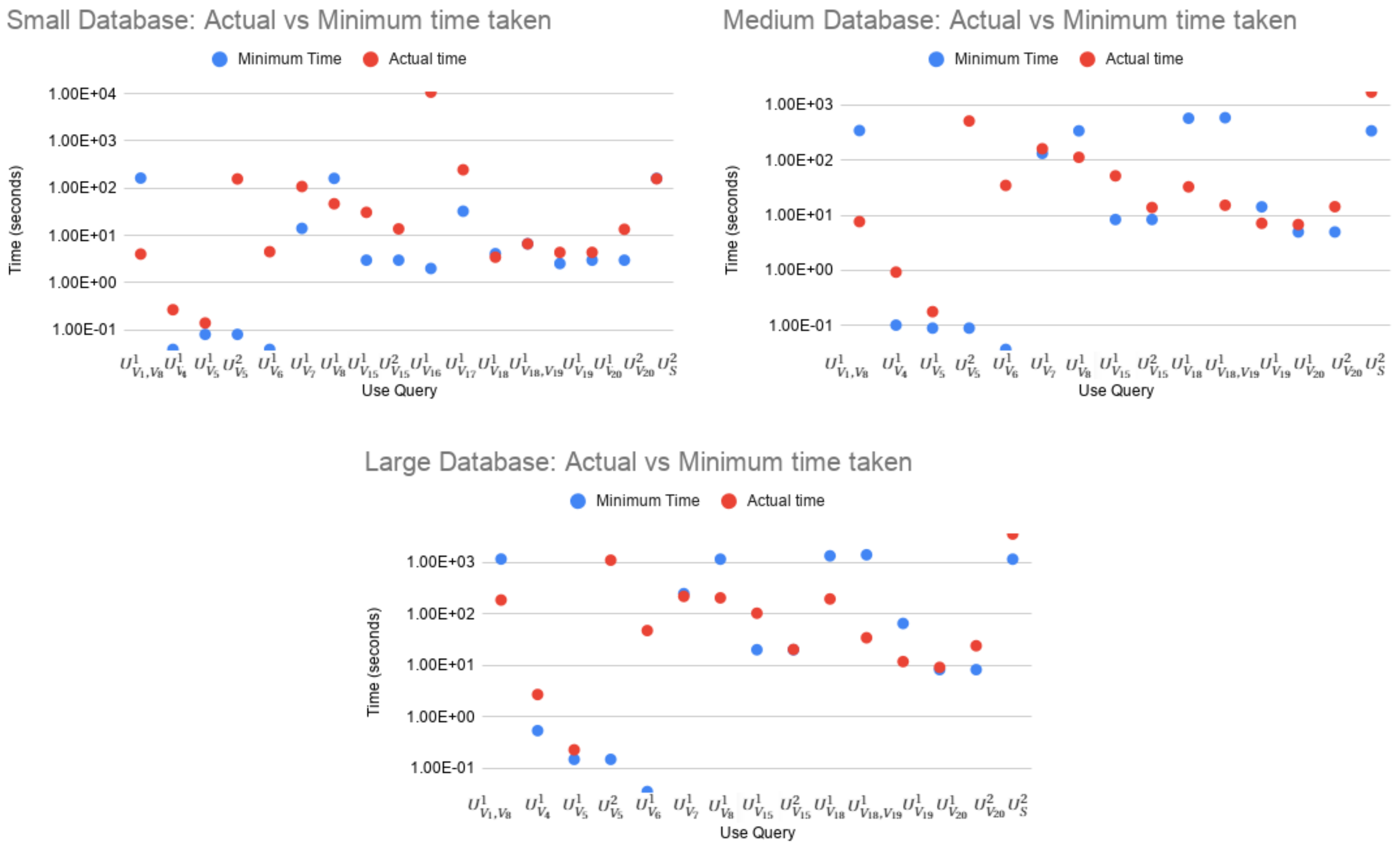

Figure 6.3: Expected minimum time taken versus actual time taken for non-materialized queries on each database (log-scale). For the small database, all queries are shown and for the medium and large databases $U^1_{V_{16}}$ and $U^1_{V_{17}}$ are omitted as these queries did not terminate within a reasonable time.

