NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

View and Index Selection on Graph Databases

Konstantinos N. Plas

Supervisor (or supervisors): **Yannis Ioannidis**, Professor at NKUA
Theofilos Mailis, Postdoctoral Researcher at NKUA

ATHENS

OCTOBER 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

View and Index Selection on Graph Databases

Κωνσταντίνος Ν. Πλας

Επιβλέπων (ή Επιβλέπουσα ή Επιβλέποντες): **Γιάννης Ιωαννίδης,** Καθηγητής ΕΚΠΑ
Θεόφιλος Μαΐλης, Μεταδιδακτορικός Ερευνητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2022

BSc THESIS

View and Index Selection on Graph Databases

Konstantinos N. Plas
S.N.: 1115201700125

SUPERVISOR: **Yannis Ioannidis**, Professor at NKUA
Theofilos Mailis, Postdoctoral Researcher at NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

View and Index Selection on Graph Databases

Κωνσταντίνος Ν. Πλας
Α.Μ.: 1115201700125

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Ιωαννίδης,** Καθηγητής ΕΚΠΑ
Θεόφιλος Μαΐλης, Μεταδιδακτορικός Ερευνητής ΕΚΠΑ

ABSTRACT

One of the most important aspects of native graph-database systems is their index-free adjacency property that enforces the nodes to have direct physical RAM addresses and physically point to other adjacent nodes. The index-free adjacency property accelerates query answering for queries that are bound to one (or more) specific nodes within the graph, namely anchor nodes. The corresponding anchor node is used as the starting point for answering the query by examining its adjacent nodes instead of the whole graph. Nevertheless, non-anchored-node queries are much harder to answer since the query planner should examine a large portion of the graph in order to answer the corresponding query. In this work we study view and index selection techniques in order to accelerate the aforementioned class of queries. We analyze different index and view selection strategies for query answering and show that, depending on the characteristics of the query, the graph database, and the corresponding answer set, a different strategy may be optimal among the indexing and view materialization alternatives. Before selecting the views and indices, our system employs pattern mining techniques in order to guess the characteristics of future queries. Thus, the initial query workload is represented by a much smaller summary of the query patterns that are most likely to appear in future queries, each pattern having a corresponding expected number of appearances. Our selection strategy is based on a greedy view & index selection strategy that at each step of its execution tries to maximize the ratio of the benefit of materializing a view/index, to the corresponding cost of storing it. Our selection algorithm is inspired by the corresponding greedy algorithm for “Maximizing a Nondecreasing Submodular Set Function Subject to a Knapsack Constraint”. Our experimental evaluation shows that all the steps of the index selection process are completed in a few seconds, while the corresponding rewritings accelerate 15.44% of the queries in the DbPedia query workload. Those queries are executed in 1.63% of their initial time on average.

SUBJECT AREA: View Materialization

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

ΠΕΡΙΛΗΨΗ

Μια από τις σημαντικότερες πτυχές των βάσεων δεδομένων γραφημάτων με εγγενή επεξεργασία γράφων είναι η ιδιότητα *γειτνίασης χωρίς ευρετήριο* (index-free-adjacency), βάση της οποίας όλοι οι κόμβοι του γράφου έχουν άμεση φυσική διεύθυνση RAM και δείκτες σε άλλους γειτονικούς κόμβους. Η ιδιότητα γειτνίασης χωρίς ευρετήριο επιταχύνει την απάντηση ερωτημάτων για ερωτήματα που συνδέονται με έναν (ή περισσότερους) συγκεκριμένους κόμβους εντός του γραφήματος, δηλαδή τους κόμβους αγκύρωσης (anchor nodes). Ο αντίστοιχος κόμβος αγκύρωσης χρησιμοποιείται ως σημείο εκκίνησης για την απάντηση στο ερώτημα εξετάζοντας τους παρακείμενους κόμβους του αντί για ολόκληρο το γράφημα. Παρόλα αυτά, τα ερωτήματα που δεν αρχίζουν από κόμβους αγκύρωσης απαντώνται πολύ πιο δύσκολα, καθώς ο σχεδιαστής ερωτημάτων(query planner) θα πρέπει να εξετάσει ένα μεγάλο μέρος του γραφήματος για να απαντήσει στο αντίστοιχο ερώτημα. Σε αυτή την εργασία μελετάμε τεχνικές επιλογής όψεων και ευρετηρίων προκειμένου να επιταχύνουμε την προαναφερθείσα κατηγορία ερωτημάτων. Αναλύουμε διαφορετικές στρατηγικές επιλογής όψεων και ευρετηρίων για την απάντηση ερωτημάτων και δείχνουμε ότι, ανάλογα με τα χαρακτηριστικά του ερωτήματος, τη βάση δεδομένων γραφημάτων και το αντίστοιχο σύνολο απαντήσεων, μια διαφορετική στρατηγική μπορεί να είναι βέλτιστη μεταξύ των εναλλακτικών λύσεων ευρετηρίασης και υλοποίησης προβολής. Πριν από την επιλογή των όψεων και των ευρετηρίων, το σύστημά μας χρησιμοποιεί τεχνικές εξόρυξης προτύπων για να μαντέψει τα χαρακτηριστικά των μελλοντικών ερωτημάτων. Έτσι, ο αρχικός φόρτος εργασίας του ερωτήματος αντιπροσωπεύεται από μια πολύ μικρότερη σύνοψη των μοτίβων ερωτημάτων που είναι πιο πιθανό να εμφανιστούν σε μελλοντικά ερωτήματα, με κάθε μοτίβο να έχει τον αντίστοιχο αναμενόμενο αριθμό εμφανίσεων. Η στρατηγική επιλογής μας βασίζεται σε μια στρατηγική επιλογής άπληστης όψεων & ευρετηρίων που σε κάθε βήμα της εκτέλεσής της προσπαθεί να μεγιστοποιήσει την αναλογία του οφέλους από την υλοποίηση μιας/ενός όψεως/ευρετηρίου, προς το αντίστοιχο κόστος αποθήκευσής τους. Ο αλγόριθμος επιλογής μας είναι εμπνευσμένος από τον αντίστοιχο άπληστο αλγόριθμο για τη «Μεγιστοποίηση μιας μη ελαττούμενης συνάρτησης υποδομοστοιχειωτού συνόλου που υπόκειται σε περιορισμό σακιδίου». Η πειραματική μας αξιολόγηση δείχνει ότι όλα τα βήματα της διαδικασίας επιλογής ευρετηρίου ολοκληρώνονται σε λίγα δευτερόλεπτα, ενώ οι αντίστοιχες επανεγγραφές επιταχύνουν το 15,44% των ερωτημάτων στον φόρτο εργασίας των ερωτημάτων της DbPedia. Αυτά τα ερωτήματα εκτελούνται στο 1,63% του αρχικού τους χρόνου κατά μέσο όρο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Υλοποίηση όψεων

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

CONTENTS

LIST OF FIGURES

LIST OF IMAGES

1. INTRODUCTION

In the past decade, and especially the last few years, a rapid increase was observed in both scientific and industrial interest for graph-database management systems. Partly responsible for this, is the changing focus of many industrial titans, namely Facebook, Twitter, Google, e.t.c.[1][2][3] from generic relational databases to graph-database systems and analytics to represent and explain the relationships and interactions of their user bases. The Facebook graph contains billions of users, corresponding to the vertices of the graph and trillion of edges representing their corresponding relationships [1][2]. Similarly, Twitter represents its thousands of million of users as nodes within its graph, and follow relationships as its directed edges [3]. In its simplest form, a social network contains individuals as vertices and edges as relationships between vertices [4]. This abstract view of human relationships, while certainly limited, has been very useful for characterizing social relationships, with structural measures of this network abstraction finding active application to the study of everything from bargaining power [5] to psychological health [6].

Alongside the aforementioned use in social networking, graph databases provide industrial solutions for various information systems. In the modeling of supply chains systems, graphs are used to prevent harm and complications along the supply line, which was highlighted with the graph database management system TigerGraph during the COVID-19 pandemic [7]. In the modeling of chemical compounds, the PubChem [8] is a dataset that contains more than half a million graphs, while ChEBI contains more than half a million graphs [9]. Further applications extend to software development and debugging [10] similarity searching in medical datasets [11], as well as recommender systems [12]. Another prominent application, where graph-databases thrive are Knowledge Graphs, i.e., collections of interconnected and annotated entities. KGs are now widely used in both academia and industry where prominent KGs such as DBpedia [13], Yago [14], Google's KG [15], and Microsoft's Satori [16] have already reached tremendous scale. Indeed, DBpedia alone currently consists of more than 1 billion triples. Hence, the demand for high performance graph-database systems that are used both in industry and academia is on the constant rise.

1.1 Graph-Database Management Systems

As a result, a large number of graph-database management systems have emerged, either general purpose systems, e.g., Neo4j [17], InfiniteGraph [18], Amazon Neptune [19], Dgraph [20], e.t.c., in-house systems designed by big data companies for their own purposes, e.g., Google's Pregel [21], Twitter's FlockDB [22], or while general purpose big-data frameworks provide extensions for graph processing, e.g. *GraphX* and *GraphFrames* for Spark [23][24].

1.2 Non-Anchored-Node Queries

One of the most important aspects of graph-database systems is their index-free adjacency property. The index-free adjacency property enforces the nodes to have direct physical RAM addresses and physically point to other adjacent nodes, resulting in a fast retrieval. Thus, query answering w.r.t. a specific node within a query, namely *anchor node*, is performed very effectively for a native graph-database system since there is no need to move through any other type of data structures to find the links to the corresponding node and its neighborhood. Nonetheless, in the case of non-anchored-node queries, the query

planner should consider every possible node or edge alternative, before taking advantage of the index-free adjacency property.

