Efficient Structure-aware OLAP Query Processing over Large Property Graphs

by

Yan Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Property graph model is a popular semantic rich model for real-world applications concerning graph structure data, e.g.,communication networks, social networks, financial transaction networks and etc. On-Line Analytical Processing (OLAP) provides an important tool for data analysis by allowing users to perform data aggregation through different combinations of dimentions. For example, given a Q&A forum dataset, in order to study if there is a correlation between user's age and his or her post quality, one may ask what is the average user's age grouped by the post score. Another example is that, in the field of music industry, we may process a query asking what is total sales of records with respect to different music companies and years so as to conduct a market activity analysis.

Surprisingly, current graph databases do not efficiently support OLAP aggregation queries. On the contrary, in most cases they transfer such queries into a sequence of operations and compute everything from scratch. For example, Neo4j, a state-of-art graph database system, processes each OLAP query in two steps. First, it expands the nodes and edges that satisfy the given query constraint. Then it performs the aggregation over all the valid substrctures returned from the first step. However, in warehousing data analysis workloads, it is common to have repeating queries from time to time. Computing everyting from scratch would be highly inefficient. Moreover, since most graph database systems are disk based due to the large size of real-world property graphs, it is infeasible to directly employ a graph database system like Neo4j for such OLAP workloads.

Materialization and view mainteance techniques developed in traditional RDBMS are proved to be efficient and critical for processing OLAP workloads. Following the generic materialization methodology, in this thesis we develop a structure aware cuboid caching solution to efficiently support OLAP aggregation queries over property graphs. Different from the table based materialization, graph queries consists of both topology structure and attribute combination. The essential idea is to precompute and materialize some views wisely using the query statistics from history workload, such that future workload processing can be accelerated.

We implemented a prototype system on top of Neo4j. Comparing to Neo4j's native support for OLAP queries, an empirical studies over real-world property graph in different size scales show that, with a reasonable space cost constraint, our solution usually achieves 10-30x speedup in time efficiency.

Acknowledgements

I would like to thank Professor M. Tamer Özsu and Dr. Xiaofei Zhang who made this thesis possible.

Dedication

This is dedicated to my mother Limei Leng whom I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Being a flexible and semantic rich model for graph structured data, the property graph model has been widely adopted and we have seen emerging Graph database systems supporting this model, like Neo4j [24], PGX [11]. Supporting OLAP (On-Line Analytic Processing) is one critical feature of modern database systems, because efficient OLAP processing is fundamental to many decision-making applications, e.g., smart business [21], market analysis [14], trend monitoring [8], risk management [5]. However, empirical studies show that existing graph database systems do not efficiently support OLAP workloads, especially structure wise aggregation queries. Moreover, current graph database systems do not support view-based query or materialize some "hot" intermediate results to serve future queries. Therefore, in this thesis, we study the efficient processing of OLAP queries over property graph data using a materialization approach.

1.1 Property Graph Model

We are living in an age with exponential growth of data, and a world that is more and more connected. With the fast development of Web2.0 and Internet of Things(IoT) [1], numerous connections of various kinds are being created every second, producing massive amount of graph structure data in the meanwhile. For example, the moment a user creates a new post on a online forum, not only a post is created, a "*creates*" connection between the user and the post is established as well; when a user tags a post, a "*hasTag*" connection is created between certain tag string and the post; or in a banking scenario, when a transfer happens, a "*transfers*" connection between two accounts is created.

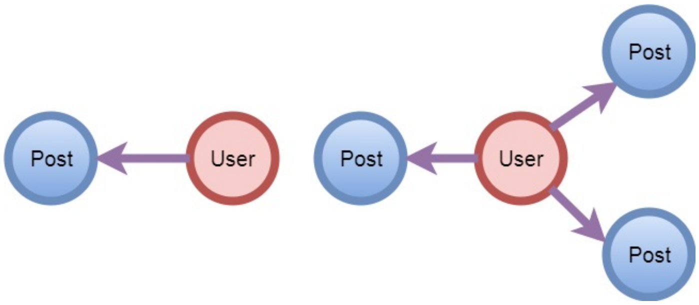

Figure 1.1: A simple property graph modeling “users post posts”(data graph).