For most of the queries, the actual time taken is equal to or higher than the minimum values. For some of these, the difference is little, like $U^1_{V_5}$, $U^1_{V_7}$ and $U^1_{V_{20}}$, and for others, the difference is very large. For instance, $U^2_{V_5}$, $U^1_{V_6}$, and $U^1_{V_{16}}$ take many orders of magnitude longer than the minimum. Aside from the latter queries, the run-time of the non-materialized queries seems to remain within one order of magnitude of the expected run-time, but there are enough queries for which this is not the case, the reason for which we are not sure.

However, there are queries that execute faster than executing the underlying views, such as $U^1_{V_1,V_8}$, and $U^1_{V_8}$ for the small database, and with an even more significant difference for the larger databases. It appears the rewrite of the query leads to a different execution plan that is beneficial in some cases.

We see a consistent result for most queries as the size of the database changes. For some queries, however, the non-materialized method improves relative to the minimum time as we increase the size of the database. For instance, $U^1_{V_{18}}$ and $U^1_{V_{18},V_{19}}$ take the minimum amount of time on the small database but both pull ahead by an order of magnitude for the medium and large databases.

6.3.6.1 Categorized Queries

While we mainly focus on materialized views when considering the categorization of use queries, there also seems to be a similar trend with non-materialized views to the one we observed earlier with materialized views. The difference is that instead of the view creation time being the leading variable, here we benefit more when the underlying view used is small. For these cases, the non-materialized approach is at least on par with the baseline queries, and significantly outperforms baseline queries in some cases ($U^1_{V_1,V8}$, $U^1_{V_{18},V_{19}}$, $U^1_{V_{19}}$, and $U^1_{V_{18}}$). While in these cases the non-materialized approach still performs worse than the materialized one, it never performs worse than the baseline.

6.3.6.2 Queries across database sizes

Unlike with the materialized approach, we see less consistency across the three databases when we compare the non-materialized approach with the baseline approach. While for most queries ($U^1_{V_1,V8}$, $U^1_{V_8}$, $U^2_{V_{15}}$, and U^2_S) the non-materialized approach performs consistently better on all three databases, there are some queries for which it is better on the large database but worse on the medium or small ones, or vice-versa (such as with $U^1_{V_5}$ and $U^1_{V_{18},V_{19}}$).

6.3.6.3 Comparison against baseline queries and materialized approach

Comparing the non-materialized approach with the baseline queries, we see from Figure 6.1 that the execution time for the non-materialized views is much better than expected. We expected a large overall performance loss due to the additional sub-queries that are executed, but it turns out that the pipeline in these queries are quite efficient - sometimes performing on-par to or even better than the materialized approach on fast views.

Furthermore, non-materialized views never significantly under-perform relative to the baseline queries, such as with $U^1_{V_5}$, and actually outperform baseline queries in several cases ($U^1_{V_1,V_8}$, $U^2_{V_{15}}$, and $U^1_{V_{18},V_{19}}$, and U^2_S). We see similar results from Table 6.7, where the non-materialized approach performs equal or better than the baseline queries, with the only exception being $U^1_{V_{17}}$. In general, filtering and limiting branch searches of a path using the "`WITH .. as`" keywords leads to a very optimal execution plan as it reduces the cardinality at a very early step [8], which the baseline queries may not be able to do. Of course, this depends both on who is writing the query and also in many cases on the nature of the path involved in the query. It turns out that the optimization we had thought we made when designing some baseline queries ended up slowing them down! This is particularly the case for $U^1_{V_1,V_8}$ which we described in section 6.3.1, but also $U^1_{V_{18},V_{19}}$ and U^2_S to a lesser degree. Our automated rewriting approach for non-materialized views provides consistently good performance that one may consider that method as a fall-back implementation for when the execution time of a view creation is very low, as it is better or equal than queries that a "human would write", but also not burdened by any maintenance that the middleware would be subject to for materialized views.