Example 1.1 For example, a query that asks for the couple(s) that have starred together in the movie “Eyes Wide Shut”:

$$\begin{aligned} &MATCH\ (n1{:}\ Person)\ -\ [{:}\ MARRIED]\ -\ (n2{:}\ Person), \\ &(n1)\ -\ [{:}\ ACTED\_IN]\ ->\ (m{:}\ Movie\ \{title{:}\ "Eyes\ Wide\ Shut"\}), \\ &(n2)\ -\ [{:}\ ACTED\_IN]\ ->\ (m) \\ &RETURN\ n1, n2 \end{aligned} \tag{1}$$

can be very effectively answered since the query planner will first visit the corresponding movie, and then explore for every actor/actress in it if its corresponding couple has also starred in the same movie.

A more general query that asks for couples who have starred in films together is much more difficult to answer:

$$\begin{aligned} &MATCH\ (n1{:}\ Person)\ -\ [{:}\ MARRIED]\ -\ (n2{:}\ Person), \\ &(n1)\ -\ [{:}\ ACTED\_IN]\ ->\ (m)\ <-\ [{:}\ ACTED\_IN]\ -\ (n2) \\ &RETURN\ n1, n2 \end{aligned} \tag{2}$$

The query planner should first consider every possible movie or every possible couple of actors and then take advantage of the free adjacency property. The corresponding query asks for triangle relations involving two actors and a movie. It may appear by itself or as part of a more complex (possibly analytical) query. It should be noted that this is a difficult query to answer that has been extensively analyzed in the bibliography, e.g., [25] investigate efficient algorithms to update this query's corresponding answer. In order to answer the corresponding query, we have to perform many searches in order to reveal the answers that satisfy it. In fact, if $|G|$ is the size of our graph database, the corresponding problem has a data complexity of $|G|^{\frac{3}{2}}$ [26][27].

If we assume that a small proportion of the nodes within the graph satisfy the corresponding query, by building an index on these nodes we would avoid much of the aforementioned overhead.

1.3 Index & View Selection in Graph Databases

A prominent approach to enhance query answering is View Materialization, ie., given a database D and a query workload Q materialize an appropriate set of computations to improve query performance. The problem of *View Selection*, the selection of the appropriate views to materialize, is achieved by finding commonalities across queries in Q the precomputation of which minimizes the execution of existing or future queries w.r.t. to a

cost function (e.g., query evaluation, storage, and subexpression maintenance costs), under a set of constraints (e.g.,space budget). Views as a concept are absent from native graph-database systems since data is not stored into tables, but modeled as vertices and edges between them and therefore tubular stored queries are typically not implemented. However, we may introduce “views” by adding edges within our graph G, while the Index Selection presented in our work uses a lot of the same principles present in *View Selection* and the materialization of the selected indices can be implemented by various tools provided by a given graph DBMS.

1.4 Contributions

In this paper, we describe a system that takes as input a graph database G, a corresponding query workload Q and builds the appropriate views and indices that will improve execution of future queries. Our system tries to “guess” the characteristics of future queries by finding the patterns that have the highest probability of appearing, and then select the appropriate indices and views for the corresponding queries. Our main contributions, regarding index and view selection for graph databases are the following:

- **Index & View Strategies:** We analyze different index and view strategies for query answering and show that, depending on the characteristics of the query, the graph database, and the corresponding answer set, a different strategy may be optimal among the indexing/view materialization alternatives.
- **Graph Summary:** We reformulate the traditional view selection multiquery optimization problem by, instead of considering the initial query workload of Q, we instead reformulate our problems in terms of the patterns that are most likely to appear within future queries. This is achieved by creating a summary $ExpQ_T$ of the initial workload, that corresponds to the most likely patterns that are expected to appear in future queries. Though our current implementation for discovering $ExpQ_T$ is based on traditional subgraph-pattern mining techniques, our work can be extended, with state-of-the-art mining algorithms [28], that take into account the temporal evolution of the query-characteristics within Q.
- **View/Index Selection & Materialization :** We employ a greedy index selection strategy, that, at each index-selection step of the algorithm tries to maximize the benefit of creating the corresponding view, compared to the cost of storing it. Additionally, we show the correspondence between the view/index selection process and the problem of “Maximizing a Nondecreasing Submodular Set Function Subject to a Knapsack Constraint” problem and discuss possible guarantees. Additionally, we suggest a *lazy materialization strategy* that incorporates all index & view related information into our corresponding graph on query-execution time. Thus an index/view is materialized only when we need to answer to a query that will be employing the corresponding index/view. This strategy allows to incorporate the materialization step into the execution step.

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

For a set S, we denote with $P(S)$ its corresponding powerset and with $P(S) \setminus \emptyset$ its corresponding powerset without the empty set.

Definition 2.1 *The Property Graph Data Model.* Assume that ***L*** is a finite set of vertex and edge labels; ***P*** is a finite set of property names; ***V*** is an infinite set of atomic values; a property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$ such that:

1) N is a finite set of vertices;
2) E is a finite set of directed edges disjoint with N;
3) $\rho : E \rightarrow N^2$ is a total function that associates each edge in E with a pair of vertices in N;
4) $\lambda : (N \cup E) \rightarrow P(S)\setminus\emptyset$ is a partial function that associates each vertex/edge with its corresponding set of labels from ***L***;
5) $\sigma : (N \cup E) \times \boldsymbol{P} \rightarrow \boldsymbol{V}$ is a partial function that associates vertices/edges with properties, and for each property it assigns a value from ***V***.

Given two vertices $n_1, n_2 \in N$ and an edge $(n_1, n_2) \in E$, we will say that n_1 is the *source vertex* and n_2 the *target* vertex of the edge. Note that our definition supports multiple labels for vertices and edges, and a single value for a property on a vertex/edge.

Definition 2.2 *Basic & Join Graph Pattern.* Given a set of variables X disjoint from the previous sets, a *basic graph pattern* (BGP) is an expression of the form :

$$(u: t_u) - [v: t_v] -> (w: t_w) \tag{3}$$

where $u, w \in X$ are variables referencing vertices; $v \in X$ is a variable referencing an edge; and t_u, t_v, t_w are labels. The expression $(u: t_u)$ is also a valid BGP which allows to obtain the vertices in the graph. Also expressions with one or more of the labels t_u, t_v, t_w missing are also acceptable. A set of BGPs is called a *join graph pattern*.

Note : for simplicity, we have ignored BGPs that also query the properties of a node or an edge.

Definition 2.3 A *join graph-pattern* query is of the form :

$$SELECT\ x_1,\ldots,x_n \tag{4}$$

$$MATCH\ J \tag{5}$$

where J is a join-graph pattern and $x_1,\ldots,x_n$ are variables appearing in J, $x_1,\ldots,x_n \in Vars(J)$.

Definition 2.4 *Query Answering.* A solution to the query in Formula 4 is a mapping $m: Vars(J) \rightarrow N \cup E$ from the variables in the join graph-pattern of the query to the vertices and edges of the graph G such that for every BGP in the form of Formula 3, appearing in J, it applies that: $m(u), m(v) \in N$, $m(v) \in E$, $\rho(m(v)) = (m(u), m(v))$, $t_u \in \lambda(m(u))$, $t_v \in \lambda(m(v))$, $t_w \in \lambda(m(w))$. The substitutions of variables appearing in the *SELECT* clause constitute the answers to the query.

2.2 Graph Databases Management System

Following the definition of a Property Graph, we will provide definitions for Graph Database Management Systems and their internal structures.

2.2.1 Graph Database Management System

A *Graph Database Management System* (GDMS) is a database management system with Create, Read, Update, and Delete methods that expose a graph data model. The Graph-Database Management System under consideration is based on Neo4j, nevertheless most of the described properties are common in most native GDMSs.

2.2.2 Native Graph Processing

A graph database management system is characterized as having *native graph processing* capabilities, regardless of its architecture and the way the graph is encoded and represented in the database engine's main memory, when it exhibits a property called *index-free adjacency*. A GDMS utilizes the aforementioned property when each vertex maintains direct references to its adjacent vertices. This results in two facts: *(i)* Each vertex 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.A nonnative graph database engine, in contrast, uses (global) indices to link

vertices together. These indices add a layer of indirection to each traversal, thereby incurring greater computational cost.

2.3 Indexing Alternatives in Neo4j

The underlying GDMS Neo4j provides 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. Neo4j allows indices on vertex/edge labels, on a single property for a given label, or on multiple properties for a given label. The indexing options available in Neo4j are *B-tree indices*, *Text Indices*, *Full-text indices*, and *Token lookup Indices*. Token lookup indices, as the name suggests, are used to look up vertices or edges with a specific label. A token lookup index is always created over all labels and hence there can only be a maximum of two token lookup indices in a database - one for vertex labels and one for edge-labels.

2.4 View Selection

2.4.1 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. In the case of native graph-databases, views can only be represented in terms of vertices and edges within the data graph. Therefore in Section 4 we study different index and view candidates for representing a query.