To capture the rich semantic of connected real-world entities, property graph model [] is becoming more and more popular considering its flexibility for semi-structured graph data. A property graph consists of nodes, edges, and properties. Like general graph data models, nodes represent entities and edges represent relationships. Graph nodes and edges can have any number of properties, or attributes, of any type. For example, Figure 1.1 shows a simple property graph of an online Q&A forum named www.StackExchange.com. It shows the connections among users (represented by red nodes) and posts (represented by blue nodes). Each arc pointing from a user node to a post node represents a “User_onws_Post” connection. From the graph, we can clearly see that there is one user who has created one post while the other usr has created 2 posts. In addition, as shown in the example, a User node can have properties like the users Age, Views, UpVotes and etc. (listed at the end of the picture). For clear presentation purpose, we shall use a property graph dataset obtained from `www.StackExchange.com` through this thesis. We name this graph “StackExchange graph”.

Note that although the property graph model does not enforce any restriction on what properties a node or edge can have, a highlevel abstraction describing the property relations, named the meta graph, is ofen defined in practice. Meta graph demonstrates the information of entities and entity correlations on a schema level, while data graph refers to the actual graph populated from the meta graph. Figure **??** and Figure **??** are the meta graph and a snapshot of the StackExchange graph, respectively. As shown in Figure **??**, there are three types of entities: User (in red), Post (in blue), and Tag (in green). Each user has a property named “View”, each post has a property named “Score”, and a property “Tagname” associated with each tag. There are two types of edges being defined: User_owns_Post and Post_hasTag_Tag.

1.2 OLAP over Property Graph

In tradition databases and ware-housing, OLAP queries enable users to interactively perform aggregations on underlying data from different perspectives(combinations of dimensions). There are three typical operations in OLAP. Roll-up operation allows user to view data in more details while drill-down operation does the opposite way. Slicing enables filtering on data. For instance, we can perform OLAP to analyze earning performance of an international company by different branch. We can perform drill-down operation by adding season as a dimension besides branch to take a closer look at profit performance of different branches in different seasons. In this case, OLAP serves as a tool for managers to better understand earning performance.

Supporting efficient OLAP processing on property graphs grants users the power to perform insightful analysis over structured graph data. For example, on the StackExchange graph, users can study the correlation between the number of UpVotes and a post's score by using the following query:

Get the average post score grouped by users upvotes.

If the result shows a tight correlation, it suggests that an authors upvotes can be used to estimate the quality of his or her post when a post is freshly posted and score of the post has not been settled.

Consider another example, using a property graph dataset on music industry, one can issue the following query to evaluate a company's strategy to increase the share of young people's market.

Get the total sum of music purchases by buyers at age 18-25 grouped by music company and month

For simplicity, we call such kind of OLAP query workloads over property graphs as "Graph OLAP". As a matter of fact, graph OLAP has already been applied in various senerios like business analysis and decision making and it is attracting increasing research interests in the database community.

1.3 Challenges of Graph OLAP

Supporting efficient OLAP in traditional RDBMS or warehousing applications is a well studied topic. There are abundent literature attacking this problem from virous different perspectives, e.g. data partition [6], view selection [16], partial materialization [7]. However, there is very few research effort on the Graph OLAP. Existing literatures concerning OLAP workload over graph data either target on accelerating graph OLAP over a special subset of property graphs [25], or focus on generic highlevel topics, such as [18] [4], other than time efficiency issue of query processing.

Our empirical studis show that existing graph databases do not provide efficent support for graph OLAP, especially when the graph size scales to real-word practices, which usually contains over millions of nodes and edges. To elaborate, Neo4j, a state-of-art graph database, processes OLAP queries in a rather straightforward manner: computing everything from scratch for each query without being aware of any history workloads. In an extreme case, even if we executed the same query repeatly with only minor change on value constraints, e.g., change the constraint of user's age from 20 to 22, the execution plan always stays the same and yields no execution time improvement.

Valuable information extracted from history workload can be helpful to accelerate incoming query processing. For example, the above exampling OLAP query on StackExchange graph dataset (of roughly 45GB in size) takes Neo4j more than 2 hours to process. It is frustrating for users to wait that long for the result of one single OLAP query, as it undermines interactivity which is one of the most distinctive features of OLAP.

As a matter of fact, history workloads provide useful information for future workloads. This is because in real case users do not generate OLAP queries randomly. Instead users often tend to be interested in some specific "hot" structures on a meta graph level and some "hot" properties. Such interest is contained in history workload and can serve as an insightful hint on future workloads. Suppose we sacrifice some memory space and materialize "hot" structures and properties even before future queries arrive, future queries can be faster processed.

We know that materializing user's interested structures and properties benefits future workload processing, at the cost of extra space overhead. The real challenge is how to design a score function to evaluate the trade-off between such benefit and cost so that we can use the score function to select best materialization. Here best materialization refers to the case where we achieve best future workload acceleration with a given memory constraint for materialization.