6.4 View Maintenance

In this section we discuss the performance of the view maintenance for materialized views. That is, we ensure that (1) the detection method is correct, (2) we determine the effectiveness of the algorithm, and (3) we determine and discuss which types of graph updates will most likely produce a false positive result (i.e, unnecessary invalidations) and why. We create a set of graph queries C_i which change the graph, and measure the frequency of correct vs unnecessary invalidations given by the middleware for each query. In particular, we keep track of the views which a query actually do affect, that is the views that must be re-evaluated. Additionally, we count the views that the middleware decides to re-evaluate.

For (1) we look at false negatives. A false negative is a view that our algorithm does not pick to be re-evaluated but the update did affect the view result. If the maintenance algorithm is correct in detecting all views that require re-evaluation, then there are no false negatives. If this is the case, then we know that the algorithm is at least correct for the changes we consider.

For (2), we are interested in how effective the algorithm is. After all, even if we know it is correct, it is not a proper sole indicator of how effective it is, as we can trivially achieve correctness by deciding to always re-evaluate every view on every graph change. Therefore to measure effectiveness, we look at false positives. A false positive is a view that is re-evaluated but does not need to be re-evaluated because its result set does not change due to the graph change.

For (3) we categorize graph changes into three types: deletions, insertions, and update changes.

6.4.1 Graph Changes

We now describe each query C_i along with the list of all views that are affected by it. Furthermore, we categorize these by the type of graph change. There are a total of 12 changes, and the summary of these graph changes are found in Table 6.8.

The objective of this set of graph changes is to encompass a wide range of possible cases. For example C_{10} should invalidate V1 since their set of conditions overlap,

Table 6.8: Graph Change Details

Graph Change Query	Query Detail	Views Affected	Change Type
C_1	MATCH (n:User) WHERE n.upvotes <100 SET n.upvotes = 0	V18	Update
C_2	MATCH (n:Post) WHERE n.score <50 SET n.score = 0	V10, V19	Update
C_3	MATCH (n:User) WHERE n.userId = 19 SET n.upvotes = 1 + n.upvotes	V3, V8, V11, V18	Update
C_4	MATCH (n:Post) WHERE n.score <500 SET n.comments = 30	V8, V18	Update
C_5	MATCH (n:User)-[:POSTED]-(p:Post) WHERE n.userId = 18 REMOVE p.score	V1, V2, V7, V10, V11, V19	Update
C_6	MATCH (n:User)-[:POSTED]-(p:Post)-[:HAS_TAG]-(t:Tag) WHERE n.userId=19 OR p.score<0 OR t.tagId='java' REMOVE t.tagId	V5, V15	Update
C_7	CREATE (n:Post)	None	Insert
C_8	CREATE (n:Tag {tagId:'html'})	V5	Insert
C_9	MATCH (t:Tag) WHERE t.tagId = 'html' CREATE (p:Post{score:500})-[:HAS_TAG]-(t)	V1, V2, V5, V12, V15	Insert
C_{10}	MATCH (n:Post) WHERE n.score >800 DELETE n	V1	Deletion
C_{11}	MATCH (n:User) WHERE n.upvotes <100 DELETE n	V4	Deletion
C_{12}	MATCH (n:Tag) WHERE n.tagId = 'java' DELETE n	V5	Deletion
C_{13}	MATCH (t:Tag) WHERE t.tagId = 'html' DELETE t	V5	Deletion

and it should not invalidate V2 because their set of conditions does not overlap. We have a total of 6 update queries, 3 insert queries, and 4 deletion queries. There are more update queries because there are more cases to consider for updates, since our invalidation scheme is more specific for these. In general, there are different levels of complexity to the changes. For instance, the insertions begin with a simple node insert with no attribute, to one with a single attribute, to a path insertion. On the other hand, deletions only delete single nodes and not paths, which we discuss later.