2.4.2 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 $\boldsymbol{\mathcal{V}} = \{V_1,\ldots,V_2\}$ if Q and Q' are equivalent and Q' contains one or more occurrences of materialized views in $\boldsymbol{\mathcal{V}}$. A *rewriting function* $Rwrt(Q,\boldsymbol{\mathcal{V}})$ takes as input the query Q and rewrites it to an equivalent query $Q' = Rwrt(Q,\boldsymbol{\mathcal{V}})$ using views from $\boldsymbol{\mathcal{V}}$. A rewriting function $Rwrt$ is optimal when there exists no other rewriting Q'' of Q such that $Cost^{\epsilon}(Q'') < Cost^{\epsilon}(Q)$, with $Cost^{\epsilon}$ being the function that maps a query to its estimated execution cost.

3. SYSTEM ARCHITECTURE

Figure 1 displays the overall architecture of our system for answering queries using views.

(i) View Selector is responsible for choosing the appropriate views for materialization. *(ii)* It employs the *Graph Pattern Miner* that discovers frequent-query patterns within a query workload, i.e., query patterns that have a high possibility of reappearing.*(iii)* The *Query Containment Engine* given a query Q and a set of materialized views I_C, finds the views that contain Q, i.e., every $V \in I_C$ such that $\pi_{\emptyset}(Q) \sqsubseteq \pi_{\emptyset}(W)$. The query containment engine employs the mv-index structure that has been presented in [29].*(iv)* The *View-Based Rewriter* engine takes the views as input and rewrites them in an optimal way regarding a cost estimation function that uses graph-database related statistics.*(v)* Finally, the *Query Optimizer* and the *Query-Evaluation Engine* are integral to every database system and are responsible for deciding the execution plan of a query and its actual execution on the underlying database.

In our system we have made the following assumption: the view based rewriter is not depended on the query optimizer. This allows our system to operate on top of an arbitrary underlying graph database. In the following we present the cost model used by the view selector and the view-based rewriter.

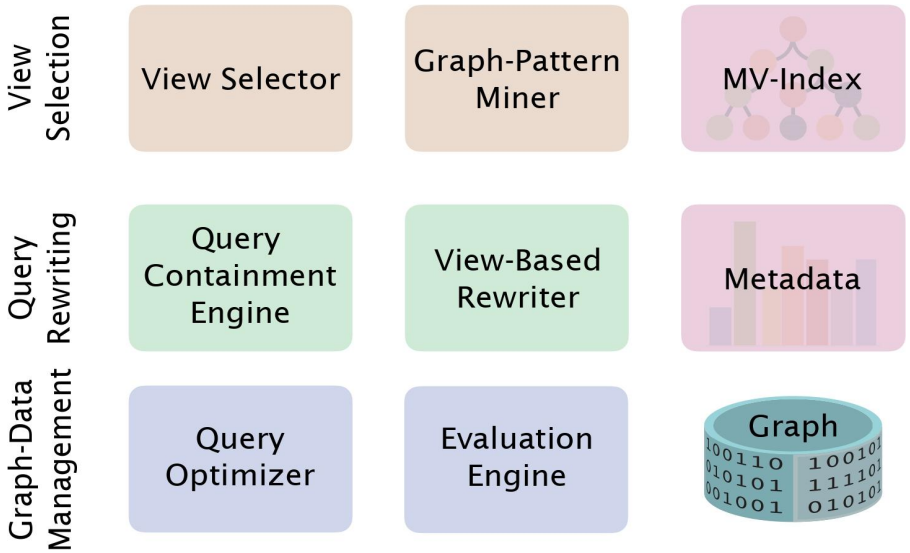


Figure 1: Frequent Graph Pattern Mining Template Databases

3.1 Cost Model

Since there is no way to find the exact cost of executing a query apart from executing it, we need to identify a function $Cost^{\epsilon}(\cdot)$ that approximates the time needed to execute a certain query. Our cost model is based on the Neo4j cost model.

3.2 Execution Plans and Neo4j Cost Model

In order to execute a query the Neo4j engine decomposes it into a number of operators. When combined, the operators form a tree-like structure called an *execution plan*. The operators represent nodes on the tree and take zero or more rows as input and produce zero or more rows in return. An example of an execution plan generated with the EXPLAIN command is shown in figure 2.

Operators that we examine in our work are the following:

- **Directed Relationship Type Scan:** defined as the *DirectedRelationshipTypeScan* operator, which fetches all relationships and their start and end nodes with a specific type from the relationship type index.
- **Produce Results:** defined as the *ProduceResults* operator, which prepares the result so that it is consumable by the user, such as transforming internal values to user values. It is present in every single query that returns data to the user, and has little bearing on performance optimization.

The evaluation of an execution plan begins at leaf nodes. Leaf nodes have no input and are generally scan or seek operators. Leaf nodes obtain rows directly from the storage engine causing database hits. Any rows produced by leaf nodes are then piped into their parent nodes, which in turn pipe their output rows to their parent nodes and so on, all the way up to the root node. The root node produces the final results of the query. Each operator maybe be annotated with the following statistic as defined in the Neo4j Cypher manual :

- **Rows:** The number of rows the operator produced. Rows are generated only when using the PROFILE Cypher call.
- **EstimatedRows:** This is the estimated number of rows that is expected to be produced by the operator. The estimate is an approximate number based on the available statistical information. The compiler uses this estimate to choose a

suitable execution plan. *EstimatedRows* is a statistic that we heavily used on the view selection process.

- **DbHits:** Each operator will send a request to the storage engine to do work such as retrieving or updating data. A database hit is an abstract unit of this storage engine work.
- **Time:** Time is only shown for some operators when using the pipelined runtime. The number shown is the time in milliseconds it took to execute the given operator.

The execution plans and the operation statistics are presented to the user when queries are executed on the database with either the PROFILE or EXPLAIN Cypher statements. The EXPLAIN option provides the execution plan for a given query without executing it. PROFILE on the other hand runs the query and keeps track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data. An example of a Neo4j query plan is depicted in Figure 2.

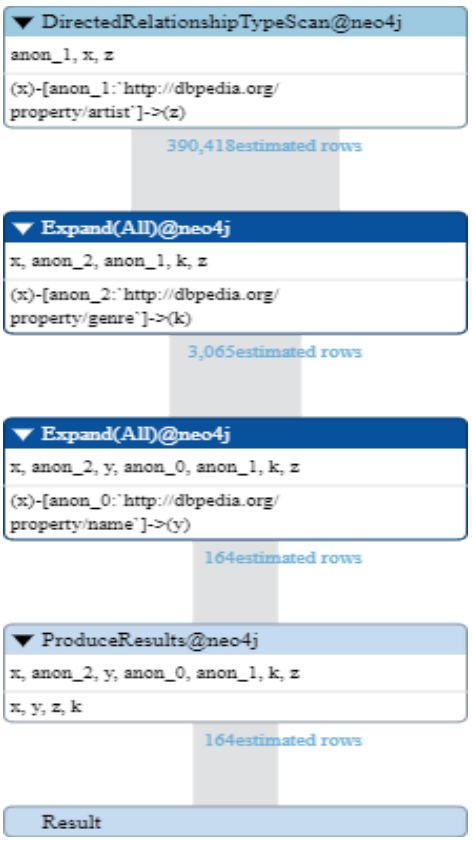


Figure 2: Query plan provided by EXPLAIN call

4. INDEX & VIEW STRATEGIES

In this section we will focus on the candidate indices and 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, and decide on the materialization alternative that is the most beneficial for our underlying system (Section 4.2).

4.1 Index & View Candidates

Given a query pattern P of a high frequency within the query workload of Q and a graph database G, we want to examine all the index and view alternatives that can be derived from the corresponding query pattern.

In retrospect (Section 2), a *database index* is a data structure that improves the speed of data retrieval operations at 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 labeling: 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 types of indices, e.g., single-property indices on nodes or edges, which do not fit the purposes of our materialization strategy.

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 by the addition of edges.

In the rest of the paper, we will use the term *index materialization* for the case of creating a new label to identify the nodes or edges within our graph G that are of specific interest. We will use the term *view/index materialization* for the case that we additionally need to create new nodes or edges in order to efficiently locate the parts of the graph G that we are interested in.

4.1.1 Index & View Alternative

We will now examine alternative indexing and view-materialization strategies w.r.t. the graph-database setting. We assume *(i)* an expected join graph-pattern P, *(ii)* its corresponding query representation namely Q_P: $SELECT\ \bar{x}\ MATCH\ P$ with $\bar{x}$ variables that also appear in P: $Vars(\bar{x}) \subseteq Vars(P)$, *(iii)* a graph database G. To recall, the join-graph pattern P corresponds to a set of BGPs $P = \{BGP_i\}_{i-1}^{k}$ with the i^{th} BGP being on the form:

$$(u_i : t_{ui}) - [v_i : t_{vi}] -> (w_j : t_{wj}) \quad (6)$$

where $u_i, v_i, w_j \in X$ are variables, while t_{ui}, t_{vi}, t_{wj} are node or edge labels.

A. A candidate strategy would be to create one or more indices on the nodes that are part of a solution to the query of Q_P. To be more specific, for the node-variable u_i appearing in P we should consider the label l_{ui} and built a corresponding index upon it. If the mapping m is a solution for the join graph-pattern of P, the node $m(u_i)$ will be labeled with l_{ui} and indexed in l_{ui}.

B. Another strategy would be to create one or more indices on the edges that are part of a solution for the join graph-pattern P. Thus, for the i^{th} BGP within the pattern P, we may create a new label l_{BGP_i} and the corresponding index on l_{BGP_i} that identifies the edges that satisfy the BGP and are also part of an answer. If the mapping m is a solution for the join graph-pattern of P, the edge $m(v_i)$ will be labeled with l_{BGP_i} and indexed in l_{BGP_i}.