1.4 Our Solution and Contributions

To address the challenges discussed above, we propose a end-to-end solution for graph database to support efficient OLAP over large property graphs.

The essence of our solution is to precompute and materialize popular intermediate results that can be reused by future workloads. Intuitively, in real practice, most OLAP queries from the same client tend to reside in several particular structures and properties (usually closely related with the topics that the client is interested in). Within a specific period of time, there are “hot” structures that the client tends to repeatedly investigate from different dimensions. Therefore, previous queries can be used as a good reference to discover structures and properties in which the client is particularly interested.

A good analogy of this is establishment of materialized views in relational databases and processing queries directly on materialized views. In relational databases, we are allowed to build materialized views on structures and attributes that we are interested in. Hopefully when future queries come, we can faster process them using pre-materialized views. Unfortunately, current graph databases do not support similar operations.

There are two most important problems that we need to solve. One key issue is smart selection of “materialized views”. We need to select and pre-compute those that are most beneficial for future queries. Another key issue is how to optimize a better execution plan for answering a future query efficiently using the precomputed materials. To address the first issue, we develop a score function to evaluate costperformance ratio of a materialization. We propose a greedy algorithm to select candidate based on their score (calculated from score function), one by one until memory limit is hit. For the second challenge, if a future query result can be directly produced using a materialization we simply do it. For other cases, we propose a scheduling policy to decompose a future query into substructures and join such substructures to produce final result.

To highlight, we summurize our contributions in this thesis as follows:

- We designed an end-to-end system that realizes structure-aware OLAP query processing on graph databases using precomputation based on previous workloads.
- We implemented our system on Neo4j.
- We proposed our algorithm for smart selection of structures and cuboids to be precomputed.
- We suggested different ways for future query processing. We tested their performances and gave explanations on the performance differences.

The following contents are organized as follows: we discuss the preliminaries and related work in Chapter 2. Followed by the background knowledge about OLAP, graph databases, and Neo4j, we give a summarization of existing literatures concerning OLAP queries over graph data. In Chapter 3 we explain our solution framework and system design in details. We present the experiment design and result disucssion in Chapter 4. Chapter 5 concludes this thesis with highlight on opening questions and future work.

Chapter 2

Background and Related Work

In this chapter, we first explain graph OLAP with real examples. Then we briefly introduce Neo4j, a state-of-art graph database system, which is employed as the back end of our proposed solution. In addition, we review and summarize the most recent relevant works on graph OLAP processing.

2.1 OLAP over Property Graph Model

Following the introduction of the property graph model given in the previous chapter, we further define the syntax of properties adopted in this thesis. In the property graph model, each node and edge could have arbitrary number and type of properties. A type of property is represented as follows:

$$[NodeType].[PropertyType]$$

For example, User.Age denotes an "Age" attribute associated with a node of type "User". In order to identify a node or edge, a unique ID is assigned to each node and edge. For simplicity, in this thesis we represent a node or an edge with its ID, denoted as ID(node) or ID(edge). Note that unique ID is sometimes treated as a special type of property.

OLAP (On-Line Analytical Processing) [2, 10, 26] usually employs a cube concept, which is constructed over multiple attributes, in order to provide users a multi-dimensional and multi-level view for effective data analysis from different perspectives and with multiple granularities. The key operations in an OLAP framework are slice/dice and roll-up/drill-down, with slice/dice focusing on a particular aspect of the data, roll-up performing generalization if users only want to see a concise overview, and drill-down performing specialization if more details are needed. We shall detail the cube technique from the graph data perspective later this chapter.

Graph OLAP is first proposed by Graph Cube [25]. It refers to OLAP over graphs. Though no formal definition of the notion "Graph OLAP" is given in [25]. Graph Cube [25] views the outcome of Graph OLAP as aggregated graphs (aggregation of data graph). On the contrary, in our work, we consider the outcome of Graph OLAP as result tables of OLAP queries.

Graph Cube [25] addresses and defines two most important notions in graph OLAP scenarios as *dimension* and *measure*. In our work emphasize *structure* (of meta graph) as a third important notion. Graph Cube [25] focuses more on OLAP senerios over a fixed *structure*, with *dimension* and *measure* varied. In our work, we are able to deal with OLAP workloads over various *structures*.

2.1.1 OLAP Examples

In order to better elaborate how "Graph OLAP" is interpreted in our thesis, consider the following four example scenarios, where we perform OLAP queries over the StackExchange graph.