6.4.2 Correctness and False Positive Rates

Over the 20 views that we have defined at the beginning of this chapter, we run the 13 graph changes. Overall, for correctness we confirm that there are no false negatives, i.e there are no views that are wrongfully skipped in the re-evaluation step.

To analyze effectiveness, we define as a false positive rate the number of false positives over the number of negatives. In other words, it is the ratio between the number of wrongly re-evaluated views and the number of views that do not need re-evaluation. We first bring an overall false positive rate for updates, insertions, and deletions. This average rate is calculated as the total number of false positives over the total number of negatives for all graph changes of that type. In regard to false positives rates, Table 6.9 shows the rate for each change query. Categorizing it by the change type we have:

1. For updates the average false positive rate is 5.8%.
2. For insertions the average false positive rate is 55.56%.
3. For deletions the average false positive rate is 42.11%.

We see that updates on existing nodes and edges give an overall low positive rate, which only wrongly marks a view as outdated with a rate of 5.8%. While updates and deletes consider conditions in the same way, an update only invalidates a view if the attribute updated is in the condition set of the view declaration or if the view declaration has no condition. This extra requirement for invalidation does not exist with insertions and deletions. Furthermore the probability for overlap clearly depends on the selectivity of the conditions. For several of our updates we refer to a single node, such as in C_5, so we can more precisely find the views that need to be invalidated. Generally, the larger the interval of condition, the more likely we are to encounter a case where we assume an overlap and invalidate, but the database does not contain nodes that are actually in the intersection.

Table 6.9: False positive rates for each query

Graph Change Query	Change Type	False Positives	Total Negatives	False Positive Rate
C_1	Update	1	19	5.2%
C_2	Update	1	18	5.55%
C_3	Update	1	16	6.25%
C_4	Update	3	18	16.67%
C_5	Update	0	14	0%
C_6	Update	0	18	0%
C_7	Insert	13	20	65%
C_8	Insert	5	19	26.31%
C_9	Insert	12	15	80%
C_{10}	Deletion	14	19	73.6%
C_{11}	Deletion	10	19	52.63%
C_{12}	Deletion	4	19	21.05%
C_{13}	Deletion	4	19	21.05%

6.4.2.1 Insertions and deletions versus updates

Updates have a lower false positive rate than insertions and deletions. This is not surprising, as updates perform the extra check on the changed attribute. Again, our insertions and deletions might also be less specific, requiring us to be more conservative in invalidation. Apart from the condition checks, however, there is an additional reason for the higher false positive rate for insertions and deletions. Recall from Chapter 4 that during maintenance we always check conditions stored in the entries of the dependency table. If an entry is evaluated as affected (i.e the node or edge in question might be affected by the graph change) then all dependent views of that entry will be re-evaluated. However, there is an extra check that we do not make, which is whether or not a dependent view contains path conditions. To demonstrate this, consider the following 1-hop view declaration query:

```
CREATE VIEW AS 1hopView
MATCH (n:User)-[:POSTED]-(p:Post)
RETURN p
```

and consider the following deletion-type graph change:

```
MATCH (n:User) WHERE n.displayname = 'foo' DELETE n
```

We know that this graph change should not actually affect the view, since any `User` node which belongs to the view must contain at least one edge, which connects itself and a `Post` node. Thus a simple `DELETE` will never delete any node which belongs to the view. However, we never make this check because all the dependency table knows is that the `User` entry contains `1hopView` in its dependent list, therefore it will (incorrectly) decide that the view must be re-evaluated. Any deletion will encounter this issue, possibly contributing to a higher false positive rate. For instance, C_{10} leads to an unnecessary invalidation of views that contain a `Post` node, but the `Post` node is part of a path. A similar problem arises with insertions that only insert nodes but not edges. As for C_8, since this is an insertion that does not insert attributes, the middleware will re-evaluate all queries which contain a Post node without any conditions; in most cases this Post node is part of a path, in which case we get a false positive.