We will now examine view materialization strategies, i.e., strategies that create additional nodes or edges to the existing graph database G.

C. A materialized view that introduces a new node for each solution/answer to the query of Q_P. The fresh node is connected to the constant nodes of the solution/answer it represents via fresh edges. If the mapping m is a solution for the join graph-pattern of P, we create a fresh node n_m within the graph, corresponding

to the solution of m. For the i^{th} node variable u_i appearing in P we could create a new edge e_i that connects n_m with $m(u_i)$: $\rho(e_i) = (n_m, m(u_i))$. The specific edge will be labeled with the label of ls_i and indexed accordingly. The corresponding strategy is called reification in the Semantic-Web [30].

D. A materialized view that introduces new edges and labels between pairs of answer constants.

We provide an example that illustrates the different materialization alternatives.

Example 4.1 Let J being the join pattern in Formula 7a and Q_P the pattern query in Formula 7b:

$$(x) - [e_1 : r_1] -> (y), (y) - [e_2 : r_2] -> (z), (y) - [e_3 : r_3] -> (w) \quad (7a)$$

$$SELECT\ x, y, z, w\ Match\ J \quad (7b)$$

The corresponding query is depicted in Figure 3a and is applied to the graph database in Figure 3b. Based on the aforementioned indexing and view-materialization strategies we have the following alternative structures:

A paradigm of Strategy A in Figure 3c will create two node indices, represented via light green and light blue colors, corresponding to the labels l_{green} and l_{bue}, to identify the nodes of G that satisfy the following two queries:

$$Create\ (x : l_{green})\ Match\ J$$
$$Create\ (y : l_{blue})\ Match\ J \quad (8)$$

A paradigm of Strategy B in Figure 3d will create three edge indices, represented via blue, green, red colors, the labeled edges satisfy the corresponding query results:

$$Create\ (e_1 : l_{blue})\ Match\ J$$

$$Create\ (e_2 : l_{green})\ Match\ J$$

$$Create\ (e_3 : l_{red})\ Match\ J \tag{9}$$

A paradigm of Strategy C in Figure 3e will represent each answer of the query-pattern by adding an additional node and the corresponding edges between the new node and the parts of the graph database that constitute the answer. Let's assume that f is a function that maps each set of constants to a fresh node, the corresponding view can be described via the following structures:

$$Create\ (f(x, y, z, w))\ Match\ J \tag{10a}$$

$$Create\ (f(x, y, z, w))\ -\ [:\varepsilon_1]\ ->\ (x)\ Match\ J \tag{10b}$$

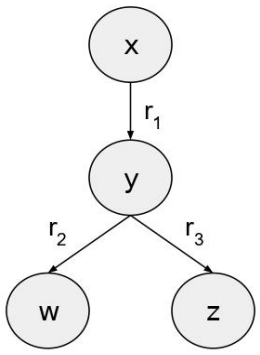
$$Create\ (f(x, y, z, w))\ -\ [:\varepsilon_2]\ ->\ (y)\ Match\ J \tag{10c}$$

. . .

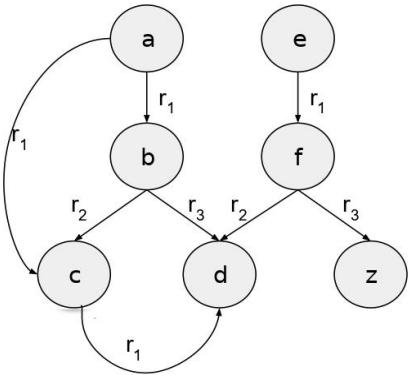
Since $f(a, b, c, d) = a_1$ node a_1 will be connected to the constants a, b, c, d via the labeled edges $\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4$ indicating that the tuple (a, b, c, d) satisfies the pattern in Formula 7b. The same applies for the node a_2 that indicates that the tuple (e, f, d, z) also satisfies the query.

Figure 3f illustrates a paradigm of Strategy D where an edge between the nodes illustrates that they are part of the same answer. The view in Figure 3fcorresponds to the following query:

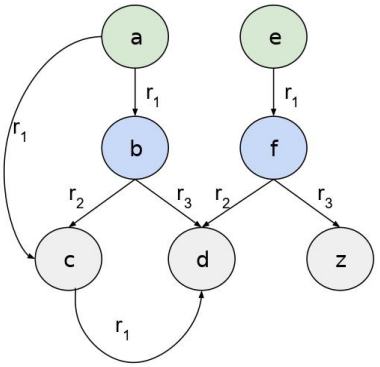
$$Create\ (x)\ -\ [:r_{x,z}]\ ->\ (z)\ Match\ J \tag{11}$$



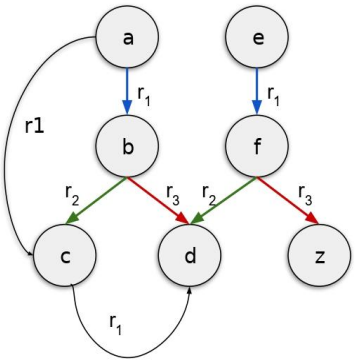
3a: Frequent Query Pattern



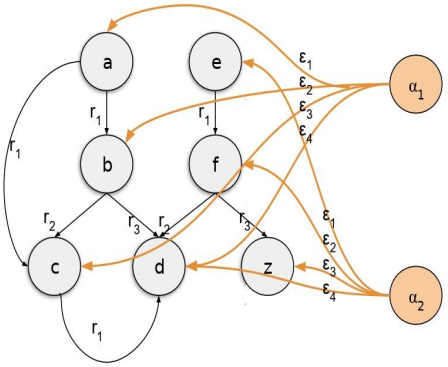
3b: Graph Database G



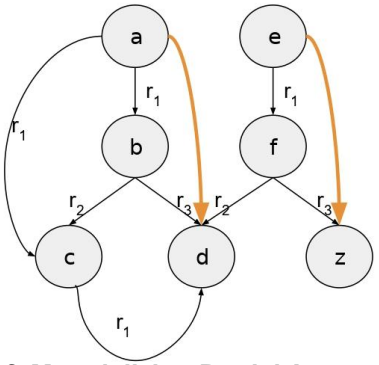
3c: Index on Node labels



3d: Index on Node labels



3e: Materializing Answer Set



3f: Materializing Partial Answer

Figure 3: A frequent pattern on a graph database G and its corresponding index and view-materialization alternatives

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 query based on a view or index, there needs to exist a containment mapping from the view to the conjunctive query. For multiset semantics, a sufficient condition is to have a subgraph isomorphism from the view or index to the corresponding query.

Example 4.2 We illustrate the available rewritings based on our view and index alternatives, assuming we have a query that is an extension of the pattern appearing in 7b, e.g.:

$$SELECT\ x, y, z, w\ Match\ J, (x) - [e_4 : s] -> (z) \quad (12)$$

The latter query is rewritten as follows based on the various indices and views in Formulas 8,9,10,11:

$$Select\ x, y, z, w\ Match$$

$$(x : l_{green}) - [e_1 : r_1] -> (y : l_{blue}), (y) - [e_2 : r_2] -> (z),$$

$$(y) - [e_3 : r_3] -> (w), (x) - [e_4 : s] -> (z) \quad (13a)$$

$$Select\ x, y, z, w\ Match$$

$$(x) - [e_1 : l_{green}] -> (y : l_{blue}), (y) - [e_2 : l_{blue}] -> (z),$$

$$(y) - [e_3 : l_{red}] -> (w), (x) - [e_4 : s] -> (z) \quad (13b)$$

$$Select\ x, y, z, w\ Match$$

$$(\chi) - [e_1 : \epsilon_1] -> (x), (\chi) - [e_2 : \epsilon_2] -> (y),$$

$$(\chi) - [e_3 : \epsilon_3] -> (z), (\chi) - [e_3 : \epsilon_4] -> (w),$$

$$(x) - [e_4 : s] -> (z) \quad (13c)$$

$$Select\ x, y, z, w\ Match\ J, (x) - [e_4 : s] -> (z),$$

$$(x) - [e_5 : r_{x,z}] -> (z) \quad (13d)$$

It should be noted, that depending on the characteristics of the query pattern and the graph database, a different type of views would be the most beneficial. For query patterns that $SELECT$ a single node, Strategy A would be the most beneficial. For tree-query patterns, Strategy B outperforms all the other strategies since it allows to label the edges that are part of the answer with the minimum overhead compared to Strategy C. The reification Strategy C becomes worst-case optimal when we need to build views for cyclic queries. A smarter strategy that we have not examined would be to represent cycles within the pattern using Strategy C and the rest of the pattern using Strategy B.

5. CANDIDATES FOR INDEXING

In Section 4 we studied indexing and view-materialization alternatives. This Section focuses on given a query workload Q, how to choose the appropriate query patterns upon which the indexing structure will be built on. The process for selecting the appropriate views is resolved into the following tasks: *(i)* forecasting the characteristics of future queries based on the up-till-now query workload Q; *(ii)* defining the appropriate set of candidate indexes for materialization I_C based on our forecasting; *(iii)* selecting the indices $I \subseteq I_C$ that will be materialized. Of primary importance through the selection process is the patterns under consideration to have a high probability of appearing in future queries.

For a query pattern Q_P and a time period of T, e.g., within the next 24 hours, we denote with $X_{Q_P,T}$ the number of appearances of the query pattern Q_P during T and $E(X_{Q_P}, T)$ its corresponding expected value.