Example 1 Does the number of high upvotes of a user indicate a high-quality post?

Query #1: Get average post score grouped by users upvotes.

Sample query result:

User.UpVotes	AVG(Post.Score)
0	1.33
1	2.23
2	2.34
3	2.77
4	3.43

From the query result we can see that upvotes can be used as a good indicator of a users post quality. Suppose we would like to propose suggested posts based on scores. When a post is freshly posted and score of the post has not been well voted been yet, we may use the authors upvotes as a factor to estimate the quality of his or her post.

Example 2 Following the context of Query #1, but this time we want to take a closer look at Query #1 for different types of questions. If we take upvotes as quality of a user, perhaps quality of a user is shown only in his or her answers, instead of questions. Or is it true that high quality user also asks much better questions?

Query #2: Get average post score grouped by users upvotes and posts post types.

Sample query result:

User.Upvotes	Post.PostTypeId	AVG(Post.Score)
0	1	2.14
1	1	2.26
2	1	2.83
3	1	3.04
4	1	3.46
0	2	1.54
1	2	2.21
2	2	2.18
3	2	2.72
4	2	3.58

The query results suggest that high-quality users not only provide good answers but ask valuable questions as well. However, there is a subtle difference on how upvotes is correlated with questions and answers. For example, a really low upvote level indicates a low-quality answer more than a low-quality question. This is probably because people tend to be more tolerate with a naive question rather than a wrong answer.

Query #1 and Query #2 simply focus on relationship between User and Post. We may switch our attention to a slightly more complicated structure by adding the Tag.

Example 3 In year 2017, which is the weighted average age of users? For instance is python more trendy than c among young users?

Query #3: Get average user age grouped by users 2017 posts tags.

Sample query result:

TagName	AVG(Age)
Router	19.6
Python	24.1
Internet	26.8
C	30.2
programmer	31.4
software	29.8

From the results, one can tell the average user age with respect to each tag clearly and easily compare them. It reveals some interesting insight: python is generally more popular among younger users; and “Router” is a relatively “younger” topic than “Internet”.

Example 4 Find out the tendency of topics “average popular user age” by years. Is there a tendency of younger age?

Query #4: Get average user age grouped by users posts tags and years.

Sample query result:

TagName	Year	AVG(Age)
Router	2012	22.1
Router	2017	19.6
Python	2012	27.3
Python	2017	24.1
Internet	2012	27.5
Internet	2017	26.8
C	2012	30.4
C	2017	30.2
programmer	2012	34.2
programmer	2017	31.4
software	2012	31.6
software	2017	29.8

Tendency of younger age on IT topics is revealed from the results. Python is getting faster embraced by younger people compared with C. Similarly we can compare two commercial products customer targeting strategy, advertising performance etc.

From the above OLAP query examples we can see that OLAP over property graphs provides an interactive and informative way to analyze property graphs from multiple dimensions, and thus helps people find the hidden correlations, aggregated effects, regularities, tendencies and so on.

2.1.2 Structure, Dimension, and Measure

We now explain the three key elements of a graph OLAP: *structure*, *dimension*, and *measure* using Query #1 as an example.

Query #1 is concerns the following structure (colored in blue) on the meta graph:

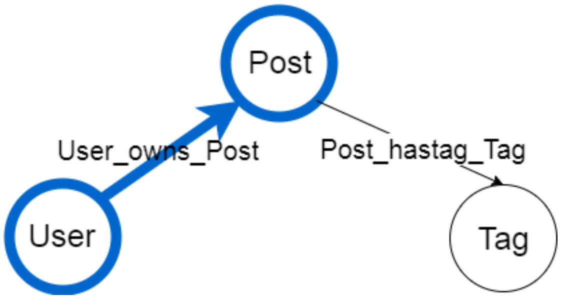

Figure 2.1: *Structure* of Query #1

We say that (User)-[User_owns_post]->(Post) is the structure of Query #1. The query is first aggregated on users upvotes. We say that User.Upvotes is the dimension of Query #1. And the output of the query is an aggregation function on posts score. We say that AVG(Post.Score) is the measure of Query #1. Similarly, consider the above Example 2, which shares the same structure as shown in Figure 2.1. The dimensions of Query #2 is User.Upvotes, Post.PostTypeId, and the measure is AVG(Post.Score). Note that Query #2 adds Post.PostTypeId to Query #1s dimensions. In other words, Query #2 asks for a more detailed partitions over dimensions. We call Query #2 a drill-down from Query #1, and Query #1 is a roll-up from Query #2. Note that possible property combinations can be modeled as a lattice-structured cube. Figure 2.2 shows what a cube is like for properties {A,B,C}. We can see that roll-up and drill-down operations allow us to navigate up and down on a cube.