We note that if we replace all the `DELETE` with `DETACH DELETE`, then these views would actually be affected, and the resulting false positive rate would be much lower, as the same views would be invalidated, but it would be a correct invalidation. In a sense, our implementation might be more suited for considering the latter type of deletion change. Nevertheless we propose this as an improvement to the invalidation algorithm; for regular deletions and insertions, an additional check can be made to see whether in the affected view, the node is part of a path or not. If so, then there is no need to reevaluate.

6.5 Summary

Finally, we summarize the results from this chapter and discuss the feasibility of using views in general. We have created a set of views and evaluated the performance of queries that use these views with both the materialized and non-materialized approach, and compared it to a set of equivalent baseline queries that work on the base graph instead of views. By categorizing these views by two parameters - creation time and view size, we found that the best performance for materialized views was found for queries that use views that are take long to materialize. Furthermore, we find that non-materialized views perform surprisingly well, with the majority of the queries performing on-par or better than the baseline queries, and we observe this to be the case especially with queries that use small views. However, further research into the parameters for classification is required to fully understand what really affects the performance, for both materialized and non-materialized views. Finally, we verify the correctness of our maintenance algorithm and discuss the false positive rates for different types of graph changes.

As a general guideline, it is safe to say that if one expects to create views with complex queries that have slow execution time, then it would be highly beneficial to use the materialized approach. On the other hand, if one expects to create small views, then it would be more beneficial to use the non-materialized approach, as the materialized approach may be worse than not using views at all (i.e., the baseline queries). In fact, when the nature of the views is unknown, the non-materialized approach is the safest approach, as it always performs as well as or better than baseline queries. The most optimal approach may be a hybrid approach, where the default is the non-materialized approach, but to switch to the materialized approach for especially complex queries that take a long time to execute.

Furthermore, the nature of the database application matters too; if the database is read-only then the overhead of maintenance becomes a non-factor; since a read-only database does not expect updates, then one would be able to benefit from materialization without any extra overhead, aside from the initial materialization. On the other hand, if the database undergoes changes often, then the constant re-execution of queries will far outweigh any benefits gained from materialization, especially if these graph changes consist of insertions. However, if the changes consist mostly of updates, then it may be possible with a low enough read/write ratio that views are still worth materializing.

Chapter 7

Related Work

In this section we discuss existing works surrounding query languages in general, views in relational database systems, and views for graph database systems.

7.1 Views in Relational Databases

Databases such as MySQL[1] and PostgreSQL[2] offer non-materialized views. In MySQL these are `MERGE` algorithm views, and in PostgreSQL, these are known as *Rules*. For these views, no result is stored and submitting a query that contains references to the view leads to a rewrite of the query.

MySQL offers the `TEMPTABLE` algorithm for views, which creates an actual table to store query results for later reference. However, these tables can not be updated, meaning that they are fixed upon declaration. Should the underlying tables change, the view simply becomes outdated.

PostgreSQL supports materialized views and persists the view query result for re-use. However, updates to each view must be manually invoked by calling `REFRESH MATERIALIZED VIEW` on the view, which results in a full re-computation of the view. PostgreSQL has not implemented IVM yet as a feature.

Details about Oracle's incremental materialized view maintenance are discussed

[1]mysql https://dev.mysql.com/doc/refman/8.0/en/view-algorithms.html

[2]psgrsql https://www.postgresql.org/docs/9.3/rules-views.html

in [5], though they also focus on deferred-mode maintenance, which is an alternate approach that does not update materialized view tables immediately upon a change on the base table, but rather only when the view is used. A deferred-mode maintenance can be a simple refresh command which re-computes the full view, or incremental, in which case auxiliary tables are needed to hold information about the tables since the last update.