Example 5.1 Suppose that we have the patterns Q_{P1} and Q_{P2} appearing in our query workload Q and we want to create the appropriate views for the next 24 hours. Now, let's assume that building an index on Q_{P1} will have a benefit of 10 (reads), while building an index on Q_{P2} will have a benefit of 8 (reads). If the two patterns have the same expected number of appearances within T="24 next hours", the choice of which pattern to index is straightforward. On the contrary, if we assume that, given our query workload, the expected number of appearance of pattern Q_{P1} within T is much lesser than of that of pattern Q_{P2}, e.g., $E(X_{Q_{P1}}, T) = 1$ while $E(X_{Q_{P2}}, T) = 100$, we would go for the second choice. Thus, we assume that our system can safely ignore patterns with a low expectancy of appearing.

This section is structured as follows: Paragraph 5.1 describes our methodology for finding the patterns of interest within future queries. Paragraph 5.2 describes how to construct a summarization of Q based on the frequent subgraph patterns.

5.1 Finding Query Patterns of a High Expected Number of Appearances

Given a query workload, our goal is to find the patterns with the highest expected number of appearances in future queries within a certain period. Intuitively, a query pattern Q_P that is expected to appear 1000 times within the next time period, can be used for the rewriting of at most 1000 queries appearing during the corresponding period. To reduce our search space in order to achieve our goals, we consider only the patterns whose expected number of appearances is above a certain threshold.

Problem 1 Given a time period of T, a query workload of Q and a threshold n of appearances, we want to find all the patterns Q_p such that $E(X_{Q_P}, T) \geq n$.

Example 5.2 We assume that we need to materialize the views for the next 24 hours, i.e., $T = [now, now + 24\,hours]$ and we are only interested in patterns that are expected to appear at least 2 times, i.e., $E(X_{Q_P}, T) \geq 2$.

5.1.1 Frequent Subgraph Mining Approach

To solve the problem, we have considered a straightforward approach that takes advantage of existing frequent subgraph mining algorithms, such that of GSpan [31]. Our approach mines all the patterns within the query workload of Q that have a number of appearance above a certain threshold n within the query workload. Then it is assumed that if a pattern appeared in Q more than n times and the queries in Q span a period with a total duration of $Duration(Q)$, then the expected number of appearances of Q_P during T is:

$$E(X_{Q_P}, T) = \frac{n \cdot Duration(T)}{Duration(Q)} \quad (14)$$

Example 5.3 Suppose that the pattern Q_P appears 1000 in Q and Q represents the queries of a year. Then, the expected number of appearances for the next day is: $\frac{1000}{365}$.

5.1.2 Graph Representation of Queries

To find frequent patterns within the query workload Q we employ the GSpan algorithm for pattern-mining which is described in [31], while using the implementation available in [32][33]. For ease of presentation, we assume that the pattern mining algorithm can also handle directed graphs. It should be noted that a prepossessing step allows us to discover additional information by appropriately representing our queries as GSpan compatible graphs. In the corresponding prepossessing step we will replace a constant a with a fresh variable $?x$ while also adding the edges $(?x, is, a)$ and $(?x, isA, constant)$. The latter rewriting allows to mine patterns that are obscured by the corresponding constant values, while at the same time keeping the information that a specific variable node was derived from a constant. Our implementation is based on the existing work for the problem of *frequent subgraph mining* assuming that a subgraph pattern that has a high frequency within Q also has a high probability of appearing in future queries. Our work can be generalized to more sophisticated methods that take into account information such as the temporal evolution of the query workload Q for forecasting graph [28].

5.2 Frequent Patterns as Query Workload Summaries

Frequent query patterns play a dual role in the view selection process: they provide the appropriate candidates for creating indexes but also allow our algorithm to represent in compact form a query workload Q via a smaller multiset of *expected query pattern* Q_P. Thus, instead of the initial workload Q we have an expected workload $ExpQ_T$ that contains the patterns with a high probability of appearing in $ExpQ_T$ within the time period of T, along with a corresponding multiplicity that depicts its expected number of appearances. For example, if for some time period T we have that the query pattern Q_P appears in $ExpQ_T$ with a multiplicity of 11.5, i.e., $ExpQ_T(Q_P) = 11.5$, then the pattern Q_P is expected to appear 11.5 times in the time period of T.

5.2.1 Adjusting Multiplicities

It should be noted that for the query patterns Q_P and Q_P' such that there exists a subgraph isomorphism from Q_P to Q_P', there is a need to adjust their corresponding multiplicities in

our summary because Q_P already appears in Q_P'. Thus we need to adjust the multiplicity of Q_P as follows $ExpQ_T(Q_P) := ExpQ_T(Q_P) - Q_P'$. It should be noted that if $ExpQ_T(Q_P) = ExpQ_T(Q_P')$, we can also remove Q_P from our summary.

6. INDEX SELECTION

In this section we focus on efficient and tractable algorithms for selecting the views that will be materialized. Having defined the query rewriting process in Section 4.2, we proceed to designate the index-selection methodology (Section 6.1).

6.1 The Index Selection Algorithm

The *degree of benefit* of a rewriting function to a query Q (a set of queries $\mathcal{Q}$) w.r.t. to a set of indexes I, defined in Section 2.4, is

$$Bnft(Q, I) = Cost^{\epsilon}(Q) - Cost^{\epsilon}(Rwrt(Q, I))$$

$$Bnft(\mathcal{Q}, I) = \sum_{Q \in \mathcal{Q}} Bnft(Q, I)$$

A set of indexes I is beneficial for a query $Q \in \mathcal{Q}$ when there exists a query execution plan $Q' = Rwrt(Q, I)$ such that $Cost^{\epsilon}(Q') < Cost^{\epsilon}(Q)$, or equivalently, when $Bnft(Q, I) > 0$. The objective of the *index selection process* is to identify the indexes in I that are the most beneficial to materialize w.r.t. the query workload $\mathcal{Q}$, i.e., that maximize $Bnft(\mathcal{Q}, I)$. In our implementation, when considering the benefit of an index I (set of indexes I) to a query workload $\mathcal{Q}$, we employ the summarization $ExpQ_T$ of the query workload based on the patterns expected to appear in future queries (Section 5.2). It should be noted that each expected query pattern is stored along its corresponding query plan and estimated overall cost (the overall cost also considers the multiplicity of the specific pattern). Each time a new index is added, the corresponding plan and estimated cost also get updated.

Problem 2 *(Index Selection Problem).* Given a set of candidate views I_C, an expected query workload $ExpQ_T$, and a storage capacity of b: which subset $I \subseteq I_C$ to materialize such that the size of I is less than b and the expected query workload of $ExpQ_T$ w.r.t. I is benefited the most.

6.1.1 Index Selection Algorithm

Our index selection algorithm is presented in Algorithm 1 shown in picture 1 for the index selection problem with I_C being the candidates indices for materialization, I the selected indices, and b the available storage. I) The first step of the algorithm is to remove the candidate indices that do not fit in the available storage space (line2). II) If the set of remaining indices is empty, then the set I of the already selected indices is returned (line 3). III) Else, the algorithm finds the index $I_t \in I_C$ whose addition to I produces the maximum benefit to storage cost ratio (lines 6,8). IV) The corresponding index is removed from the set I_C of candidates and added to the set I of indices that are selected for materialization (lines 10). V) The algorithm is then executed for the updated I_C and I (line 10). For our study, we assume that the cost estimation function $Cost^{\epsilon}$ has a O(1) complexity, thus the estimated cost of a query can be computed immediately.

6.1.2 Optimizations

Finding a locally optimal view $V_t \in \mathcal{V}$ is the most demanding step of Algorithm 1 (lines 6,8). To prune the search space, we keep a memoization table for the benefit to storage cost ratio. For updating the memoization table, we consider a variation of the technique presented in [34]. The updated benefit ratio is computed only for the view that has the optimal benefit ratio based on previous computations. If the updated ratio agrees with the previously computed ratio, or is greater than all benefit ratios of other views, the corresponding view will be selected. Otherwise, the view with the next highest benefit ratio is examined

Algorithm 1 View Selection

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

Image 1 : View Selection Algorithm

6.2 Reduction to MNssfKc problem

We will now present the problem of *Maximizing a Nondecreasing Submodular Set Function Subject to a Knapsack Constraint* (MNssfKc) and will showcase the reduction from the index-selection problem to the MNssfKc problem. The corresponding reduction offers the alternative to employ existing solvers for the MNssfKc problem in order to solve our index-selection problem. Furthermore, if the selected $Cost^{\epsilon}$-estimation function and the query planner induce submodular $Bnft$ function, the algorithm we presented in Algorithm 1 becomes a $(1 - e^{-1})$-approximation algorithm for solving it. It should be noted that similar reductions have been proposed in the existing bibliography [35][36].

We will first present the MNssfKc problem introduced in [37] and the corresponding reduction from the view-selection to the \knapsack problem, i.e., we will identify the parameters of the MNssfKc problem to solve the index-selection problem.