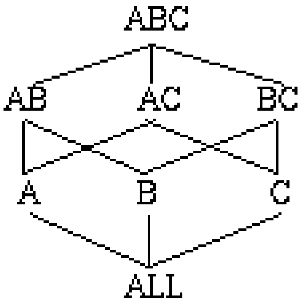

Figure 2.2: Cube of properties {A,B,C}.

Query #3: Get average user age grouped by users 2017 posts tags.

Structure: (User)-[User_owns_post]-(Post)-[Post_hastag_Tag]-(Tag)

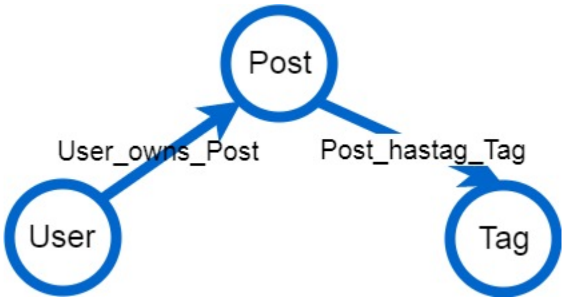

Figure 2.3: *Structure* of Query #3

Dimensions: Tag.Tagname

Measures: AVG(User.Age)

Note that Query #3 has a different *strucutre* than Query #1 and Query #2, as shown in Figure 2.3. Query #3 enforces a requirement that post must be created in year 2017, which picks out a particular subset of the posts. In OLAP this is called "slicing" operation. Slicing operation allows users to view the data with filtering requirements on selected properties. In this thesis we call the constraint Post.Year=2017 of Query #3 a *"slicing condition"*.

To summarize, graph OLAP allows clients to aggregate different *structures*, over different *dimensions*, on different *measures*, and optionally slice aggregation result by different *slicing conditions*. Clients can change their views by performing roll-up, drill-down, and slicing freely and interactively.

2.2 Graph Databases and Neo4j

Emerging online applications concerning graph processing has motivated the relational database community to support efficient graph management []. However, there has been active debate about the efficiency of using traditional RDBMS for graph computing considering the unique query workload against graph data [], which is beyond the scope of this thesis. As a matter of fact, relational databases and graph databases both have their own strengths in term of query processing. It is generally accepted that graph databases perform better at property graph data processing as it conforms more with the actual graph structure. For clear presentation purpose, we highlight some key differences between the RDBMS and graph database.

Relational databases model graph data as entity and relationship tables. For example, given a simple property graph shown in Figure 2.4, which consists of 1 user and 3 posts, a relational database stores the graph with 3 tables:

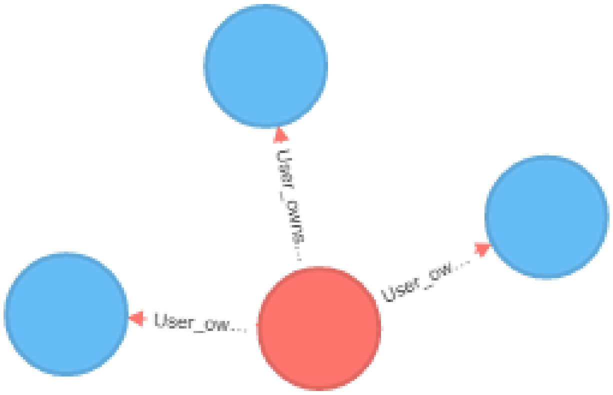

Figure 2.4: A simple property graph.

User		
Uid	Age	UpVote
1	22	5

Post	
Pid	Score
1	0.5
2	0.8
3	0.6

Owns	
Uid	Pid
1	1
1	2
1	3

There are two drawbacks of storing property graphs in a relational database. First, each node or edge in a property graph could have arbitrary types of properties. However, relational table schema would restrict nodes or edges of the same type to have a uniform set of properties (attributes). Second and more importantly, edges are stored as a separate table in relational databases. Thus, we cannot directly query all the posts of a given user without joining User and Own tables in the above example.