IBM's DB2 database also provides materialized views which they call *automatic summary tables* [21]. Automatic summary tables can be synchronized with their base or master tables, or they can use a deferred maintenance mode like with Oracle's materialized views. When possible, these summary tables are incrementally updated, but still perform full re-computations of queries when necessary.

7.2 Graph Query Languages

We mention that for graph databases there is no standard query language. In relational systems, relational algebra [10] and SQL are closed query languages used to query database tables, which leads to very structured nesting and linking of queries, with clear semantics. Graph-based languages are typically complex. As graphs contain paths, graph query languages have the option to support navigational graph queries [4], which allows queries to contain regular expressions that cannot all be expressed by a first-order language [26], such as relational algebra. Below is an overview of a few query languages that may be of interest.

G-CORE is a closed graph query language which is simple but less expressive than Cypher. G-CORE is a unique query language proposed by [3] which uses graphs as the input and output of graph queries. They extend the property graph model to a *path property graph model*, and treat paths within the graphs as first-class citizens, allowing them to have their own identifiers, labels, and properties that may be stored within the base graph. They only allow queries to return graphs, just like how SQL queries always return tables. Therefore, we believe building a view management system for G-CORE might be easier than for Cypher. In fact, views are supported within G-CORE, where a view is a sub-graph of the original graph. In fact that is conceptually similar to what our views return (nodes, or edges, or paths). However [3] does not provide details about view implementation. Furthermore, there is no notion of using the base graph together with the view with G-CORE - all their queries must be on nodes or relationships that exist in the input graph(s), making them more similar to our `LOCAL` queries.

GraphQL is another closed graph query language which extends relational algebra and generalizes the selection operator to graph pattern matching. [17] details the language with the majority of the work focused on optimizing the time complexity with various heuristics.

SPARQL [27] is another navigational graph query language with many similarities to Cypher. As with Cypher, SPARQL works on a property graph model and is navigational, which allows for more complex path queries.

XQuery and XPath XML data can also be enabled within relational database systems, or stored within a native XML database. XML data may also represent paths due to its tree-like structure. XPath [12] and XQuery [25] are two different languages used to query these trees, with XPath using similar constructs to Cypher, such as pattern matching within paths.

The inherent existence of paths in graph-based systems makes it challenging to construct views, and we can imagine that this complexity is not limited only to graph databases but also to any database which expresses data with paths.

The work that this thesis presents may be extendable and in principle, we would be able to use views with all of the above languages.

7.3 Existing Works for Views in Graph Databases

Currently there has been very little work done towards views in graph database systems. [7] use a rule-based system with a Rete network [13], along with a modified network, to create view models on *deductive* graph databases. A Rete network is a system of nodes that represents all rules of a rule-based system and typically uses working memory to store data that match on the rules. It is optimized to process changes in working memory to unlink data with rules once they no longer match. In particular, they transform Neo4j queries through several layers until it reaches a form that can be understood by graph transformation rules in their network. However, as they use a rule-based system the space overhead is expected to be very high.

[11] proposes a theoretical approach to produce view models queries from MATLAB Simulink models. The maintenance of these models involves reading model changes as notifications that can be detected by rules in a Rete network. Their process, like with [7], also involves several steps of model translation before a graphical representation can be produced. However, this work is very specific to Simulink model views and is not a general solution to property graph views.

Oracle also provides property graph views, but only as a mapping from standard RDF data to a property graph [19], rather than a view as a sub-graph of an existing graph database.

Chapter 8

Final Conclusions and Future Work

8.1 Conclusion

In this thesis we explored the concept of views in a graph database system. We first extended the existing graph query language that Neo4j uses. In doing so, we defined three different types of views - node views, relationship views, and path views - and restrict the return types to nodes, edges, or paths, and added the language for the creation of these views and queries that use the views.