Problem 3 *(MNssfKc [37])* Let $I = \{1,\dots,n\}$; $i \in I$ and b be nonnegative integers; and $f(\cdot)$ be a nonnegative, nondecreasing, submodular, polynomially computable set function. Consider the following optimization problem in picture 2:

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

Image 2: MNssfKc problem

6.2.1 Reduction

We now identify the parameters of the reduction from Problem 2 to Problem 3 : I) For the set $I_C \leftarrow \{V_1,\dots,V_n\}$ such that $|I_C| = n$, we define an arbitrary bijection $Bi: I_C \leftrightarrow [[1,n]]$ and set $I := [[1,n]]$. II) For each $i \in [[1,n]]$ we define $c_i = Cost^{\epsilon}(Bi^{-1}(i))$ to be the corresponding index's cost, depicting the cost of materializing the specific index. III) Each subset $S \subseteq I$ is mapped via the Bi^{-1} function to a subset of the candidate indices $\mathcal{V} \subseteq I_C$ (by mapping each $i \in S$ to $Bi^{-1}(i) \in \mathcal{V}$). For the specific $\mathcal{V}$, we define $f(S)$ to be the benefit of the materialized indices in I to the query workload in $ExpQ_T$, i.e. $Bnft(ExpQ_T, \mathcal{V})$. Finally, b is the available storage space for materialization.

It is straightforward to show that by solving the formula in picture 2 we acquire an optimal solution for the view selection problem. Also, based on the properties of the MNssfKc

problem there exist a $(1 - e^{-1})$-approximation algorithm for maximizing formula in picture 2 as long as $Bnft(\mathcal{Q},I)$ and consequently $f(S)$ are nonnegative, nondecreasing, polynomially computable, submodular functions [37]. The latter algorithm corresponds to Algorithm 1.

In order for the $(1 - e^{-1})$-approximation to apply, there is the need for the $Bnft$ function to be nonnegative, nondecreasing, polynomially computable, 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 $\mathcal{V}'$; III) while the benefit functions for the rewriting Algorithms are linear as illustrated in Section 4.2. Nevertheless, the submodularity property, which depends on the cost-estimation function and the query planner, won't always apply, therefore we cannot provide any guarantees regarding the optimality of our algorithm.

6.3 Lazy Index & View Selection

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 $Q_k = \{q_{k1}, q_{k2},..., q_{kn}\}$ that may be run 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 ($DbHits$ in Neo4j) 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 $q_{k2} \exists Q_k$ 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.

7. EXPERIMENTAL EVALUATION

The aim of our evaluation section is to examine the performance of the index selection methodology as well as the quality of the indexes that get selected throughout the index-selection process. For our testing scenarios, our application takes as input a knowledge graph G; a query workload $\mathcal{Q}$, corresponding to past queries; a query workload Q_T corresponding to future queries; and produces the indexes I that will be materialized for future query execution. The quality of the views in $I$is later tested for rewritings w.r.t. to the query workload Q_T . The basic scenario we considered regarding 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.

7.1 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 data are stored in a Neo4j Community v4.4.7 database running on the same computer.

7.2 Implementation Setup

We have implemented our algorithm in Java 17 using the Apache Jena 4.0.0 open source Semantic Web framework [38] 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 [29].

7.3 Benchmark

For benchmarking our methodology, we employed the DBpedia semantic knowledge graph [39][40] that has 189,511,679 triples and its corresponding size on disk is 133.93 GB. The corresponding real-world query workload [41], originating from queries on the DBpedia knowledge graph, contains 1,287,711 queries. We have randomly partitioned the query workload into the DBpedia training query workload $\mathcal{Q}$, containing 1,277,711 queries that will be used for selecting the appropriate views for materialization and the DBpedia testing query workload $\mathcal{Q}_T$, containing 62830 (5% of the initial workload size) queries that will be used for testing the efficiency of the selected materialized views. We proceed with each step of the view-selection process.

7.4 Scalability of the Index Selection process

To showcase our algorithm's capability to efficiently select the appropriate indexes to be materialized, we study its execution time and estimated results on varying query workload sizes. Figure 4 illustrates the various stages of our algorithm for training w.r.t. query-workload samples ranging from 183,958 to 1,287,706 queries, while the available storage for materialization was 25000 records. The process was broken into four parts. The first preprocessing step converts queries into Gspan-compatible graphs. The Mining step focuses on finding frequent patterns in our given workload. The cleaning step transforms the mined patterns and filters out non beneficial patterns and finally the selection step executes the view selection algorithm. It should be noted that the materialization process is not involved in this examination, since it happens lazily as was discussed. For the different training query workload samples, the mining algorithm searches for patterns that appear in 0.07% of the queries in the workload. We observe that our algorithm behaves well for augmenting workload sizes, this can be attributed to the pattern-mining step that effectively represents each workload by a corresponding summarization. Therefore, the index selection process depends on the size of the summarization and not on the actual size of the query-workload.

Figure 4 : The scalability of the index selection process for various query-workload size

7.5 Evaluation Parameters

For the test queries $\mathcal{Q}_T$ we used a total of 62830 queries. For the various examinations we considered a range of maximum storage capacity of 100, 1000, 5000, 10000, 25000, 50000. The minSup we chose was 1000, while the threshold for the benefit was 256, meaning we only examined patterns that provided a benefit of more than 256.

7.6 Effectiveness of Selected Views

We now examine the quality of the selected views by rewriting the queries within the testing-query workload containing 62830 queries. We will consider the following parameterization for our problem: We should point out that for the index selection methodology, we employ the linear cost model assumption and not a more complicated cost estimation function. Figure 5 illustrates the percentage of the queries that are benefited from the index selection process. The x-axis represents the available storage for materialization–measured in terms of records used for materialization–, while the y-axis the percentage of benefited queries. Figure 5 illustrates the overall performance for the queries in the testing workload $\mathcal{Q}_T$; and varying capacities for materialization. The x-axis in

Figure 6 illustrates the available storage for materialization , while the y-axis illustrates the overall performance for the testing workload measured in *DbHits* .