Graph databases solve the above two issues by directly adopting property graph structures to store data. In graph databases, edges are stored not as independent tables but directly attached to related nodes using data structures such as adjacency lists. Many graph database applications have been implemented and commercialized. One of the popular ones is Neo4j, which holds atomicity, consistency, isolation, durability (ACID) as traditional RDBMS does. Database instances in Neo4j are modeled and stored as property graphs. One thing special about Noe4js property graph is that its nodes and edges can be labeled with any number of labels (similar to entity and relationship types). For example, a node referring to a student could have various labels such as student, people etc.

Cypher is Neo4js query language, which is expressive and simple. For example, consider the following query: what is the average score group by different user upvotes when PostTypeID is 2? A Cypher query would be written as follows:

```
match (u:User)-[r:User\_owns\_Post]-$>$(p:Post)
where p.PostTypeId='2'
return u.Upvotes, AVG(p.Score)
```

In the above Cypher query, "User" and "Post" are node labels, PostTypeId and Score are properties of "Post", "UpVotes" is a property of "User".

2.3 Related Work

There have been a few work discussing efficient graph OLAP queries on attribute graphs or RDF graphs.

Cube-based [12] proposes the concept of graphs enriched by cubes. Each node and edge of the considered network are described by a cube. It allows the user to quickly analyze the information summarized into cubes. It works well in slowly changing dimension problem in OLAP analysis.

Gagg [18] introduces an RDF graph aggregation operator that is both expressive and flexible. It provides a formal definition of Gagg on top of SPARQL Algebra and defines its operational semantics and describe an algorithm to answer graph aggregation queries. Gagg achieves significant improvements in performance compared to plain-SPARQL graph aggregation.

Pagrol [23] provides an efficient MapReduce-based parallel graph cubing algorithm, MRGraph-Cubing, to compute the graph cube for an attributed graph.

Graph OLAP [4] studies dimensions and measures in the graph OLAP scenario and furthermore develops a conceptual framework for data cubes on graphs. It differentiates different types of measures(distributive and holistic etc) by their properties during aggregation. It looks into different semantics of OLAP operations, and classifies the framework into two major subcases: informational OLAP and topological OLAP. It points out a graph cube can be fully or partially materialized by calculating a special kind of measure called aggregated graph.

In Graph Cube [25], concepts of graph cube is introduced. Given a particular structure S, a property set P, and measure set M. We can aggregate over S on $2^{|P|}$ different combinations of dimensions. These $2^{|P|}$ queries can be mapped as a lattice structure, where each combination of dimensions corresponds to a cuboid in the lattice. We call the lattice structure of these $2^{|P|}$ queries a graph cube.

It has been pointed out in Graph OLAP [4] that as long as if domain of measure is within {count, sum, average} and M contains count(*), the following feature holds: given any two cuboids C_1 and C_2 from the same graph cube, as long as dimension(C_2) is a subset of dimension(C_1), result of C_1 can be used to generate result of C_2. This is to say once a cuboid is materialized, all roll-up operations from this cuboid could be processed simply by scanning the materialized cuboid result. This will dramatically decrease roll-up operation time compared to aggregation from data graph(often of larger size, disk I/O), scanning materialized cuboid result(often of smaller size) is often much faster.

Ideally we can materialize all cuboids. But when number of dimension is large, number of cuboids grows exponentially, making total materialization impossible due to overwhelming space cost. To solve this Graph Cube [25] proposed a partial materialization algorithm on graph cube. It is a greedy algorithm and the score function is based on benefits of deduction of total computation cost.

	G. Type	Q. Pattern	Layered	Featuer
Cube-based [12]	Property	Simple relation	yes	Cubes on edges and nodes
Gagg [18]	Property	Exact match	no	Structural patterns
Pagrol [23]	Property	edge & node attributes	yes	Map-Reduce computing
Graph Cube [25]	Homogenous	node attributes	yes	Partial materialization
Graph OLAP [4]	Property	edge & node attributes	yes	Distributive and holistic measures

Table 2.1: A summary of graph OLAP literature

We summarize some of the most related ones as follows:

From the summary, we can categorize the existing work into two lines. First, like Graph Cube [25], researches focus on a simple subset of property graphs(e.g. graphs with only homogenous nodes and edges) and proposes optimizations in order to accelerate OLAP query processing. The optimizations are attribute-aware, and since the nodes and edges are of only one kind queries over different structures and structure-aware optimizations are out of the scope. Second, like Gagg [18], researches focus on an abstract high-level framework that process generic queries over generic property graphs. However, query processing efficiency is not studied.

To conclude, we can see a lack of study on structure-aware optimizations for efficient graph OLAP. As mentioned in Section 1.3, efficiency issue is one of the most challenging issues on graph OLAP. Therefore, it is very meaningful to explore faster structure-aware OLAP processing over general property graphs.

