NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc THESIS

View Materialization Alternatives in Property-Graph Databases

Konstantinos N. Plas

Supervisor: **Yannis Ioannidis,** Professor at NKUA

Co-Supervisors: **Theofilos Mailis,** Postdoctoral Researcher at NKUA
Manolis Koubarakis, Professor at NKUA
Ioannis Kotidis, Associate Professor at AUEB

ATHENS

APRIL 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Εναλλακτικές Υλοποίησης Οψεων σε Βάσεις Δεδομένων Γραφημάτων

Κωνσταντίνος Ν. Πλας

Επιβλέπων: **Γιάννης Ιωαννίδης,** Καθηγητής ΕΚΠΑ

Συνεπιβλέποντες: **Θεόφιλος Μαΐλης,** Μεταδιδακτορικός Ερευνητής ΕΚΠΑ
Μανόλης Κουμπαράκης, Καθηγητής ΕΚΠΑ
Ιωάννης Κωτίδης, Αναπληρωτής Καθηγητής ΟΠΑ

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2024

MSc THESIS

View Materialization Alternatives in Property-Graph Databases

Konstantinos N. Plas

S.N.: 7115112200025

SUPERVISOR: **Yannis Ioannidis,** Professor at NKUA

COSUPERVISORS: **Theofilos Mailis,** Postdoctoral Researcher at NKUA
Manolis Koubarakis, Professor at NKUA
Ioannis Kotidis, Associate Professor at AUEB

THESIS COMMITTEE: **Yannis Ioannidis,** Professor
Manolis Koubarakis, Professor
Alex Delis, Professor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Εναλλακτικές Υλοποίησης Οψεων σε Βάσεις Δεδομένων Γραφημάτων

Κωνσταντίνος Ν. Πλας
Α.Μ.: 7115112200025

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Ιωαννίδης,** Καθηγητής ΕΚΠΑ

ΣΥΝΕΠΙΒΛΕΠΟΝΤΕΣ: **Θεόφιλος Μαΐλης,** Μεταδιδακτορικός Ερευνητής ΕΚΠΑ
Μανόλης Κουμπαράκης, Καθηγητής ΕΚΠΑ
Ιωάννης Κωτίδης, Αναπληρωτής Καθηγητής ΟΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Γιάννης Ιωαννίδης,** Καθηγητής
Μανόλης Κουμπαράκης, Καθηγητής
Αλέξης Δελής, Καθηγητής

ABSTRACT

Graph Database Management Systems (GDMSs) have gained significant popularity due to their inherent capability to represent information from diverse domains in the form of graphs. GDMSs facilitate the representation of data related to social networks, chemical compounds, and knowledge graphs, utilizing nodes, edges, labels, and properties. GDMSs that adhere to a native graph storage model demonstrate exceptional efficiency in both data storage and response to traversal queries within graph databases. Nevertheless, a challenge emerges when handling complex analytical queries within these systems, necessitating the development of novel techniques to accelerate their execution.

In this study, we tackle the issue of view materialization to expedite analytical-query answering in property-graph databases. More specifically, our focus is on identifying the types of views that should be materialized to accelerate the execution of queries featuring recurring patterns. We introduce various view alternatives to augment our initial graph and provide query rewriting techniques for responding to upcoming queries using these precomputed views. Our research centers on the characteristics of recurrent patterns within the query workload and explores the relationship between view characteristics and query types.

To examine the aforementioned correspondence, we have built a prototype system that identifies query patterns using frequent pattern mining techniques while employing variations of the Knapsack problem to select the best views for a query workload.

Our experimentation underscored the underperformance of subgraph materialization, the prevailing view type in GDMSs, across various query types. However, by introducing alternative approaches, we achieved a remarkable enhancement of up to 4.45 times in query execution efficiency and over a 2x reduction in storage costs. This suggests that adopting a more flexible view and indexing model can yield significant improvements in performance and storage efficiency.

SUBJECT AREA: View Materialization

KEYWORDS: graph databases, query optimization, view selection, view materialization, knowledge graphs

ΠΕΡΙΛΗΨΗ

Τα συστήματα διαχείρισης βάσεων δεδομένων γραφημάτων έχουν αποκτήσει σημαντική δημοτικότητα λόγω της εγγενούς τους ικανότητας να αναπαριστούν πληροφορίες από διάφορους τομείς με τη μορφή γράφων. Τα συστήματα διαχείρισης βάσεων δεδομένων γραφημάτων διευκολύνουν την αναπαράσταση δεδομένων που σχετίζονται με κοινωνικά δίκτυα, χημικές ενώσεις και γράφους γνώσης, χρησιμοποιώντας κόμβους, ακμές, ετικέτες και ιδιότητες. Τα συστήματα διαχείρισης βάσεων δεδομένων γραφημάτων που ακολουθούν ένα εγγενές μοντέλο αποθήκευσης γράφων επιδεικνύουν εξαιρετική αποδοτικότητα τόσο στην αποθήκευση δεδομένων όσο και στην απόκριση σε επερωτήματα διάσχισης μέσα σε βάσεις δεδομένων γράφων. Παρόλα αυτά, προκύπτει μια πρόκληση κατά το χειρισμό σύνθετων αναλυτικών επερωτημάτων, γεγονός που καθιστά αναγκαία την ανάπτυξη νέων τεχνικών για την επιτάχυνση της εκτέλεσής τους.

Στην παρούσα μελέτη, αντιμετωπίζουμε το ζήτημα της υλοποίησης όψεων για την επιτάχυνση της απάντησης αναλυτικών-ερωτημάτων σε εγγενείς βάσεις δεδομένων γραφημάτων. Πιο συγκεκριμένα, εστιάζουμε στον προσδιορισμό των τύπων όψεων που δύναται να υλοποιηθούν για να επιταχύνουμε την εκτέλεση επερωτημάτων που χαρακτηρίζονται από επαναλαμβανόμενα μοτίβα. Παρουσιάζουμε διάφορες εναλλακτικές όψεις για την βελτίωση του αρχικού μας γράφου και παρέχουμε τεχνικές αναδιατύπωσης ερωτημάτων για την απάντηση σε επερχόμενα επερωτήματα με τη χρήση αυτών των προ-υπολογισμένων προβολών. Η έρευνά μας επικεντρώνεται στα χαρακτηριστικά των επαναλαμβανόμενων μοτίβων εντός του φόρτου εργασίας των επερωτημάτων και διερευνά τη σχέση μεταξύ των χαρακτηριστικών των όψεων και των δίαφορων τύπων ερωτημάτων.

Για να εξετάσουμε την προαναφερθείσα αντιστοιχία, κατασκευάσαμε ένα πρωτότυπο σύστημα που αναγνωρίζει μοτίβα επερωτημάτων χρησιμοποιώντας τεχνικές εξόρυξης συχνών μοτίβων, ενώ παράλληλα χρησιμοποιεί παραλλαγές του προβλήματος Knapsack για να επιλέξει τις καλύτερες όψεις για ένα δεδομένο φόρτο εργασίας επερωτήματος.

Τα πειράματά μας υπογράμμισαν την υποαπόδοση του επικρατέστερου τύπου όψεων σε συστήματα διαχείρισης βάσεων δεδομένων γραφημάτων, σε διάφορους τύπους επερωτημάτων. Εισάγοντας εναλλακτικές προσεγγίσεις, επιτύχαμε μια αξιοσημείωτη βελτίωση έως και 4,45 φορές στην αποδοτικότητα εκτέλεσης επερωτημάτων και πάνω από 2 φορές μείωση του κόστους αποθήκευσης. Αυτό υποδηλώνει ότι η υιοθέτηση ενός πιο ευέλικτου μοντέλου όψεων και ευρετηρίασης μπορεί να αποφέρει σημαντικές βελτιώσεις στην απόδοση των επερωτημάτων και την αποδοτικότητα της αποθήκευσης των όψεων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Υλοποίηση όψεων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: βάσεις δεδομένων γραφημάτων, βελτιστοποίηση ερωτημάτων, επιλογή όψεων, υλοποίηση όψεων, γραφήματα γνώσης

Στον αδερφό μου τον Δημήτρη.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In recent years, there has been a significant surge in both scientific and industrial interest in graph-database management systems. Facebook, Twitter, Google, and others [91, 19, 65], have transitioned from using traditional relational databases to adopting graph-database systems and analytics. These companies utilize graphs to represent and elucidate the intricate relationships and interactions within their user bases, with their graphs encompassing billions of users, each corresponding to a node, and trillions of edges representing their relationships [91, 19, 65, 72]. This abstract representation of human relationships, while inherently limited, has proven to be of immense value in characterizing social connections. Alongside the aforementioned use in social networking, graph databases offer solutions for various industrial information systems. In the modeling of chemical compounds, datasets like PubChem [48] contain more than half a million graphs, and ChEBI contains a similar number [25]. Graph databases find further applications in software development and debugging [54], similarity searching in medical datasets [74], and recommender systems [37]. Another significant application where graph databases excel is in the domain of Knowledge Graphs (KGs), which are collections of interconnected and annotated entities. KGs such as DBpedia [27], Yago [86], Google's KG [81], and Microsoft's Satori [76] reach tremendous scale.

1.1 Graph-Database Management Systems

Graph-Database Management Systems focus on storage and querying tasks where the priority is on high-throughput and transactional operations. They use property-graph structures for semantic queries with nodes, edges, and properties to represent and store data. Systems such as Neo4j [67], OrientDB [71], Sparksee [82], JanusGraph [38], ArangoDB [7] and BlazeGraph [13] represent and process information based on the property-graph model. An alternative graph data model is based on *Resource Description Framework* (RDF), that allows the structuring of information in the form of triples, which consist of three components: the subject representing the entity which the statement is made; the predicate specifying the triple relationship; and the object representing the value or target of the triple statement. Systems such as Virtuoso [92], Stardog [85], Fuseki [30], and AllegroGraph [4], represent and process information based on the RDF model.

1.2 Native Graph Database Systems

Native graph database systems, which construct the physical storage of the graph model from the ground up, employ diverse techniques to optimize node traversal. By utilizing direct edges or analogous linking structures, these systems enable highly efficient graph traversals. With the ability to navigate through the graph structure by simply following edges, GDMS eliminate the necessity for intricate indexing operations. However, GDMS of this nature may not efficiently handle analytical queries that require aggregating information from various nodes within the property graph.

Example 1. The datalog query in Formula 1.1 that seeks information about the married couples who have acted together in the movie 'Eyes Wide Shut' can be answered very effectively. The query planner will first locate the corresponding movie based on its title

and then search for the couples within the movie.

$$\begin{aligned} q(u_1, u_2) \leftarrow & (u_{p1}, v_{mr}, u_{p2}), (u_{p1}, v_{act1}, u_{mv}), (u_{p2}, v_{act1}, u_{mv}), \\ & \mathsf{Married}(v_{mr}), \mathsf{ActedIn}(v_{act1}), \mathsf{ActedIn}(v_{act2}), \\ & = (u_{mv}.title, \text{'Eyes Wide Shut'}) \end{aligned} \tag{1.1}$$

The analytic datalog query in Formula 1.2, which seeks information about all married couples who have starred together in movies and the total number of these movies, presents a more complex challenge. The query planner must first explore all potential married actor couples and then utilize the underlying physical graph model to identify the movies in which they have co-starred.

$$\begin{aligned} q(u_1, u_2, \mathsf{Count}(u_{mv})) \leftarrow & (u_{p1}, v_{mr}, u_{p2}), (u_{p1}, v_{act1}, u_{mv}), \\ & (u_{p2}, v_{act1}, u_{mv}), \mathsf{Married}(v_{mr}), \\ & \mathsf{ActedIn}(v_{act1}), \mathsf{ActedIn}(v_{act2}) \end{aligned} \tag{1.2}$$

The query seeks triangle relations involving two actors and a movie. This query may appear either as a standalone question or as part of a more complex query. It's important to note that this is a challenging question that has been extensively analyzed in the literature. In fact, if $|G|$ represents the size of our graph database, the corresponding problem exhibits a data complexity of $|G|^{\frac{3}{2}}$ as discussed in previous works [5, 69]. If we assume that a small proportion of node pairs within the graph satisfy the given query and can be identified accordingly, we can significantly reduce much of the previously mentioned overhead.

1.3 View Selection in Graph Databases

An effective approach to enhance query performance in (graph) databases is *view materialization*. This involves the creation of a suitable set of precomputed results for a given (graph) database and query workload, with the aim of improving the efficiency of analytical queries. Query acceleration is achieved by identifying commonalities among analytical queries in the query workload. Precomputing these results minimizes the execution cost of existing or future queries w.r.t. to a cost function (e.g., query evaluation time, storage, and subexpression maintenance costs), under a set of constraints (e.g.,space budget). It's important to note that the problem of view selection differs in property-graph databases, where information is stored as nodes and edges. In such cases, it is not feasible to materialize tubular stored query results.

Kaskade [22] is the first system to materialize views for property-graph databases. It accelerates query answering in the context of property-graph databases under the assumption that each materialized view is a graph on its own. The latter approach, though it allows a clear distinction between the initial graph and its corresponding views, may induce a higher storage cost since the initial graph and its corresponding views contain redundant information. In this paper, we explore the different alternatives for materializing views by considering the structural elements of a property-graph database, i.e., nodes, edges, labels, and properties. Our work emphasizes the individual characteristics of the frequent query patterns that need to be materialized, considering materialization alternatives and best practices in relation to the characteristics of these frequent patterns.