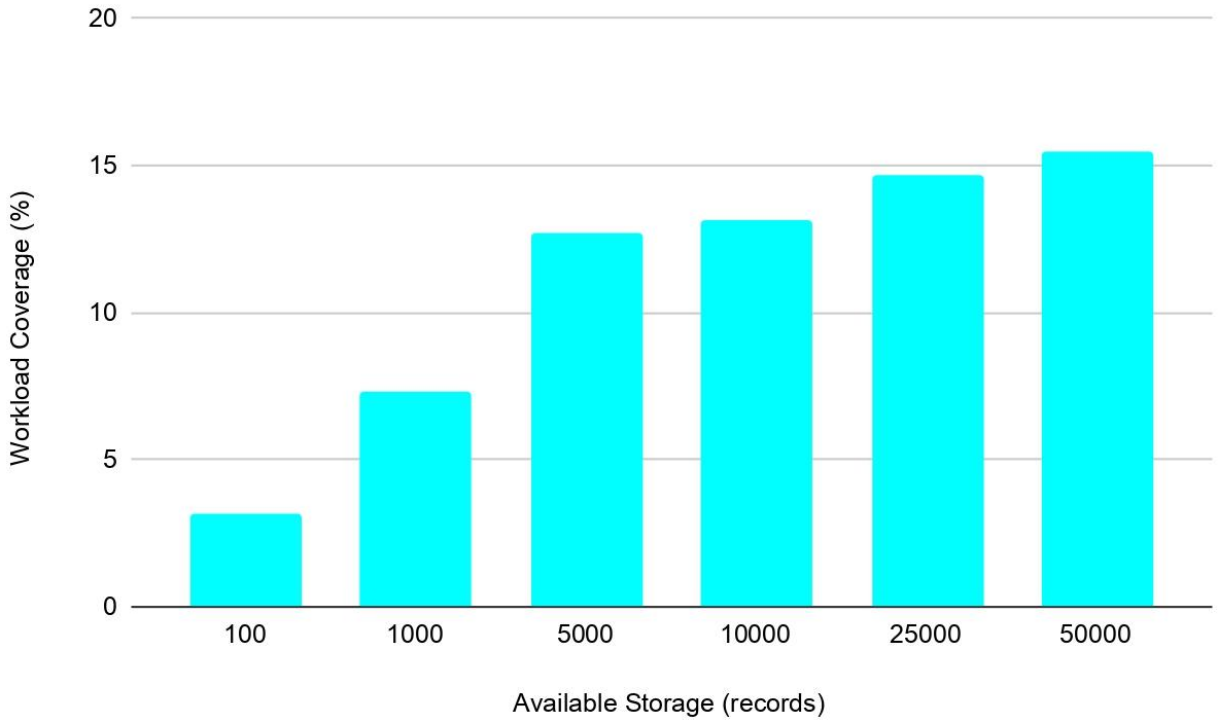


Figure 5: Benefited Queries

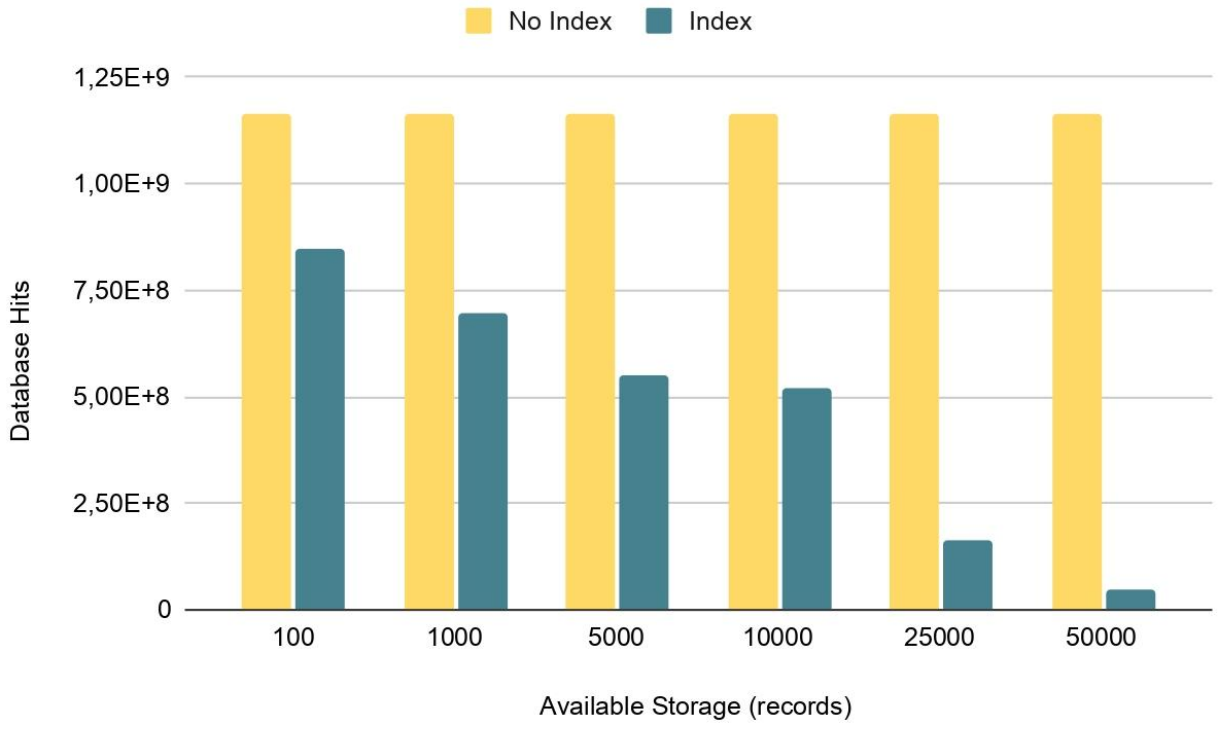


Figure 6: Index Efficiency

7.7 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. 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 7, 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 Section 6.3. 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 7: Lazy materialization method against simple index creation

8. 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.Sellis [42] studies the problem of multiple-query optimization providing its systematic analysis and considering global access plans that access subqueries.Mistry et al. [34], Roy et al [43] 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. Agarawal et al. [44] describe a system for view and index selection that incorporates several heuristics for pruning the space of possible view configurations. Zhou et al. [35] present an efficient solution for the problem of common subexpression identification by introducing a light-weight mechanism, called table signatures, for identifying sharable subexpressions. Chirkova et al. [45] formalize the view selection problem and provide a lower Exp and an upper 3Exp bound for it. Kathuria and Sudarshan [46] devised an approximation algorithm that runs in time quadratic to the number of common subexpressions and provides theoretical guarantees on the quality of the solution obtained. Jindal et al. [47] 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 [48]. A different methodology for solving the multiple-query and the view selection problem has been presented by Bayir et al [49], Chaves et al [50] that employ evolutionary techniques such as genetic algorithms. An overall analysis of the view selection problem has been presented by Mami and Bellahsene [51]. Our approach differs from previous view-materialization approaches since it allows plugging 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 it to employ pattern-mining and forecasting techniques.

- **Semantic Web & Graph Data Systems:** Much research effort has been invested in the development of scalable centralized or distributed triple stores, techniques for indexing KGs and for processing queries. Among the centralized approaches, native triple stores like Jena [52], Sesame [53], HexaStore [54], SW-Store [55], MonetDB-RDF[56], RDF-3X [57], and BitMat [58] have been carefully designed to keep up pace with the growing scale of RDF collections. Systems like TriAD [59], RDFox [60], H-RDF-3X [61], EAGRE [62] 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 [63], an approach for the materialization of shortcuts that reduces the execution cost of path queries is suggested. In [64], a different materialization strategy where an initial query workload $\mathscr{W}$ is transformed to a set of simpler views $\mathscr{V}$ along with a set of rewritings is presented. In [65], 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 [66][67]. The caching algorithms in [65] and [66][67] are based on finding subgraph-isomorphisms between incoming and cached queries. Finally, [68] 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 [69] 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 [70], along with the evolution of queries over time. Finally, a study of the Wikidata knowledge graph is presented in [71].
- **Data Warehouses:** View-selection techniques have been studied for data warehouses and problems of online analytical processing. Several early techniques were proposed including AND/OR graphs [72], modeling the problem as a state optimization [73], and lattices to represent data cube operations [74][75][76], while the problem of view management has been also studied for decentralized OLAP applications using blockchains [77]. 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.

9. CONCLUSIONS AND FUTURE WORK

In our work we studied the problem of indexing/view selection & materialization for non-anchored-node queries on graph databases. Our system is built on top of the \neo graph-database management system, but it can straightforwardly be adjusted to work on top of other databases as well. In the core of our view selection strategy is the query-workload summarization algorithm that, based on subpattern graph-mining, allows us to represent the initial query workload via the query patterns that have a high probability of appearing in the future. Our implementation considers different materialization strategies and decides upon which to employ, depending on the characteristics of the corresponding query pattern. Finally, we propose a selection strategy that greedily selects at each execution stage the view/index of the highest benefit to storage-cost ration. The latter is inspired by a variation of the knapsack problem and the corresponding algorithm for solving it. 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 experimental evaluation shows that all the steps of the index selection process are completed in a few seconds, while the corresponding rewritings accelerate 15.44% of the queries in the DbPedia query workload. Those queries are executed in 1.63% of their initial time on average.

In our future work we intend to study view-selection techniques for streaming graphs [78], focusing on stream processing for Semantic Web applications [79][80][81]; as well as complex event processing [82]. Additionally, we intend to integrate approximate counters [83] 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 [84]. 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
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

REFERENCES

[1] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, “The anatomy of the facebook social graph,” arXiv preprint arXiv:1111.4503, 2011.

[2] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” Proceedings of the VLDB Endowment, vol. 8, no. 12, pp. 1804–1815, 2015.

[3] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, “Information network or social network? the structure of the twitter follow graph,” in Proceedings of the 23rd International Conference on World Wide Web, pp. 493–498, 2014.

[4] A. Pacaci, A. Zhou, J. Lin, and M. T. 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), Association for Computing Machinery, 2017.

[5] T. Chakraborty, S. Judd, M. Kearns, and J. Tan, “A behavioral study of bargaining in social networks,” in Proceedings of the 11th ACM Conference on Electronic Commerce, (New York, NY, USA), p. 243–252, Association for Computing Machinery, 2010.

[6] N. P. Lorant V, Nazroo J, “Optimal network for patients with severe mental illness: A social network analysis,” 2017.

[7] B. Virginia, “Why tigergraph was a differentiator for jaguar land rover during the pandemic,” 2021.

[8] S. Kim, J. Chen, T. Cheng, A. Gindulyte, J. He, S. He, Q. Li, B. A. Shoemaker, P. A. Thiessen, B. Yu, et al., “Pubchem 2019 update: improved access to chemical data,” Nucleic acids research, vol. 47, no. D1, pp. D1102–D1109, 2019.

[9] K. Degtyarenko, J. Hastings, P. de Matos, and M. Ennis, “Chebi: an open bioinformatics and cheminformatics resource,” Current protocols in bioinformatics, vol. 26, no. 1, pp. 14–9, 2009.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “Sober: statistical model-based bug localization,” ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, pp. 286–295, 2005.

[11] E. G. M. Petrakis and A. Faloutsos, “Similarity searching in medical image databases,” IEEE transactions on knowledge and data engineering, vol. 9, no. 3,pp. 435–447, 1997.

[12] Z. Huang, W. Chung, T.-H. Ong, and H. Chen, “A graph-based recommender system for digital library,” in Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries, pp. 65–73, 2002.

[13] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger, “Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago,” Semantic Web, vol. 9, no. 1, pp. 77–129, 2018.

[14] F. M. Suchanek, G. Kasneci, and G. Weikum, “Yago: a core of semantic knowledge,” in Proceedings of the 16th international conference on World Wide Web, pp. 697–706, 2007.

[15] A. Singhal, “Introducing the knowledge graph: Things, not strings,” May 2012. [Online; accessed 16-September-2021].

[16] R. Qian, “Understand your world with bing,” May 2013. [Online; accessed 16-September-2021].

[17] J. J. Miller, “Graph database applications and concepts with neo4j,” in Proceedings of the southern association for information systems conference, Atlanta, GA, USA, vol. 2324, 2013.

[18] R. F. van der Lans, “Infinitegraph: Extending business, social and government intelligence with graph analytics,” tech. rep., Technical report, 20, 2010.

[19] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, et al., “Amazon neptune: Graph data management in the cloud.,” in ISWC (P&D/Industry/BlueSky), 2018.

[20] M. Jain, “Dgraph: synchronously replicated, transactional and distributed graph database,” birth, 2005.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146, 2010.

[22] R. Pointer, N. Kallen, E. Ceaser, and J. Kalucki, “Introducing flockdb,” Twitter, Inc. May, 2010.

[23] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “Graphx: A resilient distributed graph system on spark,” in First international workshop on graph data management experiences and systems, pp. 1–6, 2013.

[24] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, “Graphframes: an integrated api for mixing graph and relational queries,” in Proceedings of the fourth international workshop on graph data management experiences and systems, pp. 1–8, 2016

[25] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang, “Maintaining triangle queries under updates,” ACM Transactions on Database Systems (TODS), vol. 45, no. 3, pp. 1–46, 2020.

[26] N. Alon, R. Yuster, and U. Zwick, “Finding and counting given length cycles,” Algorithmica, vol. 17, no. 3, pp. 209–223, 1997.