Chapter 3

Problem Definition

In this section, we first introduce the terminology and notations adopted in this thesis. Then we formally define the efficient OLAP query processing problem.

3.1 Terminologies

We first present terminologies on property graph model and OLAP query. Then we introduce the concepts of “cuboid” and “substructure”, which are two types of materializations we will use in our solution.

3.1.1 Definition of Property Graph

Given the property graph examples elaborated in Section 1.1, we formally define the property graph model employed in this thesis.

A property graph G is defined as $(V, Vid, E, Eid, A, L, f)$. V is the set of nodes. Vid is the set of unique IDs of each node in V. E is the set of edges, where $E \subseteq V \times V$. Eid is the set of unique IDs of each edge in E. A is a set of predefined properties. L is a set of predefined labels. f=$\{f_{VA}, f_{VL}, f_{Vid}, f_{EA}, f_{EL}, f_{Eid}\}$ is a set of mapping functions, such that:

- $f_{VA} : v_i \rightarrow A_i, v_i \in V, A_i \subseteq A$, maps each node to its properties;
- $f_{VL} : v_i \rightarrow L_i, v_i \in V, L_i \subseteq L$, maps each node to its labels;

- $f_{Vid} : v_i \to vid_i, v_i \in V, vid_i \subseteq Vid$, maps each node to its unique ID;
- $f_{EA} : e_i \to A_i, e_i \in E, L_i \subseteq A$, maps each edge to its properties;
- $f_{EL} : e_i \to L_i, e_i \in E, L_i \subseteq L$, maps each edge to its labels;
- $f_{Eid} : e_i \to eid_i, e_i \in V, eid_i \subseteq Eid$, maps each edge to its unique ID.

3.1.2 OLAP Query

As discussed in Section 2.1.2, four elements of a graph OLAP query are *Structure*, *Dimension*, *Measure*, and *Slicing Condition*(optional). We now define how we represent these four elements in an OLAP query. We will use Query #3 in Section 2.1.1 as an example.

Structure : A *structure* consists of *edges*. We write a *structure* by listing all its *edges* separated by comma, where an *edge* is represented by "*Starting Node Label - Edge Label - Ending Node Label*". For instance, Query #3's *structure* as shown in Figure 2.3 is written as *"User-owns-Post, Post-has-Tag"*.

Dimension: A *Dimension* is written by listing all properties that act as dimensions in an OLAP query. For example, Query #3's *dimension* is written as *"Tag.Tagname"*.

Measure: We focus on three most common types of *measure*: *COUNT, SUM and AVG.* Query #3's *measure* is written as *"AVG(User.Age)"*.

Slicing Conditions: A *Slicing Conditions* is written as "*Property = value*". Query #3's *slicing conditions* is written as *"Post.Year=2017"*.

With the four elements defined above, we write an OLAP query in the following format:

Structure : Dimension, Measure, Slicing Condition

Recall that Query #3 is written as

User-owns-Post, Post-has-Tag: Tag.Tagname, AVG(User.Age), Post.Year=2017

where *User-owns-Post, Post-has-Tag* refers to *structure*; *Tag.Tagname* refers to *dimension*, *AVG(User.Age)* refers to *measure*; and Post.Year=2017 refers to *slicing condition.*

Now we define some notations adopted in this thesis. Given a OLAP query q, we use *"q.properties"* to refer to the set of **all properties** appeared in *Dimension, Measure, and Slicing Condition* of q. We use *"q.structure"* to refer to the structure of q. For example, *Query #3.properties*={*Tag.Tagname, User.Age, Post.Year*}, and *Query #3.structure*={ *User-owns-Post, Post-has-Tag*}.

3.1.3 Materialization: Cuboid & Substructure

As previously discussed, a key issue of our work is to find out most useful queries based on previous workload. For fast access to results of these useful queries, we process them and store their results (preferably in main memory) even before starting on future queries. We call such a result of a query a _materialization_ of the query.