1.4 Contributions

Our main contributions in the domain of view materialization for property graph databases are summarized as follows:

►*View Materialization Strategies.* We analyze different view strategies for query answering and show that, depending on the characteristics of the query pattern, the features of the property-graph database, and the characteristics of the corresponding answer set, a different strategy maybe be optimal among the view materialization alternatives. In this context, we examine view materialization strategies for various query characteristics such as acyclicity, the existence of edge-paths, as well as the existence of aggregate functions in the head of the query.

►*Query Rewriting.* For each strategy, we describe its corresponding query rewriting w.r.t. multiset semantics. We allow rewritings that are based on homomorphisms or isomorphism. It should be noted that isomorphism is a weaker form of homomorphism and we employ it in order to ensure sound rewritings for multiset semantics.

►*System building.* On the basis of the various materialization strategies and their corresponding rewritings, we build a view selection and query-rewriting system. Our system is based on existing work on frequent query pattern analysis within a query workload. It allows to represents a complicated query workload in a compact form and based on it creates the candidate views. To find the most beneficial set of view to materialize, we consider reduction to various knapsack variations and the available greedy strategies for solving them.

►*Experimental Evaluation.* The experiments conducted evaluate Neo4j's performance on various datasets and workloads, focusing on query execution time improvement and storage cost for each view alternative. Utilizing the GSpan algorithm, different minimum supports were applied to generate view candidates, revealing varying efficiencies across datasets. Results indicate subgraph materialization, which is commonly used in the literature, is not the most performant in most query types, while node labels consistently emerge as the most cost-effective option. Notably, answer-set hyperedges excel in reducing required rows for query execution. Our storage cost analysis showcased that subgraph materialization is, in most cases, the most expensive solution. We also explore the effectiveness of lazy indexing, presenting significant efficiency gains in execution time and storage space.

►*Lazy View Materialization.* We suggest a *lazy materialization strategy*, that incorporates all view related information into our corresponding graph on query-execution time. Thus an view is materialized only when we need to answer to a query that will be employing the corresponding view. This strategy allows to incorporate the materialization step into the execution step.

►*Path Queries.* For the rewriting of path queries, we consider various optimizations related to the length of the materialized path. I.e., whenever the a path of length between k and l is asked, we will consider various lengths that will accelerate the execution of the path query.

1.5 Structure

The rest of the paper is structured as follows: In Section 2 we provide some preliminary definitions. In Section 3 we analyze the various alternatives for views and indexes, as well as the query rewriting methods applied to each alternative. In Section 4 we provide an explanation of the view selection problem, different view selection algorithms and present the lazy view optimization. In Section 5 we perform an experimental evaluation of our index and view materialization alternatives. Finally, Section 6 presents the current literature on view materialization, while Section 7 summarizes the paper and mentions directions for future work.

2. PRELIMINARIES

Initially, we will present some preliminary definitions to formalize the view selection problem on a graph-database.

2.1 The Property Graph Data Model

In this section we will define the semantics of the property-graph data model. Our property graph semantics are based on the work of [6, 29]. We will employ datalog notation to represent a theoretical model for query representation. The datalog model we present has analogous semantics to the abstract language described in [6] as well as the language of Cypher [29].

2.1.1 Property Graph

For the set of labels $\mathbf{L}$, the set of properties $\mathbf{P}$, and the set of values V a property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:

1. N is a finite set of *node* (*vertices*);
2. E is a finite set of *edges* such that E had no elements in common with N;
3. $\rho : E \to (N \times N)$ is a *total function* that associates each edge in E with a pair of nodes in N;
4. $\lambda : (N \cup E) \to 2^{\mathbf{L}}$ is a *total function* that associates nodes and edges to a, possibly empty, set of labels from $\mathbf{L}$;
5. $\sigma : (N \cup E) \times \mathbf{P} \to \mathbf{V}$ is a *partial function* that associates nodes and edges with properties, and for each property it assigns a single value from V.

Given two nodes $n_1, n_2 \in N$ and an edge $e \in E$, such that $\rho(e) = (n1, n2)$, we will say that n_1 and n_2 are the *source node* and the *target node* of e respectively, i.e., Source$(e) = n_1$ and Target$(e) = n_2$.

2.2 Conjunctive Queries

In order to query our property graph, we assume the set of variables X and Y that are disjoint with N (nodes), E (edges), $\mathbf{L}$ (labels), $\mathbf{P}$ (properties), and V (values). We will call variables in X node and edge variables, they are used to represent nodes or edges in the context of a conjunctive query. We will call variables in Y edge-path variables, they are used to represent paths of arbitrary length in the context of a conjunctive query. An edge-path variable in Y has the form of $\vec{v}_{m,v}$ with $m, n \in \mathbb{N}$ and is used to represent a chain of k unknown edges for some $m \leq k \leq n$. The function Vars$(expression)$ returns the set of variables appearing in an arbitrary expression.

2.2.1 Edge Atom

For the variables $u, v, w \in X$, the atom (u, v, w) represents all edges in the queried property graph. The variable v represents the corresponding edge, with u representing its source node and w representing its target node.

2.2.2 Edge-Path Atom

For the variables $u, w \in X$ and the edge-path variable $\vec{v}_{m,v} \in Y$, an *edge-path atom* has the form of $(u, \vec{v}_{m,v}, w)$ where $\vec{v}_{m,v}$ represents a chain of edges with u representing the source node of the first edge and w representing the target node of the last edge.

2.2.3 Label Atom

For the variable $x \in X$, representing a node or an edge in the queried property graph and the label $l \in \mathbf{L}$, the label atom $l(x)$ indicates that the variable x is labeled with l. For a vector variable $\vec{v}_{m,v} \in Y$ representing a chain of unknown edges, the atom $l(\vec{v}_{m,v})$ indicates that all the edges represented via $\vec{v}_{m,v}$ are labeled with l.

2.2.4 Value-Predicate Atom

For the set of values V, a *binary value predicate* ρ corresponds to a subset of V^2. I.e., if V corresponds to the set of natural numbers, the binary predicate $\geq$ contains every pair $x, y \in \mathbb{N}$ such that $x \geq y$.

When considering a variable $x \in X$, which represents either a node or an edge, the notation $x.p$ refers to the assigned value of property $p \in \mathbf{P}$ on the specific node or edge represented by x. A *value-predicate atom* has the form $\rho(s_1, s_2)$ where ρ is a binary value predicate and s_1, s_2 are either property values or values in V. E.g., the value-predicate atom $= (u.name,$ 'John') will ask for the nodes or edges within the graph that are named as John. In the case that the value-predicate atom has the form of $\rho(u.p_1, w.p_2)$, we call the value-predicate atom a *theta-join*.

2.2.5 Conjunctive Query

For the variable vector $\vec{x}$ the body atoms $a_1, a_2, \ldots, a_n$ representing node-labeled, edge-labeled, property-path, and value-predicate atoms, a conjunctive query for our property graph is an expression of the form:

$$q(\vec{x}) \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n \tag{2.1}$$

The atom $q(\vec{x})$ is called the head of the query, the conjunction of atoms its body, while each variable appearing in the head of the query should also appear in its body: Vars$(\vec{x}) \subseteq$ Vars$(a_1, a_2, \ldots, a_n)$. The variables appearing in X are called bound variables, while non-bound variables are called free. It should be noted that all vector variables are free.

In our work, we will examine various categories of queries, such as acyclic queries and Free-connex queries. Queries of this particular type have properties that we can leverage to make more efficient view materialization choices.

2.2.6 Acyclic Query

A conjunctive query of the previous form is considered acyclic if its corresponding undirected graph, constructed from edge and edge-path atoms, contains no cycles.

2.3 Query Answering

We now provide the semantics of our datalog queries.

2.3.1 Atom satisfaction

Given an atom a and an assignment θ that maps the variables in Vars(a) to the nodes and edges $N \cup E$ in our property graph $G = (N, E, \rho, \lambda, \sigma)$, the variable assignment θ satisfies the atom a when:

I. a is an edge atom of the form (u, v, w) and the following apply: $\theta(v) \in E$; $\theta(u), \theta(w) \in N$; and $\rho(v) = (\theta(u), \theta(w))$.

II. a is a label atom of the form $l(x)$ and $l \in \lambda(\theta(x))$.

III. a is an edge-path atom of the form $(u, \vec{v}_{m,v}, w)$ and the following apply: (i) $\theta(\vec{v}_{m,v}) = (e_1, \ldots, e_k)$ for some $m \leq k \leq n$ with $e_1, \ldots, e_k \in E$; (ii) Target(e_i) = Source(e_{i+1}) for all $1 \leq i \leq k-1$; (iii) Source$(e_1) = \theta(u)$; (iv) Target$(e_k) = \theta(w)$.

   For the special case of an edge-path atom $(u, \vec{v}_{0,n}, w)$, a variable assignment θ such that $\theta(\vec{v}_{0,n}) = ()$ and $\theta(u) = \theta(w) = n$ for some $n \in N$ also satisfies the atom.

IV. a is a value-predicate atom $\rho(s_1, s_2)$ and the following apply: (i) s_1 is either some value $\tau_1 \in V$, or some property value $x_1.p_1$ of a node or an edge such that $\sigma(\theta(x_1), p) = \tau_1$; (ii) s_2 is either some value $\tau_2 \in V$, or some property value $x_2.p_2$ of a node or an edge such that $\sigma(\theta(x_2), p) = \tau_2$; (iii) $\rho(\tau_1, \tau_2)$ applies.

2.3.2 Solution

A variable assignment θ is a *solution* to a conjunctive query when it satisfies all the atoms in the body of the query.

2.3.3 Answer

For the vector of variables $X = (x_1, x_2, \ldots, x_k)$ appearing in the head of the query in Formula 2.1 , the constant vector $(a_1, a_2, \ldots, a_k) = (\theta(x_1), \theta(x_2), \ldots, \theta(x_k))$ is an answer to the query when there exists some θ that is a solution to the query. For multiset semantics, the answer $(a_1, a_2, \ldots, a_k)$ has a multiplicity of m when the corresponding query has exactly m solutions θ whose domain contains exactly the variables appearing in the body of the query.

2.4 Aggregate Conjunctive Queries

Suppose that Γ is the set of aggregate functions. An aggregate function takes as input a multiset S of values and returns a single value that may or may not belong to the multiset. E.g., given the multiset of integers $S = \{1, 2, 3, 3, 5\}$, the aggregate function Avg will return their average value Avg$(S) = 2.8$

2.4.1 Aggregate Conjunctive Query

For (i) the conjunction of atoms B; (ii) the variables $x_1, \ldots, x_k \in$ Vars$(B) \wedge X$ representing nodes or edges; (iii) the variables $x_{k+1}, \ldots, x_m \in$ Vars$(B) \wedge X$ representing nodes or edges; (iv) such that $\{x_1, \ldots, x_k\}$ and $\{x_1, \ldots, x_m\}$ are disjoint sets; (v) the *aggregate functions* $\gamma_1, \ldots \gamma_m \in \Gamma$; (vi) and the properties $p_{k+1}, \ldots, p_m \in \mathbf{P}$ an *aggregate query* has the form:

$$q(x_1, \ldots, x_k, \gamma_1(x_{k+1}.p_{k+1}), \ldots, \gamma_m(x_m.p_m)) \leftarrow B \tag{2.2}$$

$x_1, \ldots x_k$ correspond to the grouping variables of the query and $x_{k+1}, \ldots x_m$ correspond to the aggregate variables of the query. An exception to the previous syntax, is the Count$(\cdot)$ aggregate function. We may allow a head atom of the form Count(x) for some $x \in X$ representing either a node or an edge.

2.4.2 Answer

Given the query Q in Formula 2.2 and it's corresponding set of solutions Θ. Suppose that Θ is the set of solutions to the query in Formula 2.2. We say that two solutions θ_1 and θ_2 agree on the grouping variables, $\theta_1 \sim \theta_2$, when $\theta_1(x_i) = \theta_2(x_i)$ for all $i \in \{1, \ldots, k\}$. Based on the $\sim$ relation between solutions, we define equivalence classes in Θ . Each equivalence class $\Theta' \subseteq \Theta$ corresponds to the answer of $(a_1, \ldots, a_k, a_{k+1}, \ldots, a_m)$ such that I. for $i \in \{1, \ldots, k\}$ it applies $a_i = \theta(x_i)$ for all $\theta \in \Theta'$. II. for $j \in \{k+1, \ldots, m\}$ it applies $a_j = \gamma_j(S_j)$ where γ_j is the aggregate function on the j^{th} variable and S_j is the *multiset* $S_j = \{\theta(x_j) | \theta \in \Theta\}$.

2.4.3 Note on Aggregate Functions and Edge Paths

The presence of an edge-path atom of the form $(u, \vec{v}_{m,\infty}, w)$ in the body of a query may result to an infinite number of solutions w.r.t. to the aforementioned semantics. E.g., suppose that we have the conjunctive query:

$$q(u, \text{sum}(w.age)) \leftarrow (u, \vec{v}_{1,\infty}, w), (u.id = 12345)$$

asking for the transitive closure of the friends of some person and the corresponding summation of their ages. The existence of cyclic paths in our property graph, indicates that there exist an infinite number of solutions for the latter query and that the summation on age would be ∞. To handle this abnormality, we need to enforce a finite number of solutions. A straightforward way to accomplish is by assuming that each solution θ maps path variables only to acyclic paths within the graph. A simpler to implement solution would be to consider semantics such as the multiplicity of 1 is assumed for each answer between the source and target node of an answer to an edge-path atom.