We then discussed the implementation of materialized and non-materialized approaches for these views. For the materialized approach, we use Neo4j's internal node identifiers to materialize the data, and re-write incoming queries that use views with these node identifiers, speeding up the queries thanks to the index within Neo4j. As for the maintenance, we create a dependency table structure, which store components of a view, including the conditions on the nodes and/or edges relevant to the view. We create a step-by-step maintenance scheme for any incoming graph changes to determine whether a view in the system should be re-evaluated or not, and we described how we use the dependency table to do so. For the non-materialized approach, we automatically re-write queries to pipeline view declaration sub-query results into the view use query.

For our evaluation, we mainly focus on the performance of using materialized and non-materialized views, and we gauged this by comparing their performance against an equivalent set of baseline queries. We created a benchmark with a wide range of view declaration queries, view use queries, and graph changes to evaluate our approaches. By categorizing views with certain parameters, we successfully identified trends for both materialized views and non-materialized views. For the maintenance evaluation, we verified the correctness of our maintenance scheme and also look at the false positive rates for each different type of graph change: updates, insertions, and deletions. While we did discover a high false positive rate for insertions and deletions, the updates in our benchmark tend to cause a very low percentage of false positives.

In general, our performance evaluations show that both materialized and non-materialized views are very promising for graph-based databases. In particular, materialized views can be extremely effective when the underlying view takes long to materialize, and non-materialized views are almost always better than the baseline queries. We suggest a hybrid approach to use non-materialized views by default, and to materialize it when we know the underlying view declaration query will be slow to execute.

8.2 Future Works

8.2.1 Support for Other Features in Cypher

In this thesis, we have not looked in detail at aggregations that can be used at various locations in a Cypher query. In commercial RDBMs, simple aggregations such as COUNT, SUM, MAX,.. etc. are allowed in materialized view declarations. It may be worth exploring if there are simple aggregations that can be covered easily for graph-based views.

Furthermore, we have not looked at variable-length paths. While we have explored explicit paths in a query, Cypher also allows queries to contain variable length paths between nodes. The length of such a path may be unbound (in which case, it will return as long as there is some path between the nodes), or within a specified range. Adding support for these paths would improve the practicality of the language extension.

In general, we have not looked at all the features that Cypher provides but focused on the constructs that we believe are generally important for queries on nodes, covering predicates similar to those found in relational database systems and paths.

Looking into detail into all the language features Cypher provides could be interesting specifically for the Neo4j database system.

8.2.2 Integration of Approach in Database Engine

The work in this thesis is done only at the middleware layer. That is, we do not modify Cypher itself, nor do we modify the internals of the Neo4j database which we used. An improvement would be to integrate our system with the database engine. It would be especially useful when using views; we would be able to leverage our data structures and directly pass the lists of node/edge identifiers to the query, rather than rewriting the query.

8.2.3 Materialized Views Maintenance Algorithm for Views Involving Paths

In Chapter 6 we observe that part of the reason why deletions and insertions might suffer from unnecessary invalidations is because the maintenance does not care about paths within a view declaration query. That is, a deletion or insertion of a node will not affect any view with a path, but the middleware may still invalidate such a view purely because it looks at nodes and relationships of a view declaration query independently. That is, our dependency table does not contain any notion of a view depending on a *path*. For instance, if we can store the context of a view, then we would be able to easily determine that a graph change which inserts `(:Label1)-[:REL1]-(:Label2)` will never affect a view that only contains a path involving `(:Label2)-[:REL2]-(:Label2)`. This quickly becomes very complex, but remains as a possible improvement to the maintenance scheme.

8.2.4 More Information on Parameters which affect Performance

As we only focus on two parameters - execution speed of the underlying view declaration query, and the size of the return set of that query, there is a lot of information we do not yet know. There are likely many more parameters that contribute to the complexity of a view declaration query, which affects the performance of any query which uses that view. A detailed study on that would be essential to pinpoint exactly what variables affect performance the most.