[27] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, “Worst-case optimal join algorithms,” Journal of the ACM (JACM), vol. 65, no. 3, pp. 1–40, 2018.

[28] P. Fournier-Viger, G. He, C. Cheng, J. Li, M. Zhou, J. C.-W. Lin, and U. Yun, “A survey of pattern mining in dynamic graphs,” Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 10, no. 6, p. e1372, 2020.
[29] T. Mailis, Y. Kotidis, V. Nikolopoulos, E. Kharlamov, I. Horrocks, and Y. E. Ioannidis, “An efficient index for RDF query containment,” in Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019 (P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, eds.), pp. 1499–1516, ACM, 2019.
[30] A. Olivé, Conceptual modeling of information systems. Springer Science & Business Media, 2007.
[31] X. Yan and J. 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, pp. 721–724, IEEE Computer Society, 2002.
[32] P.Fournier-Viger, C. Cheng, J. C.-W. Lin, U. Yun, and R. U. Kiran, “Tkg: Efficient mining of top-k frequent subgraphs,” in International Conference on Big Data Analytics, pp. 209–226, Springer, 2019.
[33] P. Fournier-Viger and C. Cheng, “Hue-span,” May 2019. [Online; accessed 16-September-2021].
[34] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, “Efficient and extensible algorithms for multi query optimization,” in Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 249–260, 2000.
[35] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, “Efficient exploitation of similar subexpressions for query processing,” in Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 533–544, 2007.
[36] T. Mailis, Y. Kotidis, S. Christoforidis, E. Kharlamov, and Y. Ioannidis, “View selection over knowledge graphs in triple stores,” Proceedings of the VLDB Endowment,vol. 14, no. 13, pp. 3281–3294, 2021.
[37] M. Sviridenko, “A note on maximizing a submodular set function subject to a knapsack constraint,” Operations Research Letters, vol. 32, no. 1, pp. 41–43, 2004.
[38] A. Jena, “semantic web framework for java,” 2007.
[39] “Dbpedia 3.9,” 2019. [Online; accessed 16-September-2021].
[40] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “Dbpedia: A nucleus for a web of open data,” ISWC/ASWC, pp. 722–735, 2007.
[41] “Dbpedia log,” 2012. [Online; accessed 16-September-2021].
[42] T. K. Sellis, “Multiple-query optimization,” ACM Transactions on Database Systems (TODS), vol. 13, no. 1, pp. 23–52, 1988.
[43] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, “Materialized view selection and maintenance using multi-query optimization,” in ACM SIGMOD Record, vol. 30, pp. 307–318, ACM, 2001.
[44] S. Agarawal, S. Chaudhuri, and V. Narasayya, “Automated selection of materialized views and indexes for sql databases,” in Proceedings of 26th International Conference on Very Large Databases, Cairo, Egypt, pp. 191–207, 2000.
[45] R. Chirkova, A. Y. Halevy, and D. Suciu, “A formal perspective on the view selection problem,” The VLDB Journal—The International Journal on Very Large Data Bases, vol. 11, no. 3, pp. 216–237, 2002.
[46] T. 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, pp. 53–67, 2017.
[47] A. Jindal, K. Karanasos, S. Rao, and H. Patel, “Selecting subexpressions to materialize at datacenter scale,” Proceedings of the VLDB Endowment, vol. 11, no. 7, pp. 800–812, 2018.
[48] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, “Computation reuse in analytics job service at microsoft,” in Proceedings of the 2018 International Conference on Management of Data, pp. 191–203, 2018.
[49] M. A. Bayir, I. H. Toroslu, and A. Cosar, “Genetic algorithm for the multiple-query optimization problem,” IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 37, no. 1, pp. 147–153, 2006.
[50] L. W. F. Chaves, E. Buchmann, F. Hueske, and K. Böhm, “Towards materialized view selection for distributed databases,” in Proceedings of the 12th international conference on extending database technology: advances in database technology, pp. 1088–1099, 2009.
[51] I. Mami and Z. Bellahsene, “A survey of view selection methods,” Acm Sigmod Record, vol. 41, no. 1, pp. 20–29, 2012.
[52] B. McBride, “Jena: Implementing the rdf model and syntax specification,” in ISWC, pp. 23–28, CEUR-WS. org, 2001.
[53] J. Broekstra, A. Kampman, and F. Van Harmelen, “Sesame: A generic architecture for storing and querying rdf and rdf schema,” in ISWC, pp. 54–68, 2002.
[54] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: sextuple indexing for semantic web data management,” PVLDB, vol. 1, no. 1, pp. 1008–1019, 2008.
[55] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, “Sw-store: a vertically partitioned dbms for semantic web data management,” VLDB J., vol. 18, no. 2, pp. 385–406, 2009.
[56] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, “Column-store support for rdf data management: not all swans are white,” PVLDB, vol. 1, no. 2, pp. 1553–1563, 2008.

[57] T. Neumann and G. Weikum, “x-rdf-3x: fast querying, high update rates, and consistency for rdf databases,” PVLDB, vol. 3, no. 1-2, pp. 256–263, 2010.
[58] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, “Matrix bit loaded: a scalable lightweight join query processor for rdf data,” in WWW, pp. 41–50, ACM, 2010.
[59] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, “Triad: a distributed shared-nothing rdf engine based on asynchronous message passing,” in SIGMOD, pp. 289–300, ACM, 2014.
[60] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, “Rdfox: A highly-scalable rdf store,” in ISWC, pp. 3–20, 2015.
[61] J. Huang, D. J. Abadi, and K. Ren, “Scalable sparql querying of large rdf graphs,” PVLDB, vol. 4, no. 11, pp. 1123–1134, 2011.
[62] X. Zhang, L. Chen, Y. Tong, and M. Wang, “Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud,” in ICDE, 2013.
[63] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis, “Optimizing query shortcuts in rdf databases,” ESWC, pp. 77–92, 2011.
[64] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, “View selection in semantic web databases,” PVLDB, vol. 5, no. 2, pp. 97–108, 2011.
[65] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, “Graph-aware, workload-adaptive sparql query caching,” in SIGMOD, pp. 1777–1792, ACM, 2015.
[66] J. Wang, N. Ntarmos, and P. Triantafillou, “Graphcache: A caching system for graph queries,” in EDBT, pp. 13–24, 2017.
[67] J. Wang, N. Ntarmos, and P. Triantafillou, “Indexing query graphs to speedup graph query processing,” in EDBT, pp. 41–52, 2016.
[68] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos, “Extended characteristic sets: graph indexing for sparql query optimization,” in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 497–508, IEEE, 2017.
[69] F. Picalausa and S. Vansummeren, “What are real sparql queries like?,” in SWIM, p. 7, ACM, 2011.
[70] A. Bonifati, W. Martens, and T. Timm, “An analytical study of large sparql query logs,” PVLDB, vol. 11, no. 2, pp. 149–161, 2017.
[71] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt, “Getting the most out of wikidata: Semantic technology usage in wikipedia’s knowledge graph,” in ISWC, pp. 376–394, 2018.
[72] H. Gupta, “Selection of views to materialize in a data warehouse,” in International Conference on Database Theory, pp. 98–112, Springer, 1997.
[73] D. Theodoratos, T. Sellis, et al., “Data warehouse configuration,” in VLDB, vol. 97, pp. 126–135, 1997.
[74] V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Implementing data cubes efficiently,” ACM SIGMOD Record, vol. 25, pp. 205–216, 1996.
[75] Y. Kotidis and N. Roussopoulos, “A case for dynamic view management,” TODS, vol. 26, no. 4, pp. 388–423, 2001.
[76] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis, “Rolap implementations of the data cube,” ACM Computing Surveys (CSUR), vol. 39, no. 4, p. 12, 2007.
[77] K. Messanakis, P. Demetrakopoulos, and Y. Kotidis, “Smart-views: Decentralized OLAP view management using blockchains,” in Big Data Analytics and Knowledge Discovery, vol. 12925 of Lecture Notes in Computer Science, pp. 216–221, Springer, 2021.
[78] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, “Streaming graph partitioning: an experimental study,” Proceedings of the VLDB Endowment , vol. 11, no. 11, pp. 1590–1603, 2018.
[79] E. Kharlamov, T. Mailis, G. Mehdi, C. Neuenstadt, Ö. L. Özçep, M. Roshchin, N. Solomakhina, A. Soylu, C. Svingos, S. Brandt, M. Giese, Y. E. Ioannidis, S. Lamparter, R. Möller, Y. Kotidis, and A. Waaler, “Semantic access to streaming and static data at Siemens,” J. Web Semant., vol. 44, pp. 54–74, 2017.
[80] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller, “Towards analytics aware ontology based access to static and streaming data,” in ISWC, pp. 344–362, 2016.
[81] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özçep, C. Svingos, D. Zheleznyakov, Y. Ioannidis, S. Lamparter, R. Möller, and A. Waaler, “An ontology-mediated analytics-aware approach to support monitoring and diagnostics of static and streaming data,” J. Web Semant., 2019.
[82] E. Kougioumtzi, A. Kontaxakis, A. Deligiannakis, and Y. 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, pp. 188–189, 2021.
[83] D. Ting, “Approximate distinct counts for billions of datasets,” in Proceedings of the 2019 International Conference on Management of Data, pp. 69–86, 2019.
[84] D. Bleco and Y. Kotidis, “Using entropy metrics for pruning very large graph cubes,” Information Systems, vol. 81, pp. 49–62, 2019.