2.5 Native Graph-Databases Management Systems

The view and index alternatives that we examine in this paper focus on the problem of view selection for *Native Graph-Database Management Systems* (GDMS). A native graph database management system is a type of database management system specifically designed to store, manage, and query property graphs.

2.5.1 Native System Architecture

A graph database management system can be characterized as *native*, when the system is built from scratch to accommodate graphs as stated by Lissandrini et al. in [53], to facilitate a direct linkage between the logical graph model and the physical graph storage. Native graph database systems, like OrientDB or Neo4j store the nodes and edges as separate record files with node and edge records maintaining direct links to other node or edge records accordingly. System's like Neo4j or TigerGraph [89] also employee a property called index-free adjacency. A GDMS utilizes the latter property when each node maintains direct references to its adjacent nodes. Thus, for each node, there is a constant time $O(1)$ enumeration delay of all its adjacent nodes. This results in two facts: *(i)* Each node acts as a micro-index, which is much cheaper than global indices. *(ii)* Query times are independent of the total size of the graph, they are proportional to the amount of graph searched. Sparksee, is a high performance native graph database, which employees multiple , seperate data structures to store and interlink the nodes and edges. one structure for objects, both nodes and edges, two for relationships which describe which nodes and edges are linked to each other, and a data structure for each attribute name [53]

2.5.2 Non-native GDMSs

It's worth noting that there are also GDMSs that are not native, meaning they are built on top of existing database technologies, such as relational databases or key-value stores, and provide graph-like functionality through additional layers or extensions. In [53], these systems are described as hybrid graph databases, demonstrating superior performance over native graph databases in limited scenarios. However, since the view and index alternatives we examine rely on the distinct properties of native graph databases, our emphasis will be on analyzing their performance on native graph databases. Nonnative GDMSs use (global) indices to link nodes together. These indices add a layer of indirection to each traversal, thereby incurring greater computational cost. Popular Non-native graph databases are ArangoDB [7], BlazeGraph [13], Sqlg [83]. Non-native graph databases have also been developed in academic settings, as is the case with the efficient open-source graph database MillenniumDB [93].

2.5.3 Indexing in GDMSs

Native and non-native GDMS systems provide various index alternatives. An index is a data structure that improves the speed of data retrieval operations on the cost of additional writes and storage space to maintain the index data structure. Traditional graph databases allow token lookup indices based on node and edge labelling: node indices allow to efficiently locate all the nodes of a specific label; edge indices allow to efficiently

locate all the edges of a specific label. Graph-databases also provide other type of indices, e.g., single-property indices on nodes or edges. Most mainstream graph database systems, such as Neo4j, utilize conventional B+ trees for indexing. OrientDB offers the flexibility to create hash indexes alongside SB-Tree, an indexing algorithm based on the B-tree structure. ArangoDB employs a variety of indexing methods, including hash indexes, skiplist indexes, and node or edge-specific indexes. Additionally, most modern GDMS also incorporate support for full-text search indexes.

2.6 View Selection

We now provide some definitions related to the view-selection problem.

2.6.1 Materialized View

A *view* is a stored query, while a *materialized view* is the result set of the stored query on a specific database instance.

2.6.2 Query Rewriting

Two queries are equivalent if they have the same answer set for every possible database. A query Q' is a *rewriting* of Q that uses the views $\mathcal{V} = \{V_1, \ldots, V_m\}$ if Q and Q' are equivalent and Q' contains one or more occurrences of materialized views in $\mathcal{V}$. A *rewriting function* Rwrt$(Q, \mathcal{V})$ takes as input the query Q and rewrites it to an equivalent query $Q' =$ Rwrt$(Q, \mathcal{V})$ using views from $\mathcal{V}$. A rewriting function Rwrt is optimal when there exists no other rewriting Q'' of Q such that Cost$(Q'') <$ Cost(Q'), with Cost being the function that maps a query to its estimated execution cost. Levy et al. [52] prove that for the conjunctive queries Q and W, there is a rewriting of Q using W iff $\pi_\emptyset(Q) \sqsubseteq \pi_\emptyset(W)$ i.e., the projection of Q onto the empty set of columns is contained in the projection of W onto the empty set of columns (the projections $\pi_\emptyset(Q)$, $\pi_\emptyset(W)$ are actually Boolean conjunctive queries).

2.6.3 Rewriting Benefit

The *degree of benefit* of a rewriting function to a query Q w.r.t. to a set of views $\mathcal{V}$ is defined as

$$\text{Bnft}(Q, \mathcal{V}) = \text{Cost}(Q) - \text{Cost}(\text{Rwrt}(Q, \mathcal{V})). \tag{2.3}$$

We also denote with Bnft$(\mathcal{Q}, \mathcal{V})$ the benefit of a set of views $\mathcal{V}$ to a query workload $\mathcal{Q}$. It is obvious that the benefit depends on the adopted cost model.

3. RELATED WORK

View materialization techniques have been extensively studied by the data-management community in the context of multiple-query optimization, Semantic Web & graph data systems, and data warehouses that are used to accelerate On-Line Analytical Processing.

►*Multiple-Query Optimization.* The view-selection process for the multiple-query optimization problem identifies the appropriate views that will be used for answering to a given set of queries. [79] studies the problem of multiple-query optimization providing its systematic analysis and considering global access plans that access subqueries. [78, 63] examine algorithms for multi-query optimization by selecting materialized views and indexes based on the Directed-Acyclic-Graph representation of the query plan to identify common subexpressions. [3] describe as system for view and index selection that incorporates several heuristics for pruning the space of possible view configurations. [99] present an efficient solution for the problem of common subexpression identification by introducing a light-weight mechanism, called tables signatures, for identifying shareable subexpressions.

[20] formalize the view selection problem and provide a lower Exp and an upper 3Exp bound for it. [43] devised an approximation algorithm that runs in time quadratic to the number of common subexpressions and provide theoretical guarantees on the quality of the solution obtained. [40] focus on the problem of subexpression selection, i.e., computing the subexpressions of a query that are most beneficial to be materialized and reduce it to the bipartite graph labeling problem, and integrate their implementation into the Cloudviews system [41].

A different methodology for solving the multiple-query and the view selection problem has been presented by [10, 18] that employ evolutionary techniques such as genetic algorithms. An overall analysis of the view selection problem has been presented by [59].

Our approach differs from previews view-materialization approaches since it allows to plug in various subgraph mining & forecasting solutions in order to predict the graph-patterns that will appear in future queries. It takes advantage of the graph-nature of knowledge-graph queries that allows to employ pattern-mining and forecasting techniques.

►*Semantic Web & Graph Data Systems.* Much research effort has been invested in the development of scalable centralised or distributed triple stores, techniques for indexing KGs and for processing queries. Among the centralised approaches, native triple stores like Jena [60], Sesame [16], HexaStore [96], SW-Store [1], MonetDB-RDF [80], RDF-3X [68], and BitMat [8] have been carefully designed to keep up pace with the growing scale of RDF collections. Systems like TriAD [34], RDFox [66], H-RDF-3X [36], EAGRE [98] implement various optimizations for the distributed execution of joins.

View materialization techniques have recently gained attention by the Semantic Web community and graph data systems. In [26], an approach for the materialization of shortcuts that reduces the execution cost of path queries is suggested. In [31], a different materialization strategy where an initial query workload $\mathcal{W}$ is transformed to a set of simpler views $\mathcal{V}$ along with a set of rewritings is presented. In [73], a strategy that caches SPARQL-query results and uses them to rewrite queries is studied. Caching strategies for graph query processing have been studied in [95, 94]. The caching algorithms in [73] and [95, 94] are based on finding subgraph-isomorphisms between incoming and cached queries. The approach in [73] is based on a canonical labelling algorithm, while [95, 94] adopt a *filter then verify* strategy where candidate graphs for isomorphism are filtered out based on cer-

tain features and then the actual test for isomorphism is performed. Finally, [61] studies the creation of an indexing structure that classifies triples based on the properties of their subjects and objects.

For a detailed analysis of Knowledge Graphs such that of DbPedia, the reader may refer to the existing bibliography. The query workload of DBPedia is studied in [75] and an analysis of the different operators that appear within DBPedia queries is performed. For various workloads, the structural characteristics related to the graph representation of queries are studied in [15], along with the evolution of queries over time. Finally, a study of the Wikidata knowledge graph is presented in [58].

►*Data Warehouses.* View-selection techniques have been studied for data warehouses and problem of online analytical processing. Several early techniques were proposed including AND/OR graphs [33], modeling the problem as a state optimization [88], and lattices to represent data cube operations [35, 49, 64], while the problem of view management has been also studied for decentralized OLAP applications using blockchains [62]. It should be noted that the problem of view materialization for data warehouses has different objectives targeting the improvement of Roll-up, Drill-down, and Slicing & Dicing operations.

4. VIEW-MATERIALIZATION STRATEGIES

In this section we focus on the candidate views for view materialization. I.e., given a graph database G, what are the materialization alternatives that we should consider (Section 4.1). Then, for each materialization alternative, we will investigate how our initial query will be rewritten (Section 4.2). In our experimental evaluation (Section 6), we decide on the materialization alternative that are the most beneficial for our underlying system and the corresponding characteristics of individual queries.

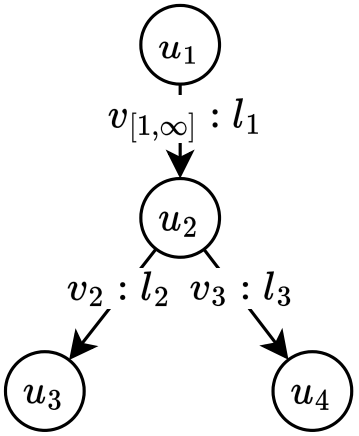

Figure 4.1: Frequent-Query Pattern

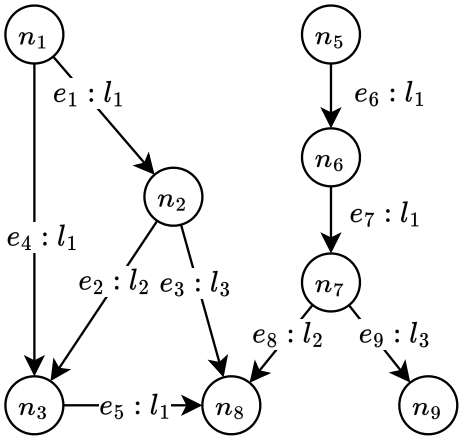

Figure 4.2: Graph Database G

4.1 View Candidates

Given the frequent query pattern of P within the query workload of $\mathcal{Q}$ and a graph database G, we want to examine all the view alternatives that can be inferred depending on the characteristics of the query pattern. I.e., given a query workload of $\mathcal{Q}$, the query pattern of P represented as a set of atoms involves operations that are being repeated during the execution of multiple queries. Our purpose is to materialize the result of these operations in order for them to be reused for existing and upcoming queries.

For traditional relational databases, a *view* is a stored query, while a *materialized view* is the result set of the stored query on a specific database instance. It should be noted that for graph databases materialized views can only be represented via the various components available on a property graph database, i.e., labels, properties, nodes, edges as well as their corresponding indexes. Thus we employ the term *view materialization* when introducing new labels, properties, fresh nodes, edges, as well as their corresponding indexes.

4.1.1 View Alternative

We will now examine alternative materialization strategies w.r.t. the graph-database setting. In the context of our work, we assume: *(i)* The property graph of $G = (N, E, \rho, \lambda, \sigma)$; *(ii)* a frequent query-graph pattern P, *(iii)* and its corresponding query representation namely:

$$Q_P : \quad q_P(\vec{x}) \leftarrow B_P \tag{4.1}$$

A variation of the previous query is one containing aggregate functions in its head. In Section 5.2 we will show how to determine the body, as well as the head of a frequent query-graph pattern P that is derived from an existing query workload.

We will now analyze different indexing strategies, that will allow to accelerate the execution of the query graph-pattern of Q_P, i.e., strategies that are based on the introduction of information that restricts the exploration for query answering on specific parts of the graph database. Additionally to creating the corresponding views for our property-graph database, we create a rule set $\mathfrak{R}$ that allows the more efficient rewriting of queries using views.

4.1.2 Node Labels as Views

A view materialization strategy that creates one or more labels for the nodes that are part of an answer to a query for P. To be more specific, for some node-variable u appearing in B_P we could introduce a fresh label l_{nd} for the answer set of the query of:

$$V_{\mathsf{l}_{\mathsf{nd}}}: \quad \mathsf{l}_{\mathsf{nd}}(u) \leftarrow B_P \tag{4.2}$$

For every solution $\theta : \mathrm{Vars}(P) \rightarrow N$, the node $\theta(u)$ will be labeled with l_{nd} and indexed accordingly by the underlying GDMS.

For every node-label atom $l(u)$ appearing in B_P, we will add in $\mathfrak{R}$ the datalog rule that the label of l_{nd} implies the label of l: i.e., $\mathsf{l}_{\mathsf{nd}}(x) \rightarrow l(x)$. Additionally, for every value-predicate atom of the form $\rho(u.p, \tau)$, for some $\tau \in V$ we add into $\mathfrak{R}$ the following rule $\mathsf{l}_{\mathsf{nd}}(u) \rightarrow \rho(u.p, \tau)$.