Furthermore, there is the challenge of determining a guideline for the best thresholds for these parameters. We chose an arbitrary threshold of 100,000 milliseconds and 100,000 records for the large database, but these should change depending on the size of the database. Thus, it is necessary to determine which factors affect the thresholds as well, if it is not only the database size.

Bibliography

[1] Adobe behance scales to millions of users for lower cost with help of neo4j. `https://neo4j.com/users/adobe/`.

[2] Apache airflow. `https://airflow.apache.org/`, 2020.

[3] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-core: A core for future graph query languages. In *Proc. of Int. Conference on Management of Data (SIGMOD)*, page 1421–1432, 2018.

[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[5] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proc. of VLDB Endowment*, volume 98, pages 24–27, 1998.

[6] S. Berretti, A. Del Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1089–1105, 2001.

[7] T. Beyhl and H. Giese. Incremental View Maintenance for Deductive Graph Databases using Generalized Discrimination Networks. In *Proc. of Graphs as Models Workshop at European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2016.

[8] A. Bowman. Tuning Cypher queries by understanding cardinality. `https://neo4j.com/developer/kb/understanding-cypher-cardinality/`, 2020.

[9] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of VLDB Endowment*, 1991.

[10] A. K. Chandra. Theory of database queries. In *Proc. of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems (PODS)*, pages 1–9, 1988.

[11] C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró. Query-driven incremental synchronization of view models. In *Proc. of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, pages 31–38, 2014.

[12] S. DeRose and J. Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. https://www.w3.org/TR/1999/REC-xpath-19991116/.

[13] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*, pages 547–559. 1989.

[14] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, et al. Formal semantics of the language Cypher. *arXiv preprint arXiv:1802.09984*, 2018.

[15] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proc. of Int. Conference on Management of Data (SIGMOD)*, page 1433–1445, 2018.

[16] A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Schuster, P. Selmer, and H. Voigt. Updating graph databases with Cypher. *Proc. of VLDB Endowment*, 12(12):2242–2254, 2019.

[17] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc of Int. Conference on Management of Data (SIGMOD)*, pages 405–418, 2008.

[18] M. Hunger. Import 10m stack overflow questions into neo4j in just 3 minutes. `https://neo4j.com/blog/import-10m-stack-overflow-questions/`, 2015.

[19] L. Jayapalan. Oracle spatial and graph property graph developer's guide. `https://docs.oracle.com/en/database/oracle/property-graph/20.4/spgdg/oracle-graph-property-graph-developers-guide.pdf`, 2020.

[20] J. Lee, J. Oh, and S. Hwang. Strg-index: Spatio-temporal region graph indexing for large video databases. In *Proc. of Int. Conference on Management of Data (SIGMOD)*, pages 718–729, 2005.

[21] W. Lehner, R. Sidle, H. Pirahesh, and R. W. Cochrane. Maintenance of cube automatic summary tables. *ACM SIGMOD Record*, 29(2):512–513, 2000.

[22] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[23] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE transactions on knowledge and data engineering*, 3(3):337–341, 1991.

[24] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications*, pages 29–46. 2012.

[25] M. Rys. Xquery in relational database systems. In *XML 2004 Conference and Exposition Proceedings*, 2004.

[26] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al. The Future is Big Graphs! A Community View on Graph Processing Systems. *arXiv preprint arXiv:2012.06171*, 2020.

[27] A. Seaborne and S. Harris. SPARQL 1.1 query language. W3C recommendation, W3C, Mar. 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[28] S. Srinivasa. Data, storage and index models for graph databases. In *Graph Data Management: Techniques and Applications*, pages 47–70. 2012.

[29] J. Webber. A programmatic introduction to Neo4j. In *Proc. of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218, 2012.