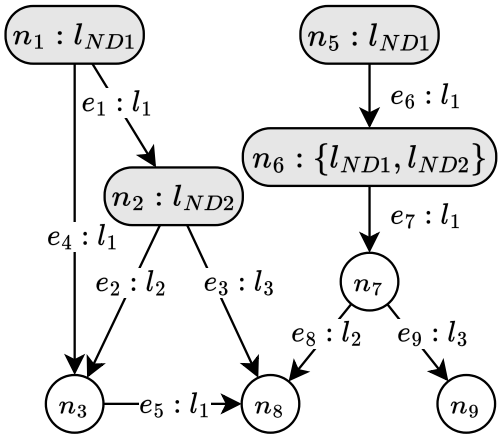

Figure 4.3: Node Labels as Views

4.1.3 Edge Labels as Views

A view materialization strategy that creates one or more labels on the edges that are part of a solution for the query of P. For some edge atom $a_i = (u, v, w)$ in P such that $u, v, w \in X$ is a variable representing an edge we may create the fresh label $\mathsf{l}_{\mathsf{edge}}$ and the corresponding index that identifies the edges that satisfy the atom and are also part of an answer to:

$$V_{\mathsf{l}_{\mathsf{edge}}}: \quad \mathsf{l}_{\mathsf{edge}}(v) \leftarrow B_P \tag{4.3}$$

For each solution θ to the previous query, the edge corresponding to $\theta(v)$ will be labeled with $\mathsf{l}_{\mathsf{edge}}$ and indexed accordingly.

As before, for every edge-label atom $l(v)$ in B_P, we will add in $\mathfrak{R}$ the rule: $\mathsf{l}_{\mathsf{edge}}(x) \rightarrow l(x)$. We also add the corresponding rules for value-predicate atoms. For every node-label atom $l(u)$ appearing in B_P with u corresponding to the source node of v, we will add in $\mathfrak{R}$ the datalog rule: $\mathsf{l}_{\mathsf{edge}}(v), (u, v, w) \rightarrow l(u)$. We also add the corresponding rules for value-predicate atoms as well for the target node.

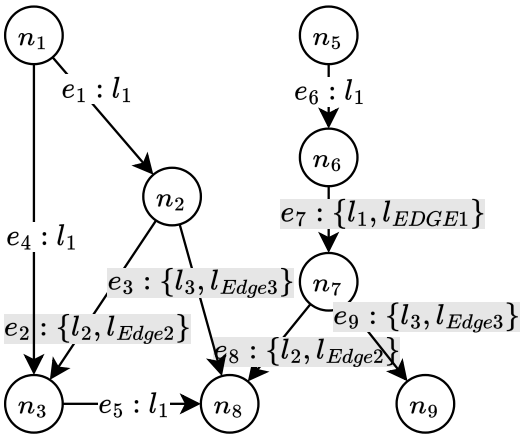

Figure 4.4: Edge Labels as Views

We will now examine view-materialization strategies that create additional nodes and edges in the property graph of G.

4.1.4 Answer-Set Hyperedges as Views

A materialized view that introduces and labels a directed-hyperedge between the nodes of each answer/solution to the query of Q_P. A hyperedge, in contrast to an ordinary edge, is one that connects any number of nodes. Since most native graph-databases do not directly support hyperedges, we can represent them using a bipartite graph [70]:

Assuming that all bound variables are node variables, for each answer $\vec{c}$ to the query in Formula 4.4 (the variables in $\vec{y}$ are the node-variables appearing in the head of the query in Formulae 4.1):

$$\mathsf{V}_{\mathsf{ans}} : \quad q_{\mathsf{ans}}(\vec{y}) \leftarrow B_P \tag{4.4}$$

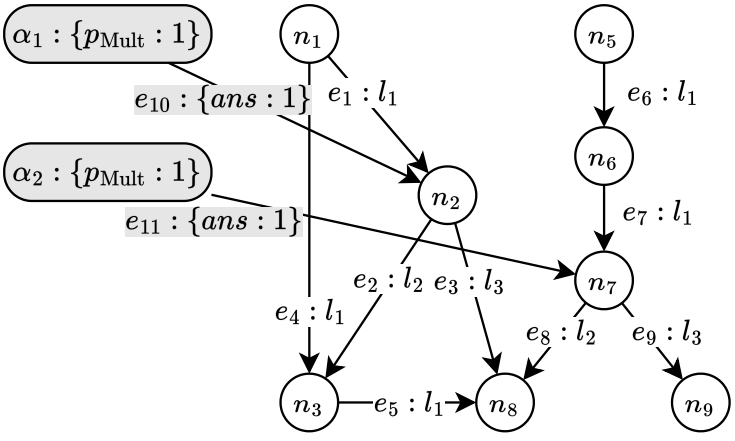

Figure 4.5: Answer Set as View

To represent our view: (i) we introduce in N a fresh node α representing each answer; (ii) for the i^{th} node c_i in $\vec{c}$, we add a fresh edge e_i in E such that $\rho(e_i) = (\alpha, c_i)$; (iii) the fresh

edge e_i is associated to the value of i via the fresh property of $\mathsf{p}_{\mathsf{ansr}}$, i.e., $\mathsf{p}_{\mathsf{ansr}}(e_i) = i$; (iv) for multiset semantics, we additionally store the multiplicity of the corresponding answer via a fresh property on each node-for answer α; (v) in the case that the query contains an aggregate function in it's head, we additionally store the aggregate value via a fresh property for each aggregate function appearing in the head of the query.

4.1.5 Materialized Paths

A materialized view that introduces new edge labels between pairs of answer tuples to a property path query. In contrast to the strategy of edge labels (Section 4.1.3) this approach introduces edges and the corresponding labels between nodes that may not be previously connected via an edge. For the edge-labeled path $(u, \vec{v}_{m,n}, w)$, its label $(\vec{v}_{m,n} : l)$ in B_P, and the query:

$$\mathsf{V}_{\mathsf{pth}} : \quad \mathsf{l}_{\mathsf{pth}}(u, w) \leftarrow B_P \tag{4.5}$$

for each solution θ , we (i) add the fresh edge e' in E; (ii) extend ρ such that $\rho(e') = (\theta(u), \theta(w))$; (iii) and add a corresponding label $\mathsf{l}_{\mathsf{pth}}$ for the specific view;

The introduction of fresh edges between nodes may also be used for the following cases: (i) a fresh edge, may be used to materialize a view representing a hyperedge of the previous case with an arity of two. (ii) a fresh edge, may be used to materialize a view representing explicit θ-join operations on value predicate atoms.

Finally, we examine materialization strategies that assume that the materialized view corresponds to a fresh graph G', isomorphic to a subgraph of our initial graph G.

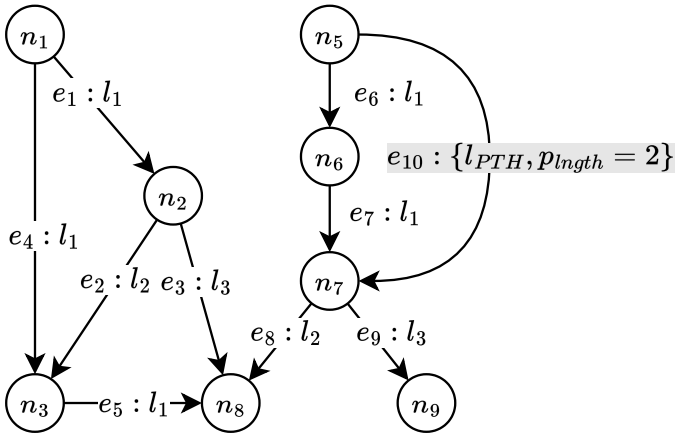

Figure 4.6: Path as View

4.1.6 Subgraphs as Views

A materialized view, that stores a subgraph, i.e., a set of nodes, edges, labels, and property values corresponding to the pattern of P. I.e., given the graph of G, and the smallest subgraph G_P of G such that $Q_P(G_P) = Q_P(G)$. By subgraph, we mean the subset of the nodes, edges, labels, and property values needed to get the same result for the query of Q_P. The materialized view G_V, is actually the a copy of the corresponding subgraph.

An extension of the previous approach assumes that we should explicitly keep track of the corresponding subgraph isomorphism. A straightforward approach to keep track of

the subgraph isomorphism is by introducing a key property (if there is none) that will allow to move between the two property graphs, i.e., the originating property graph of G and the view of G_V. Thus, the node v' in G_V corresponds to the node v in G if they have the same value on the key property. If we have a query Q that is contained in V, by keeping track of the isomorphism we can use multiple views to answer the query, or combine information from both G and G_V.

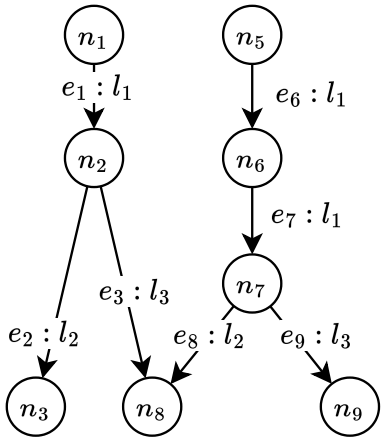

Figure 4.7: Materializing Subgraph

We provide an example that illustrates the different materialization alternative.

View Type	Query or Pattern Type			
	Acyclic	Cyclic	Edge Path	Aggregate
Node Labels	✓	✓	-	✓
Edge Labels	✓	✓	-	✓
Answer-Set Hyperedges	✓	✓	✓	✓
Materialized Paths	-	-	✓	-
Subgraph Materialization	✓	✓	✓	✓

Table 4.1: Query Rewriting Alternatives

Example 1.

For the variable names $u_1, u_2, u_3, u_4, \vec{v}_{1,\infty}, v_2, v_3 \in X$ and the labels $l_1, l_2, l_3 \in \mathbf{L}$ we assume the frequent graph-pattern query in Formula 4.6 where B_P is its corresponding body:

$$B_P = (u_1, \vec{v}_{1,\infty}, u_2), (u_2, v_2, u_3), (u_2, v_3, u_4), l_1(\vec{v}_{1,\infty}), l_2(v_2), l_3(v_3)$$
$$Q_p : \; q_P(u_2, u_3) \leftarrow B_P \tag{4.6}$$

The corresponding query is depicted in Fig. 4.1 and is applied to the graph database in Fig. 4.2. We analyze each of the view-materialization strategies:

The paradigm of Strategy 4.1.2 in Figure 4.3 will create two node indices, represented via the fresh labels $\mathsf{l}_{\mathsf{nd}_1}, \mathsf{l}_{\mathsf{nd}_2}$ to identify the nodes of G that satisfy the following query:

$$\mathsf{l}_{\mathsf{nd}_1}(u_1), \mathsf{l}_{\mathsf{nd}_2}(u_2) \leftarrow B_P \tag{4.7}$$

The paradigm of Strategy 4.1.3 in Figure 4.4 will introduce the fresh edge labels $\mathsf{l}_{\mathsf{edge}_1}$, $\mathsf{l}_{\mathsf{edge}_2}$, and annotate the edges that satisfy the corresponding query results:

$$\mathsf{l}_{\mathsf{edge}_1}(e_1), \mathsf{l}_{\mathsf{edge}_2}(e_2) \leftarrow B_P \tag{4.8}$$

A paradigm of Strategy 4.1.4 in Figure 4.5 will represent each answer of the query in Formula 4.6 by adding an additional node for each answer, the corresponding edges between the new node and the parts of the graph database that constitute the answer, and the fresh property values needed to represent multiplicity, or aggregate information. Therefore, for each answer vector $(n_{1,i}, n_{2,i}) \in N \times \mathbb{N}$ to the query in Formula 4.6: we create the fresh node of α_i in N; we add the edge $e_{1,i}$ in E such that $\rho(e_{1,i}) = (\alpha_1, n_{1,i})$; we add the edge $e_{2,i}$ in E such that $\rho(e_{2,i}) = (\alpha_1, n_{2,i})$; we add the property value $\sigma(e_{1,i}, p_{ans}) = 1$ that indicates that $n_{1,i}$ corresponds to the first variable within the answer vector; we add the property value $\sigma(e_{2,i}, p_{ans}) = 2$ that indicates that $n_{2,i}$ corresponds to the second variable within the answer vector; we add the property value $\sigma(\alpha_i, p_{\mathsf{mult}})$ that represents the multiplicity of the corresponding answer. Note, that this is the only strategy that only consider the information in th head of the frequent query pattern and not only its body.

A paradigm of Strategy 4.1.5 is illustrated in Figure 4.6 where the fresh label of $\mathsf{l}_{\mathsf{pth}}$ is created to indicate the paths connecting answer nodes according to the query of Formula 4.6. E.g., for each solution θ_i to the query of

$$\mathsf{l}_{\mathsf{pth}}(u, w) \leftarrow B_P \tag{4.9}$$

we will create the fresh edge of $e_i \in E$ such that $\rho(e_i) = (\theta_i(u), \theta_i(w))$ and label it with $\mathsf{l}_{\mathsf{pth}}$. Additionally to the $\mathsf{l}_{\mathsf{pth}}$ label, we explicitly keep track of the originating path's length range, i.e., the property $p_{\mathsf{range}} : \langle 1.\infty \rangle$ indicates that the corresponding materialized edge corresponded to a path of a length between 1 and ∞. For multiset semantics, we add additional information representing the multiplicity of the path.

A paradigm of Strategy 4.1.6 is illustrated in Figure 4.6 where instead of introducing labels and properties, or adding nodes and edges in the existing graph, the strategy assumes only the vital parts of the graph that are necessary for query answering. It should be noted that the available languages for querying property graphs cannot explicitly return the actual path corresponding to a path variable, out materialized subgaph will contain $\mathsf{l}_{\mathsf{pth}}$ labels according to the previous strategy.

4.2 Query Rewriting

We will now examine, depending on the available indices and materialized views, the different rewriting strategies that can be applied. For set semantics, in order to rewrite a conjunctive query based on a view, there need to exist a homomorphism from the view to

the conjunctive query. For multiset semantics, a sufficient condition is to have a subgraph isomorphism from the view to the corresponding query.

Given a conjunctive query of Q, a view of V, with B_Q, V_Q being the bodies of the query and the view, and an homomorphism or isomorphism $h : \mathsf{Vars}(V_Q) \rightarrow \mathsf{Vars}(Q)$. We now analyze the rewriting strategy for each of the cases:

4.2.1 Rewriting using Node Labels

Suppose that we label each answer to the view-definition in Formula 4.2 with l_{nd} and h is the corresponding homomorphism from the view-definition in Formula 4.2 to the query of Q that needs to be rewritten. We rewrite the query of Q to Q' by adding the label $\mathsf{l}_{\mathsf{nd}}(h(u))$ in its body. We need to remove every label atom $l(h(u))$ from Q when the rule $\mathsf{l}_{\mathsf{nd}}(x) \rightarrow l(x)$ appears in $\mathfrak{R}$. Similarly we need to remove every value-predicate atom $\rho(h(u).p, \tau)$ when the rule $\mathsf{l}_{\mathsf{nd}}(x) \rightarrow \rho(x.p, \tau)$ appears in $\mathfrak{R}$. The corresponding rewriting allows to focus our search into the parts of the graph that generate an answer.

4.2.2 Rewriting using Edge Labels

For edge labels, suppose that $\mathsf{l}_{\mathsf{edge}}(v)$ is the head of the view as in Formula 4.3 and h the corresponding homomorphism, we need to rewrite the Query of Q by adding the edge label $\mathsf{l}_{\mathsf{edge}}(h(v))$ in the body of the rewritten query. We additionally need to remove all the label or value-predicate atoms according to the rules in $\mathfrak{R}$, similar way as before.

4.2.3 Rewriting using Answer-Set Hyperedges

Suppose that our materialized view $\mathsf{V}_{\mathsf{ans}}$ corresponds to the answer-set of the query in Formula 4.4 and there exists a subgraph isomorphism ι from the body of the view $\mathsf{V}_{\mathsf{ans}}$ to the body of the query Q that needs to be answered. Let's assume that $\iota(\mathsf{V}_{\mathsf{ans}})$ is the corresponding subgaph of Q acquired via the isomorphism of ι. We employ the subgraph isomorphism ι when all the free variables appearing in V are mapped to variables in Q that do not appear outside $\iota(V)$. For the latter case, the rewriting Q' of Q is performed as follows: (i) we consider the fresh node-variable $u_r \notin \mathsf{Vars}(Q)$; (ii) for the i^{th} variable w_i appearing in the head of V, we add the edge atom $(u_r, v_i, h(w_i))$ in Q' for the fresh variable $v_i \in X \setminus \mathsf{Vars}(Q)$; (iii) for the variable $v'_i = h(v_i)$ we add the predicate atom $= (v'_i.\mathsf{p}_{\mathsf{ansr}}, i)$; (iv) finally we remove the subgraph $\iota(V)$ of Q from Q'. (v) In order to handle multiset semantics, we need to adjust the multiplicity of each answer to Q' by taking into account the multiplicity of the view answer.

4.2.4 Rewriting using Answer-Set Hyperedges with Aggregate Operations

We now examine the case that the view $\mathsf{V}_{\mathsf{ans}}$, represented as a hyperedge, contains an aggregate operation. For arbitrary aggregate operations in order for $\mathsf{V}_{\mathsf{ans}}$ to be used for answering the query of Q also containing an aggregate operation, there needs to exist an isomorphism between the two. The latter has been shown for relational databases [21] and also applies for graph databases.

4.2.5 Rewriting using Materialized Paths

Suppose that our materialized view $\mathsf{V}_{\mathsf{pth}}$ corresponds to the answer-set of the query in Formula 4.5 and there exists a subgraph isomorphism ι from the body of the view $\mathsf{V}_{\mathsf{pth}}$ to the body of the query Q that needs to be answered. The rewriting replaces each query It should be noted, that we restrict our study on queries that do not contain consecutive edge-path atoms that have the same path label. The rewriting of more complex queries has been studied in [17].

4.2.6 Rewriting using Subgraphs as Views

For the case that our materialized view is the subgraph of the initial graph that corresponds to the parts of the graph that are needed to find each and every solution, we once again consider two cases. (i) for the first case, we assume that the query to create the view and the asked query are equivalent; (ii) for the second case, we assume that the two are isomorphic, but The first type of query can be directly answered by using the graph corresponding to the materialized view. For the second type of query, we need to fetch additional information from the initial graph (or another available view). Thus, we need to keep track, i.e., by employing a primary key on existing nodes of the initial and materialized-view graph, to be able to fetch additional information between the two.

Example 1. We examine the available rewritings based on our view alternatives, assuming we have a query that is an extension of the one in Formula 4.6, e.g.:

$$Q': \quad q'(u_2) \leftarrow B_P \wedge (u_2, v_4, u_5)$$

The body of the query is rewritten as follows for each and every view:

$$\begin{aligned}
B' :=& (u_1, \vec{v}_{1,\infty}, u_2), (u_2, v_2, u_3), (u_2, v_3, u_4), (u_2, v_4, u_5) \\
& l_1(\vec{v}_{1,\infty}), l_2(v_2), l_3(v_3) \mathsf{I}_{\mathsf{nd}_1}(u_1), \mathsf{I}_{\mathsf{nd}_2}(u_2) \\
B'' :=& (u_1, \vec{v}_{1,\infty}, u_2), (u_2, v_2, u_3), (u_2, v_3, u_4), (u_2, v_4, u_5) \\
& l_1(\vec{v}_{1,\infty}), \mathsf{I}_{\mathsf{edge}_2}(v_2), \mathsf{I}_{\mathsf{edge}_3}(v_3) \\
B''' :=& (u_1, v_1, u_2), (u_2, v_2, u_3), (u_2, v_3, u_4), (u_2, v_4, u_5) \\
& \mathsf{I}_{\mathsf{pth}}(v_1), l_2(v_2), l_3(v_3) \\
B'''' :=& l_{ANS}(n), (n, v_{n_1}, u_1), (n, v_{n_2}, u_2), (n, v_{n_3}, u_3), \\
& (n, v_{n_4}, u_4), (u_2, v_4, u_5), l_{ANS_1}(v_{n_1}), l_{ANS_2}(v_{n_2}), \\
& l_{ANS_3}(v_{n_3}), l_{ANS_4}(v_{n_4})
\end{aligned}$$

For the case of the view being a materialized subgraph G' of the initial graph of G, the rewriting evaluates the actual query on the materialized subgraph. It should be noted that for the query in Formula 1, the evaluation of Q' on G' will actually produce an empty (incomplete) answer. For subgraphs as views, the cases that the rewritten query Q' produces a complete answer are either the ones that Q and Q' have isomorphic bodies, or the cases that there exists a subgraph isomorphism from the body of Q' to the body of Q.

5. VIEW SELECTION & MATERIALIZATION

In Section 4 we studied indexing and view-materialization alternatives. In this section, our focus will be on exploring the patterns that an algorithm designed for view selection takes into account. By precomputing relevant query fragments based on these patterns, the system accelerates subsequent query executions. Integrating view selection algorithms with frequent patterns enhances query performance, reducing computational overhead and ensuring responsive information retrieval.

5.1 Finding Recurring Patterns Within the Query Workload

Various methods have been proposed for finding recurring patterns within a query workload.

Several multi-query optimization algorithms rely on the properties of relational algebra, that allow to rewrite a query in various equivalent forms. These techniques enable to rewrite multiple queries in ways that reveal recurring patterns in them. For instance, if we have the queries Q_1 and Q_2, we can rewrite them to reveal the pattern of P by also renaming the variable names if needed:

$$\begin{aligned} Q_1 &: q(u_1, w_1, w_2) \leftarrow (u_1, v_1, w_1), (u_1, v_2, w_2), l_1(v_1), l_2(v_2), l_3(w_2) \\ Q_2 &: q(u'_1, w'_1, w'_2) \leftarrow (u'_1, v'_2, w'_2), (u'_1, v'_1, w'_1), l_1(v'_1), l_2(v'_2), l_4(w'_2) \\ P &: \quad (u_1, v_1, w_1), (u_1, v_2, w_2), l_1(v_1), l_2(v_2) \end{aligned}$$

Such techniques have been extensively employed in the domain of multi-query optimization (e.g. [78, 63, 3]). Intuitively, if there exists a subgraph isomorphism between two execution plans, there will also be a subgraph isomorphism between their corresponding queries.

Various alternatives for frequent subgraph mining have been suggested to accelerate the process of frequent pattern mining that introduce randomness into the process based on genetic algorithms and random walks. Finally, datalog-based methodologies have been suggested for finding frequent patterns within a query workload. These methodologies employ datalog in order to reveal recurring patterns as well as for query rewriting. The main advantage of these methodologies is that allow us to introduce complex rules in our mining algorithm. Their main disadvantage is that they do not consider any specific frequent subgraph mining optimizations inherent in the pattern mining process.

5.1.1 Our Approach

Our model is based on [56] that frequent pattern mining algorithms are directly applied to a query workload of $\mathcal{Q}$ in order to identify recurring patterns as well as creating efficient summaries of the initial workload that will allow to better identify the benefit of each view. Intuitively, a summary $\mathcal{Q}_S$ of the query workload of $\mathcal{Q}$ allows to efficiently compute the benefit of a materialized view by examining the summary instead of the initial query workload.

It should be noted that the focus of the paper is not on frequent pattern mining, rather than what should be the characteristics of the selected views and indexes based on the characteristics of the frequent patterns within our workload.

5.1.2 Frequent Pattern Mining

The problem of frequent pattern mining is defined as follows [97]: *"Given a graph dataset,* $D = \{G_0, G_1, \ldots, G_n\}$*,* Support$(g)$ *denotes the number of graphs in* D *in which* g *is a subgraph. The problem of frequent subgraph mining is to find any subgaph* g *s.t.* Support$(g) \geq$ *minSup (a minimum support threshold)."*

To find frequent query patterns within the query workload of $\mathcal{Q} = \{Q_0, Q_1, \ldots, Q_n\}$ we first need to represent it in the form of a labeled graphs conforming to the definition in [97]. Thus, each query Q_i will be transformed as accordingly in order to represent for multiple labels per node, as well as, node and edge properties.

5.1.3 Query-Workload Summary

Frequent query patterns play a dual role in the view selection process: they provide the candidates for creating views and indexes but also allow our algorithm to represent in compact form a query workload $\mathcal{Q}$ via a smaller multiset of *frequent query patterns* $\mathcal{Q}_\mathbf{P}$. Thus, instead of the initial workload $\mathcal{Q}$ we examine the workload of $\mathcal{Q}_\mathbf{P}$ that contains frequent patterns along with a corresponding multiplicity corresponding to each patterns frequency. For example, if we have the queries $Q_1, Q_2, \ldots Q_k$ and the pattern of Q_P appearing in all of them, we will employ the query Q_P as their corresponding representative with a multiplicity of k.

It should be noted that for the query patterns Q_P and Q'_P, such that the pattern of Q_P appears in Q'_P, there is a need to adjust Q_P's multiplicity in our summary. Thus, if $m(\cdot)$ returns the multiplicity of a query, we update the multiplicity of Q_P as follows $m(Q_P) := m(Q_P) - m(Q'_P)$[1]. The aforementioned adjustment ensures that a query that a frequent pattern P appearing in the initial and the summarized workload, will have the same support in both of them. To correctly adjust our summary in the case of multiple frequent patterns, we need to build a hierarchy based on subgraph isomorphism and update it from top to bottom.

5.2 View and Index Selection

Having defined the query rewriting process in Section 4.2, we proceed to designate the view/index-selection methodology.

5.2.1 Cost Model

The purpose of view materialization is the reduction of the execution cost of a set of queries $\mathcal{Q}$. The execution cost of a query, Cost(Q), is determined by considering various aspects such as: the number of disk accesses; the query execution time taken by the CPU; the involved communication costs in distributed and parallel database systems. A database cost estimation function, Cost$^\epsilon(\cdot)$,commonly known as a *cost model*, is a fundamental component of a graph-database management system used to estimate the execution cost of a specific query operation before it is actually executed. The main purpose of the cost

[1]It should be noted that in case $m(Q_P) = m(Q'_P)$, Q_P is a closed pattern, that does not need to be examined and represented within the summary

model function is to approach the actual execution cost of a query so as to aid the query optimizer in making decisions about the most efficient execution plan for a given query.

5.2.2 View Benefit

Given the query of Q, the set of views $\mathcal{V}$ and the rewriting function of Rwrt$(\cdot,\cdot)$, $Q' =$ Rwrt$(Q,\mathcal{V})$ is the *rewriting* of Q using the views appearing in $\mathcal{V}$. The benefit of rewriting the query of Q based on the views appearing in $\mathcal{V}$, Bnft$(Q,\mathcal{V})$, corresponds to the reduction in the execution cost of the query, while the overall benefit for the query workload of $\mathcal{Q}$ using the view appearing in $\mathcal{V}$ corresponds to the overall reduction of the queries in Q:

$$\begin{aligned} \text{Bnft}(Q,\mathcal{V}) &= \text{Cost}(Q) - \text{Cost}(Q') \\ \text{Bnft}(Q,\mathcal{V}) &= \sum_{Q\in\mathcal{Q}} \text{Bnft}(Q,\mathcal{V}). \end{aligned} \tag{5.1}$$

We will employ Bnft$^{\epsilon}$ instead of Bnft in order to denote that our view-selection process employs the estimated and not the actual benefit of a query or a set of queries, i.e., the benefit based on the Cost$^{\epsilon}$ estimation function.

5.2.3 View Selection

Given the query workload $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$; a set of candidates views $\mathcal{W} = \{V_1, V_2, \ldots V_n\}$ with their corresponding indexes; and a storage capacity of b: the view selection problem is to find the subset $\mathcal{V} \subseteq \mathcal{W}$ that requires a storage capacity less than b and benefits the query workload $\mathcal{Q}$ the most.

In our implementation, when considering the benefit of a view V (set of views $\mathcal{V}$) to a query workload $\mathcal{Q}$, we employ the summary of the query workload based on the frequent patterns within our workload (Section 5.1.3). The candidate views are derived from the frequent patterns in P using the materialization alternatives presented in Section 5.1.3.

5.2.4 Knapsack

The 0-1 knapsack, is classic constraint optimization problem problem, which consists of a set of items, each with a weight and a value. The goal is to determine the combination of items to include in the knapsack that maximizes the total value, without exceeding the weight capacity of the knapsack. In the 0-1 knapsack, each item must be completely included in the knapsack or ignored.

The problem of view selection can be readily simplified to a 0-1 knapsack problem. As in [22], the view selection problem can be formulated in the form of a 0-1 knapsack problem, where the size of the knapsack is the space budget for the materialized views. The items that we want to fit in the knapsack are the candidate views, the weight of each item is the view's storage cost, while the value of each item is the view's benefit.

5.2.5 Maximizing a Set Function Subject to a Knapsack Constraint

We will now present the problem of *Maximizing a Set Function Subject to a Knapsack Constraint* (MSFSKC) and showcase the reduction from the view-selection problem to the

aforementioned *knapsack* variation.

5.2.6 MSFSKC

Let $I = \{1, 2, \ldots, n\}$; $i \in I$ and b be nonnegative integers; and $f(\cdot)$ be a set function. MSFSKC is the following optimization problem:

$$\max_{S \subseteq I} \left\{ f(S) : \sum_{i \in S} c_i \leqslant b \right\}. \tag{5.2}$$

5.2.7 Reduction

We now identify the parameters of the reduction from the view-selection problem to MSFSKC. I. For the set of candidate views $\mathcal{W} \leftarrow \{V_1, \ldots, V_n\}$ such that $|\mathcal{W}| = n$, we define the set of integers $I := [\![1, n]\!]$ and a bijection function Bi : $I \leftrightarrow \mathcal{W}$. For simplicity, for $i \in I$ and $V_i \in \mathcal{W}$ we assume that Bi$(i) = V_i$. II. For each $i \in I$ we define $c_i =$ Strg(V_i) to be the corresponding view's storage space, i.e., the required storage for V_i's materialization. III. Each subset $S \subseteq I$ is mapped via the Bi function to a subset of the candidate indices $\mathcal{V} \subseteq \mathcal{W}$, by mapping each $i \in S$ to $V_i \in \mathcal{V}$. IV. For the corresponding subset, we define $f(S)$ to be the estimated benefit of the materialized views in $\mathcal{V}$ to the query workload in $\mathcal{Q}$, i.e., $f(S) =$ Bnft$(\mathcal{Q}, \mathcal{V})$. V. Finally, b is the available storage space for materialization. It is straightforward to show that by solving the formula in 5.2 we acquire an optimal solution for the view selection problem.

Algorithm 1 View Selection

```
1:  function ViewSelection(StorageSpace b,    CandidateView W, MaterializedView V)
2:      W := {I ∈ W : Cost(I) ⩽ b}
3:      if W = ∅ then return V
4:      θ_max := 0
5:      for all I ∈ W do
6:          θ := (Bnft(Q,V∪{V}) − Bnft(Q,V)) / Size^ϵ(V)
7:          if θ > θ_max then
8:              V_t := V
9:              θ_max := θ
10:     return ViewSelection(b − Cost(I), W \ {V_t}, V ∪ {V_t})
```

5.2.8 MNssfKc with an Approximation Solution

Sviridenko proves that there exists a $(1 - e^{-1})$-approximation algorithm for maximizing formula 5.2 when $f(S)$ is a non-negative, non-decreasing, polynomially-computable, submodular set function [87].

5.2.9 Properties

In order for the $(1 - e^{-1})$-approximation to apply, there is the need for the Bnft$^\epsilon$ function to be non-negative, non-decreasing, polynomially computable, and submodular. Bnft$(\mathcal{Q}, \mathcal{V})$

is: I. non-negative since every query Q is a valid rewriting of itself; II. non-decreasing since for the sets of views $\mathcal{V} \subseteq \mathcal{V}'$, each rewriting of Q using $\mathcal{V}$ is also a valid rewriting of Q using $\mathcal{V}'$; In order for the Bnft(f,u) nction to be polynomially computable , its corresponding rewriting algorithm, i.e. Rwrt and cost mode, i.e. Cost$^{\epsilon}$, need also be of polynomially computable.

5.3 Lazy View Materialization

We will now describe an optimization concerning the materialization of a selected index after its selection. Specifically, a selected candidate index needs to be inserted into the database in the form of a query, which might prove to be a costly operation when executing a query per index. However, this additional cost can be avoided by creating the given index when answering a query that can be benefited from it. This lazy approach to the index materialization fully utilizes the selected indexes, by executing queries that were going to be ran against the database regardless of the indexing process and providing future queries with the newly created index. This process can be formalized in the following way: Given a selected index I_k that can accelerate a set of queries $Q_k = \{q_{k_1}, q_{k_2}, ..., q_{k_n}\}$ that may be ran against the database in random order. The index can be implemented into the database by its equivalent creation query c_k. To avoid the execution cost of c_k and the redundant use of database resources we employ the following tactic: We discard the query c_k and wait for the first query in Q_k to appear. When a query $q_{k_i} \exists Q_k$ is the first query of Q_k to be executed, we augment it with additional information needed to create the index I_k. Subsequently, I_k is materialized and all queries in Q_k can be benefited from the new index.

6. EVALUATION

6.1 Goal

The objective of our experimental evaluation is to analyze the efficacy of diverse materialization alternatives, focusing on different parameters of the view evaluation process. Specifically, we compare various view-materialization alternatives with respect to: (i) the inherent characteristics of the queries within the different query workloads; (ii) the features encapsulated in the underlying dataset; (iii) the attributes inherent in different property-graph databases; (iv) the efficiency of each view alternative concerning the designated space budget. To ensure the thoroughness of our evaluation, we conducted a detailed analysis of our view alternatives on many different datasets.

6.2 Experimental Setup

6.2.1 Methodology

For our testing scenarios, our application takes as input a knowledge graph G; a query workload $\mathcal{Q}$ and produces candidate views V, by mining the most frequent patterns in $\mathcal{Q}$. The effectiveness of the candidate views in V is then tested for all view alternatives, comparing the query execution cost and storage cost between them.

The basic scenario we considered regarding the the storage of the knowledge graph G into a graph database storage engine was the following: The triples representing the knowledge graph were stored into the Neo4j system with emphasis on maintaining their RDF form. Specifically: *(entity, type, class)* are stored as node labels on the entity node, *(entity, relation, entity)* are stored as Neo4j relationships, namely as edges between the two entity nodes and *(entity, property, value)* are stored as a relationship of the given entity node between a node containing the given value. Datasets such as Roadnet-USA, LiveJournal, and Colours, which were initially not structured as knowledge graphs, underwent a transformation process to be represented as such. Subsequently, the aforementioned procedure was followed.

6.2.1.1 Property Graph Database Setting

We now analyze the Neo4j GDMS, as a representative of the native property graph database systems:

6.2.2 Neo4j

Neo4j, a leading graph database, employs a native graph model that uniquely organizes data into nodes, relationships, and properties. Nodes signify entities, relationships define connections, and properties store key-value pairs, making Neo4j ideal for scenarios with complex relationships, such as social networks or recommendation engines. The native graph model, coupled with the expressive Cypher query language, enables fast graph traversal and information extraction. With an emphasis in scalability and performance, Neo4j is widely used in diverse industries seeking to utilize the benefits of native graph storage

for applications involving complex data relationships. Additionally, Noe4j employees the index-free adjacency property. A DBMS utilizes the aforementioned property when each node maintains direct references to its adjacent nodes. This results in two facts:

- Each node acts as a micro-index, which is much cheaper than global indexes.
- Query times are independent of the total size of the graph, they are proportional to the amount of graph searched

6.2.3 Hardware and memory

We deployed our implementation on a single system of 1 11th Gen Intel(R) Core(TM) i5-11400 CPU @2.6GHz with 12 cores/20 threads per CPU and 24GB of main memory. The DBMS used to store the data is Neo4j Community v4.4.30 database running on the same computer.

6.2.4 Implementation Setup

We have implemented our algorithm in Java 17 using the Apache Jena 4.0.0 open source Semantic Web framework [39] to parse SPARQL query workloads and translate them to the equivalent Cypher queries. For efficiently, computing containment mappings from a set of indexes I to an examined query, we have employed the MV-index structure introduced in our previous work [57].

6.3 Datasets and Workloads

6.3.1 Datasets

To benchmark our methodology, we utilized a range of openly available datasets and query workloads. Additionally, we created the Colours dataset for a more thorough experimental evaluation of edge cases. These datasets include: (i) The DbPedia semantic knowledge graph [23, 9]; (ii) A combination of the BioModels and BioPortal datasets from the Bio2RDF semantic knowledge graph [11, 12]; (iii) The generated Colours dataset that we created to examine the scenario of cyclic queries with interleaved solutions.

6.3.2 Property Graphs

DbPedia and Bio2RDF are two large real-life knowledge graphs containing a multitude of classes and complex schemata, that are both widely used and studied both in industrial and academic settings. The DBpedia contains, in the form of an RDF graph, information extracted from Wikipedia articles. Bio2RDF is an RDF dataset that constitutes the largest network of linked data for the Life Sciences. Bio2RDF defines its knowledge graph from a diverse set of heterogeneously formatted sources obtained from multiple data providers. When translated into property-graph models, these datasets encompass a wide variety of node labels and relationship types, leading to two substantial heterogeneous graphs.

In table 6.1 we present the statistics of the two datasets employed in our experimental evaluation along with the generated Colours dataset. We showcase the number of nodes,

edge, node labels and edge labels for each dataset. We also show the size on disk that each dataset occupies. Entries with a star, denote that the dataset is generated.

Dataset	Nodes	Edges	Node Labels	Edge Labels	Size On Disk
DbPedia	132M	730M	438	52k	114GB
Bio2RDF	9.3M	45.5M	2227	1928	5.54GB
Colours *	100k	500k	3	1	37.7MB

Table 6.1: Dataset Statistics

6.3.3 The Colours Dataset

Many of the real-world datasets we examined depict graph structures where two solutions to a cyclic query do not *interleave*. This means that different solutions map the same edge-atom of a conjunctive query to different edges in the actual graph. In these datasets, we observed similar performance between edge-labels and answer-set hyperedges as views. To explore the limitations of both techniques, we constructed an artificial dataset where solutions to a simple conjunctive query, such as finding triangles, are interleaved.

The conjunctive query on which our generated dataset is based requests all triangles formed by nodes labeled red, blue, and green:

$$q(u_1, u_2 m u_3) \leftarrow (u_1, v_1, u_2), (u_2, v_2, u_3), (u_3, v_3, u_1), \\ red(u_1), blue(u_2), green(u_3). \tag{6.1}$$

For the dataset, we have the following features: each red node has k outgoing edges to blue nodes; each blue node has k outgoing edges to green nodes; and each green node has k outgoing edges to red nodes; each node appears in exactly k solutions to the query in Equation 6.1.

On the left-hand side, Figure 6.1 displays a generated dataset consisting of 90 nodes. Each node has 4 outgoing (and 4 incoming) edges, appearing in exactly 4 solutions to the query in Equation 6.1. On the right-hand side, Figure 6.1 displays its edge-labeled form, where each edge-view is labeled with the same color as its corresponding source node. When examining the dataset with edge-labeled views, starting from a red node, we need to consider 4 red edges, 16 blue edges, and 64 green edges. In the general case that our query represents a polygon with p sides and each node has k outgoing edges, we need to examine $O(k^{p-1})$ edges for each node. In contrast, when using answer-set hyperedges as views, we examine precisely the edges that constitute the corresponding solution.

6.3.4 Query Workloads

To test the efficiency of view and index alternatives we employed various real-world and generated query workloads. For DbPedia we used a corresponding real-world query workload [24], originating from queries on the DbPedia knowledge graph. For BioModels+BioPortal we used the corresponding logs from a real-world query workload taken provided by LSQ [84, 55].

Additionally, we had to generate the queries encompassing property paths and cycles for both knowledge graph datasets. Regarding the Colours datasets, we constructed all query

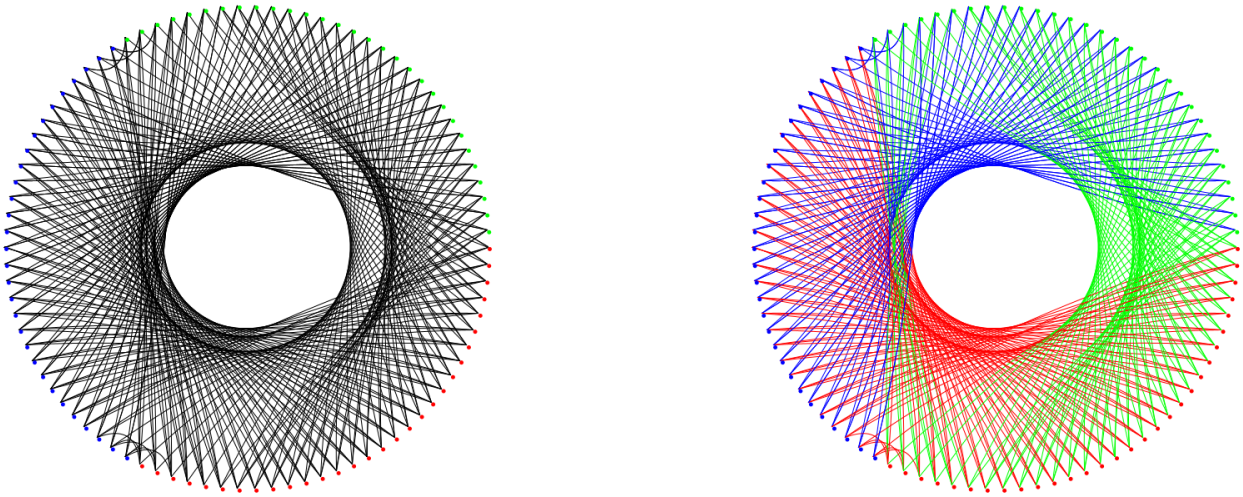

Figure 6.1: On the left-hand side: the generated-property graph corresponding to the case of interleaved solutions for triangular queries. On the right-hand side: its corresponding edge-labeled views.

categories outlined in Section 4. This decision stemmed from the absence of real-world query workloads for the Colours dataset, since it was internally designed by our team. Table 6.2 displays the statistics for the query workloads employed in our experiments, including the counts of acyclic, cyclic, and edge path queries. Entries marked with a star indicate generated queries.

Dataset	Acyclic	Cyclic	Edge Path
DbPedia	1.2M	120k*	112k*
Bio2RDF	1.5M	104k*	13.5k*
Colours	26k*	64k*	52k*

Table 6.2: Query Workload Statistics

6.4 Experiments

6.4.1 View Efficiency on Different Query Types

In our first set of experiments, we study the performance of Neo4j on the datasets and workloads discussed in the previous section. We examine, for various types of queries, the improvement in query execution time and the total storage cost required for each view alternative. We measure query-response times based on the rows needed to be collected from the database for the query to be completed.

6.4.2 Evaluation Parameters

To explore varying candidate patterns for materialization within each dataset, we employed the GSpan frequent pattern mining algorithm, adjusting the minimum support (minSup) accordingly. Specifically, for DbPedia, we set minSup to 500, for BioModels + BioPortal it was 150, and for the Colours dataset, it was 100. Consequently, this yielded 658 view candidates for DbPedia, 250 for BioModels + BioPortal, and 13 for the Colours dataset.

6.4.3 Comparison of View and Index Alternatives

We now examine the quality of the candidate views and indexes by generating for each pattern all possible view and index alternatives, rewriting the query patterns in all possible alternatives, for each query type.

In figure 6.2, we compare the execution costs of all the different view alternatives for each of the four query types as discussed in Section 4, for the DbPedia dataset and workload. The x-axis represents the rows needed to be collected from the database for the query to be completed, while the y-axis represents the query type. It can be observed that subgraph materialization outperforms all other alternatives in all cases but the Acyclic queries, where the edge label indexes were 18.51% faster. Additionally, our observations indicate that node labels consistently underperformed across various cases where they were applicable. Particularly notable was their performance in Cyclic queries, where 3x times more rows collected from the database were required to answer the query, than the second worse alternative.

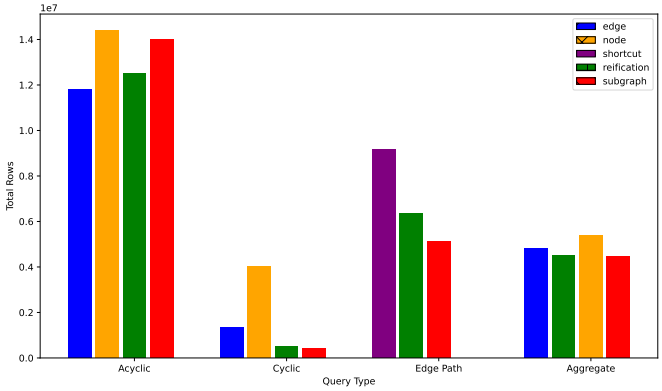

Figure 6.2: DbPedia Views Alternatives

In figure 6.3, we depict the same experiment conducted on the Bio2RDF dataset and workload. Notably, our observations revealed that subgraph materialization consistently ranked as the least effective method across most query types. Conversely, the most efficient alternative proved to be answer-set hyperedges, showcasing a substantial reduction in required rows compared to subgraph materialization, ranging from 0.5x to 4.45x fewer rows needed to answer the query. Additionally, it's noteworthy that for edge path queries, materialized paths emerged as the most efficient alternative, exhibiting a 35.12% improvement in performance over the answer-set hyperedges alternative.

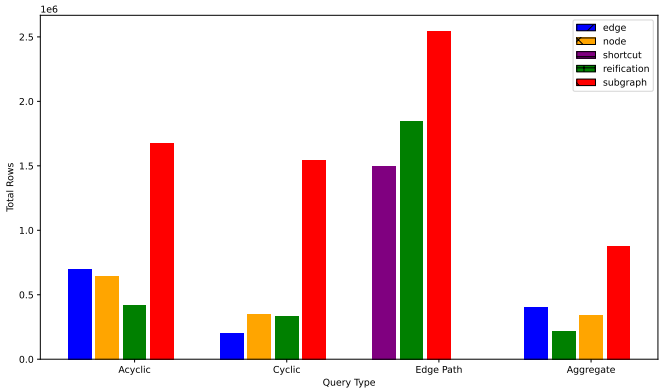

Figure 6.3: Bio2RDF View Costs

In figure 6.4, we present the results of the experiment conducted on the Colours generated dataset and workload. Our observations indicate that across acyclic queries and aggregate queries, all view alternatives exhibited similar performance, with a slight edge in efficiency noted for answer-set hyperedges. However, when it comes to cyclic queries, subgraph materialization significantly underperformed, with the reification alternative demonstrating a 58.82% increase in efficiency compared to subgraph materialization. Conversely, for edge path queries, subgraph materialization emerged as the most efficient alternative, with an 8% improvement over the reification alternative.

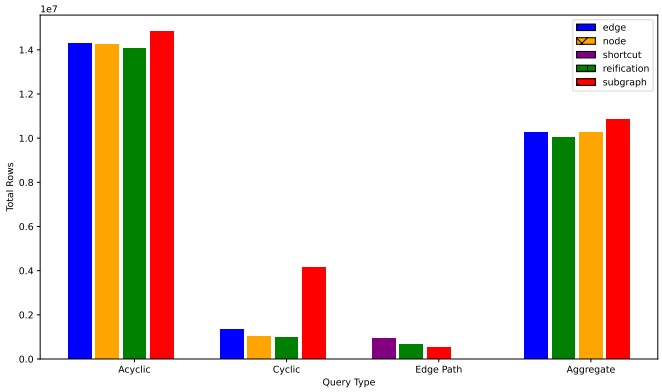

Figure 6.4: Colours Views Alternatives

6.4.4 Storage Cost of View and Index Alternatives

We proceed to analyze the storage cost in bytes of the candidate views and indexes. This calculation entails assessing the cost of each pattern by aggregating the node and edge records inserted into the database and then multiplying them by their respective byte sizes, as outlined in [77]. In figure 6.5, we present a comparative analysis of storage costs between view and index alternatives across various query types. The x-axis denotes the total cost in bytes, while the y-axis represents different query types. It is evident that in the majority of cases, subgraph materialization emerges as the most expensive solution due to its requirement of creating a complete duplicate of all information in the result set, encompassing nodes, edges, and properties. As anticipated for the Neo4j GDMS, the Node Label index alternative proves to be the most economical across most scenarios, exhibiting storage costs ranging from 2.5x to 2.9x less than the second-best alternative. Particularly in the case of cyclic queries, subgraph materialization presents itself as the most cost-effective solution, boasting a 2.5x reduction in storage costs compared to the second-best alternative.

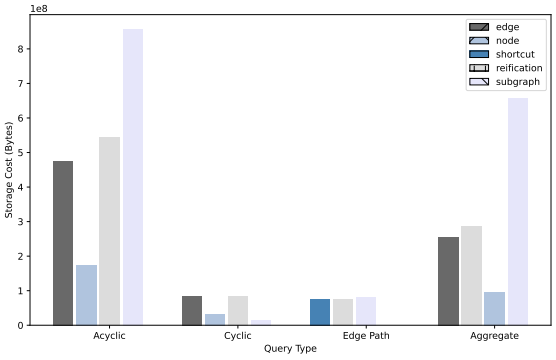

Figure 6.5: DbPedia View Costs

In figure 6.6, we illustrate the replication of the experiment performed on the Bio2RDF dataset and workload. The findings were similar to those outlined in the DbPedia cost section. Once more, node labels emerged as the most cost-effective option across all applicable cases. Notably, in edge path queries, the answer-set hyperedges (reification) proved to be the most economical alternative, boasting a remarkable 72.98% reduction in cost compared to the runner-up alternative.

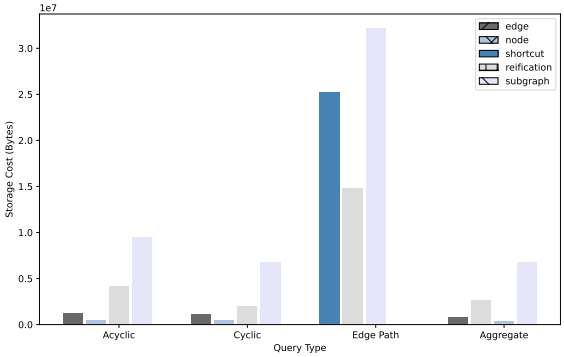

Figure 6.6: Bio2RDF View Costs

In figure 6.7, we illustrate the replication of the experiment performed on the Colours dataset and workload. The findings were again similar to those outlined in the DbPedia cost section. Once more, node labels emerged as the most cost-effective option across all applicable cases. However, in cyclic queries, the subgraph materialization proved to be the cheapest alternative, with a 2x reduction in cost compared to the runner-up alternative and a 5x reduction compared to the edge label and reification alternatives.

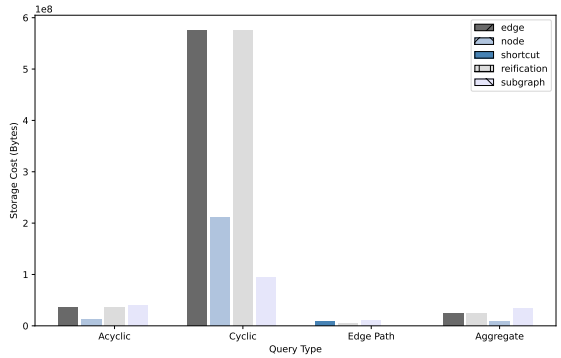

Figure 6.7: Colours View Costs

6.4.5 Effectiveness of Lazy Indexing

Finally, we examine the proposed optimization method considering the materialization of a selected index. For this experiment we used a test set of 1000 queries selected randomly from the 62830 queries we used as our test workload, using the DbPedia dataset and workload. In our experiments we compared the index creation and query execution time of the aforementioned queries with and without the lazy index method. This optimization method presented improved efficiency in execution time and storage space without any significant trade-off. In Figure 6.8, the x-axis represents the indices selected and materialized, while the y-axis represents the overall elapsed time in minutes. In the Query Execution and Index Creation indicate the elapsed time to complete the query execution and index materialization into the database, respectively. Lazy Method represents the

overall time passed for both the execution of the queries alongside the creation of the indices as described in 5.3. From our experiments we discovered that the lazy method enhances greatly the performance on time, since indices will only be created only when needed. Additionally, storage space is saved, since indices that are never requested in future queries will not be inserted into the database.

Figure 6.8: Lazy Materialization Performance in DbPedia

7. CONCLUSIONS AND FUTURE WORK

In our work we studied a variety of view and indexing alternatives for native GDMSs. We examined strategies for view and index materialization in graph databases (GDMS), aiming to enhance query performance. We explored various materialization alternatives based on query patterns, including node and edge labels, answer-set hyperedges, materialized paths, and subgraphs. Each strategy is analyzed in detail, considering their implementation and impact on query rewriting. Furthermore, we provided examples illustrating the application of these strategies and their implications for query optimization. Additionally, we discussed query rewriting techniques tailored to each materialization strategy, emphasizing the importance of homomorphisms and subgraph isomorphisms for efficient query processing. We investigated view selection algorithms and techniques, drawing inspiration from variations of the knapsack problem. Our exploration led us to consider how the view materialization problem can be effectively framed and addressed as a form of the knapsack problem. Finally, we consider a lazy index/view materialization strategy that materializes structures during query execution and only if the corresponding structures are being asked for at least one time. The latter optimization allows to avoid materializing views based on patterns of a high expected number of appearances, that do never appear in practice. Our experimentation underscored the underperformance of subgraph materialization, the prevailing view type in GDMSs, across various query types. However, by introducing alternative approaches, we achieved a remarkable enhancement of up to 4.45 times in query execution efficiency and over a 2x reduction in storage costs. This suggests that adopting a more flexible view and indexing model can yield significant improvements in performance and storage efficiency.

In our future work we intend to study view-selection techniques for streaming graphs [2], focusing on stream processing for Semantic Web applications [47, 45, 46]; as well as complex event processing [50]. Furthermore, we aim to explore the applicability of view and index selection techniques in geospatial knowledge graphs, such as those discussed in works like [42]. Our goal is to enhance query performance in geospatial triple stores like Strabon [51] or GraphDB [32] through such approaches, working with query workloads like GeoQuestions1089 [44]. Additionally, we intend to integrate approximate counters [90] into our index/view-selection methodology that will be used by our cost-estimation function and examine entropy-based techniques when computing the benefit of different view alternatives [14]. Finally, we intend to generalize our work towards more sophisticated pattern-mining methods for streaming-subgraph pattern mining (survey [28]).

ABBREVIATIONS - ACRONYMS

W3C	World Wide Web Consortium
NKUA	National and Kapodistrian University of Athens
DBMS	Database Management System
GDMS	Graph Database Management System
MNssfKc	Maximizing a Nondecreasing Submodular Set Function Subject to a Knapsack Constraint
BGP	Basic Graph Pattern
EXP	EXPTIME
SPARQL	SPARQL Protocol and RDF Query Language
RDF	Resource Description Framework
KG	Knowledge Graph
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

BIBLIOGRAPHY

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[2] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[3] S Agarawal, S Chaudhuri, and V Narasayya. Automated selection of materialized views and indexes for sql databses. In *Proceedings of 26th International Conference on Very Large Databases, Cairo, Egypt*, pages 191–207, 2000.

[4] Allegrograph. `https://franz.com/agraph/allegrograph/`.

[5] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[6] Renzo Angles. The property graph database model. In *AMW*, 2018.

[7] ArangoDB. `https://arangodb.com/`.

[8] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50. ACM, 2010.

[9] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, pages 722–735, 2007.

[10] Murat Ali Bayir, Ismail H Toroslu, and Ahmet Cosar. Genetic algorithm for the multiple-query optimization problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(1):147–153, 2006.

[11] Fran$\vec{c}$cois Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *J. Biomed. Informatics*, 41(5):706–716, 2008.

[12] Bio2Rdf Release 3. `https://download.bio2rdf.org/#/current/`.

[13] BlazeGraph. `https://www.blazegraph.com/`.

[14] Dritan Bleco and Yannis Kotidis. Using entropy metrics for pruning very large graph cubes. *Information Systems*, 81:49–62, 2019.

[15] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large sparql query logs. *PVLDB*, 11(2):149–161, 2017.

[16] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68, 2002.

[17] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Containment of conjunctive regular path queries with inverse. *KR*, 2000:176–185, 2000.

[18] Leonardo Weiss F Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. Towards materialized view selection for distributed databases. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*, pages 1088–1099, 2009.

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[20] Rada Chirkova, Alon Y Halevy, and Dan Suciu. A formal perspective on the view selection problem. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(3):216–237, 2002.

[21] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 155–166, 1999.

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

[23] Dbpedia 3.9, 2019. [Online; accessed 16-September-2021].

[24] Dbpedia log, 2012. [Online; accessed 16-September-2021].

[25] Kirill Degtyarenko, Janna Hastings, Paula de Matos, and Marcus Ennis. Chebi: an open bioinformatics and cheminformatics resource. *Current protocols in bioinformatics*, 26(1):14–9, 2009.

[26] Vicky Dritsou, Panos Constantopoulos, Antonios Deligiannakis, and Yannis Kotidis. Optimizing query shortcuts in rdf databases. *ESWC*, pages 77–92, 2011.

[27] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9(1):77–129, 2018.

[28] Philippe Fournier-Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Chun-Wei Lin, and Unil Yun. A survey of pattern mining in dynamic graphs. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 10(6):e1372, 2020.

[29] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[30] Apache jena fuseki. `https://jena.apache.org/documentation/fuseki2/`.

[31] Fran$\vec{c}$cois Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[32] GraphDB. `https://graphdb.ontotext.com/`.

[33] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory*, pages 98–112. Springer, 1997.

[34] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, pages 289–300. ACM, 2014.

[35] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25:205–216, 1996.

[36] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[37] Zan Huang, Wingyan Chung, Thian-Huat Ong, and Hsinchun Chen. A graph-based recommender system for digital library. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 65–73, 2002.

[38] JanusGraph. `https://janusgraph.org/`.

[39] Apache Jena. semantic web framework for java, 2007.

[40] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, 2018.

[41] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, pages 191–203, 2018.

[42] Nikolaos Karalis, Georgios M. Mandilaras, and Manolis Koubarakis. Extending the YAGO2 knowledge graph with precise geospatial knowledge. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefran$\vec{c}$cois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2019.

[43] Tarun Kathuria and S Sudarshan. Efficient and provable multi-query optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 53–67, 2017.

[44] Sergios-Anestis Kefalidis, Dharmen Punjani, Eleni Tsalapati, Konstantinos Plas, Mariangela Pollali, Michail Mitsios, Myrto Tsokanaridou, Manolis Koubarakis, and Pierre Maret. Benchmarking geospatial question answering engines using the dataset geoquestions1089. In Terry R. Payne, Valentina Presutti, Guilin Qi, María Poveda-Villalón, Giorgos Stoilos, Laura Hollink, Zoi Kaoudi, Gong Cheng, and Juanzi Li, editors, *The Semantic Web - ISWC 2023 - 22nd International Semantic Web Conference, Athens, Greece, November 6-10, 2023, Proceedings, Part II*, volume 14266 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2023.

[45] Evgeny Kharlamov, Yannis Kotidis, Theofilos Mailis, Christian Neuenstadt, Charalampos Nikolaou, Özgür Özc̈ep, Christoforos Svingos, Dmitriy Zheleznyakov, Sebastian Brandt, Ian Horrocks, Yannis E. Ioannidis, Steffen Lamparter, and Ralf Möller. Towards analytics aware ontology based access to static and streaming data. In *ISWC*, pages 344–362, 2016.

[46] Evgeny Kharlamov, Yannis Kotidis, Theofilos Mailis, Christian Neuenstadt, Charalampos Nikolaou, Özgür Özc̈ep, Christoforos Svingos, Dmitriy Zheleznyakov, Yannis Ioannnidis, Steffen Lamparter, Ralf Möller, and Arild Waaler. An ontology-mediated analytics-aware approach to support monitoring and diagnostics of static and streaming data. *J. Web Semant.*, 2019.

[47] Evgeny Kharlamov, Theofilos Mailis, Gulnar Mehdi, Christian Neuenstadt, Özgür L. Özc̈ep, Mikhail Roshchin, Nina Solomakhina, Ahmet Soylu, Christoforos Svingos, Sebastian Brandt, Martin Giese, Yannis E. Ioannidis, Steffen Lamparter, Ralf Möller, Yannis Kotidis, and Arild Waaler. Semantic access to streaming and static data at Siemens. *J. Web Semant.*, 44:54–74, 2017.

[48] Sunghwan Kim, Jie Chen, Tiejun Cheng, Asta Gindulyte, Jia He, Siqian He, Qingliang Li, Benjamin A Shoemaker, Paul A Thiessen, Bo Yu, et al. Pubchem 2019 update: improved access to chemical data. *Nucleic acids research*, 47(D1):D1102–D1109, 2019.

[49] Yannis Kotidis and Nick Roussopoulos. A case for dynamic view management. *TODS*, 26(4):388–423, 2001.

[50] Eleni Kougioumtzi, Antonios Kontaxakis, Antonios Deligiannakis, and Yannis Kotidis. Towards creating a generalized complex event processing operator using flinkcep: architecture & benchmark. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 188–189, 2021.

[51] Kostis Kyzirakos, Manos Karpathiotakis, Konstantina Bereta, George Garbis, Charalampos Nikolaou, Panayiotis Smeros, Stella Giannakopoulou, Kallirroi Dogani, and Manolis Koubarakis. The spatiotemporal RDF store strabon. In Mario A. Nascimento, Timos K. Sellis, Reynold Cheng, Jörg Sander, Yu Zheng, Hans-Peter Kriegel, Matthias Renz, and Christian Sengstock, editors, *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings*, volume 8098 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2013.

[52] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *PODS*, pages 95–104. ACM, 1995.

[53] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *Proc. VLDB Endow.*, 12(4):390–403, 2018.

[54] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[55] LSQ 2. `https://files.dice-research.org/archive/lsqv2/dumps/`.

[56] Theofilos Mailis, Yannis Kotidis, Stamatis Christoforidis, Evgeny Kharlamov, and Yannis Ioannidis. View selection over knowledge graphs in triple stores. *Proceedings of the VLDB Endowment*, 14(13):3281–3294, 2021.

[57] Theofilos Mailis, Yannis Kotidis, Vaggelis Nikolopoulos, Evgeny Kharlamov, Ian Horrocks, and Yannis E. Ioannidis. An efficient index for RDF query containment. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1499–1516. ACM, 2019.

[58] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In *ISWC*, pages 376–394, 2018.

[59] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *Acm Sigmod Record*, 41(1):20–29, 2012.

[60] Brian McBride. Jena: Implementing the rdf model and syntax specification. In *ISWC*, pages 23–28. CEUR-WS. org, 2001.

[61] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. Extended characteristic sets: graph indexing for sparql query optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 497–508. IEEE, 2017.

[62] Kostas Messanakis, Petros Demetrakopoulos, and Yannis Kotidis. Smart-views: Decentralized OLAP view management using blockchains. In *Big Data Analytics and Knowledge Discovery*, volume 12925 of *Lecture Notes in Computer Science*, pages 216–221. Springer, 2021.

[63] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, volume 30, pages 307–318. ACM, 2001.

[64] Konstantinos Morfonios, Stratis Konakas, Yannis Ioannidis, and Nikolaos Kotsis. Rolap implementations of the data cube. *ACM Computing Surveys (CSUR)*, 39(4):12, 2007.

[65] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network? the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 493–498, 2014.

[66] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable rdf store. In *ISWC*, pages 3–20, 2015.

[67] Neo4j. `http://neo4j.com`.

[68] Thomas Neumann and Gerhard Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1-2):256–263, 2010.

[69] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[70] Antoni Olivé. *Conceptual modeling of information systems*. Springer Science & Business Media, 2007.

[71] OrientDB. `https://orientdb.org/`.

[72] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. Do we need specialized graph databases? benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES'17, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. Graph-aware, workload-adaptive sparql query caching. In *SIGMOD*, pages 1777–1792. ACM, 2015.

[74] Euripides G. M. Petrakis and A Faloutsos. Similarity searching in medical image databases. *IEEE transactions on knowledge and data engineering*, 9(3):435–447, 1997.

[75] Francois Picalausa and Stijn Vansummeren. What are real sparql queries like? In *SWIM*, page 7. ACM, 2011.

[76] Richard Qian. Understand your world with bing, May 2013. [Online; accessed 16-September-2021].

[77] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly, 2 edition, 2015.

[78] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 249–260, 2000.

[79] Timos Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[80] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[81] Amit Singhal. Introducing the knowledge graph: Things, not strings, May 2012. [Online; accessed 16-September-2021].

[82] Sparksee. `http://www.sparsity-technologies.com/`.

[83] Sqlg. `https://www.sqlg.org/docs/3.0.2/`.

[84] Claus Stadler, Muhammad Saleem, Qaiser Mehmood, Carlos Buil Aranda, Michel Dumontier, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. LSQ 2.0: A linked dataset of SPARQL query logs. *Semantic Web*, 15(1):167–189, 2024.

[85] Stardog. `https://www.stardog.com/`.

[86] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

[87] Maxim Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Operations Research Letters*, 32(1):41–43, 2004.

[88] Dimitri Theodoratos, Timos Sellis, et al. Data warehouse configuration. In *VLDB*, volume 97, pages 126–135, 1997.

[89] TigerGraph. `https://www.tigergraph.com/`.

[90] Daniel Ting. Approximate distinct counts for billions of datasets. In *Proceedings of the 2019 International Conference on Management of Data*, pages 69–86, 2019.

[91] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[92] Virtuoso open-source edition. `https://virtuoso.openlinksw.com/`.

[93] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. Millenniumdb: A persistent, open-source, graph database. *CoRR*, abs/2111.01540, 2021.

[94] Jing Wang, Nikos Ntarmos, and Peter Triantafillou. Indexing query graphs to speedup graph query processing. In *EDBT*, pages 41–52, 2016.

[95] Jing Wang, Nikos Ntarmos, and Peter Triantafillou. Graphcache: A caching system for graph queries. In *EDBT*, pages 13–24, 2017.

[96] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[97] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, pages 721–724. IEEE Computer Society, 2002.

[98] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In *ICDE*, 2013.

[99] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544, 2007.