As defined above, in a property graph, each node or edge has an unique ID, which can be treated as a special property. Whether a materialization keeps unique ID is an important issue. This is because keeping unique ID often increases the space cost. We consider two types of materializations, namely the _cuboid_ and the _substructure_, based on whether unique IDs of nodes (or edges) are kept or not. To better elaborate the differences between cuboid and substructure, we consider the following example. Suppose we have following query workload containing two history queries (Q_1 and Q_2) and two in-coming queries (Q_3 and Q_4):

Q_1 User-owns-Post: User.Age
Q_2 User-owns-Post: User.Age, (AVG)Post.Score
Q_3 User-owns-Post: (AVG)User.Age, Post.Score
Q_4 User-owns-Post, Post-has-Tag: User.Age, Tag.TagName

We can tell that users are most interested in the _User-owns-Post_ structure. {_User.Age, Post.Score_} is the set of properties being involved in queries over _User-owns-Post._ Thus, we can build a cuboid lattice of all combinations of {_User.Age, Post.Score_}. Materialization of the base cuboid of the lattice is

M_1 _\$User-owns-Post: User.Age, Post.Score, COUNT(*)_

Note that for the rest of this thesis, we use the \$ symbol followed by structures and dimensions to denote a materialization, represented by M. Apparently, the above materialized view is useful for Q_3. We can process Q_3 by aggregation over M_1. We call such materialization a **cuboid**.

However, M_1 is not useful for Q_4 since they have different _structures._ On the contrary, If we add _ID(Post)_ into _dimension_ and materialize _\$User-owns-Post: User.Age, Post.Score, ID(Post) COUNT(*)_, _Post_ is “activated” to be able to join with other materializations containing _Post_ and produce results for OLAP over more complicated _structures._ For example, Q_4 can be processed through the following steps:

Step 1: joining _\$User-owns-Post: User.Age, Post.Score, ID(Post) COUNT(*)_ and _\$Post-has-Tag: ID(Post), Tag.TagName, COUNT(*)_ on _ID(Post)_;

Step 2: perform aggregation on {_User.Age, Tag.TagName_}.

	Cuboid	Substructure
Dimension	Only properties	Properties and ID(s)
Space Cost	“Low”	“High”
Potential benefit	Aggregation	Aggregation & Joining

Table 3.1: Comparisons between Cuboid and Substructure.

In this case, we only need to fetch *$Post-has-Tag: ID(Post), Tag.TagName, COUNT(*)* from database to produce result for future workload #2. We call such materialization with ID(s) in *dimension* as **substructure**.

Table 3.1 shows a comparison between cuboid and substructure. Note that cuboids can only be used in queries with exactly the same structure. They are good for roll-up and slicing operations but not useful for queries with different structures. Substructures can be used to join with other materializations to help with future queries of various types of structures. The drawback is that structures are generally more space-costly than cuboids, as IDs are unique keys. The trade-off between cuboids and substructures is the trade-off between space cost against the potential saving of join processing.

3.2 Problem Definition

Intuitively, our goal is to answer OLAP queries efficiently by taking the advantage of materialized views, which are constructed based on the knowledge of previous workloads. Therefore, our solution needs to take two steps:

- Materialization step: materialized view selection.
- Query processing step: answer future queries as quickly as possible (using materializations).

The materialization step is in fact a “Materialization Selection” (MS for short) problem, as using materialization is good for query efficiency, but comes with a storage cost. So we want to study the problem of how to best utilize materialization within a space budget limit σ. While the query processing step lies in “Execution Planning” (EP for short). We formally define these two problems as follows.

Materialization Selection Problem: Given a property graph dataset G, a set of previous queries P on G, space limit σ, find cuboids C and substructures S, such that: 1) $\sum\limits_{c_i \in C} c_i.\mathit{space} + \sum\limits_{s_i \in S} s_i.\mathit{space} \leq \sigma$; and 2) $\sum\limits_{p_i \in P} T(G, p_i, C, S)$ is minimized.

Here $T(G, p_i, C, S)$ is a function to estimate the query processing time of p_i on G using C and S. "*.space*" refers to the estimated space cost of a cuboid or substructure. Note that the real running time of a particular query is hard to estimate. Therefore, we use $T(G, p_i, C, S)$ to serve as a cost function to measure the time cost of query processing.

Execution Planning: Given a property graph dataset G, a future query q, materialized cuboids C and substructures S, find a processing plan $\mathit{process}(G, q, C, S)$, such that the processing time of q, denoted by $\mathit{process}(G, q, C, S).\mathit{time}$, is minimized.

As a matter of fact, the execution planning can be further divided into two subproblems. First, which materialized views in C and S should be used to answer q? Second, how to answer q as fast as possible using the view selected from the first question. Thus, we define the first question as the *Decomposition Problem*, which decomposes q into two parts, one part is covered by the views from C and S, while the other part is not covered, which is named as the *remaining views*. Note that the remaining views refer to the data that needs to be fetched from database server on the fly. We define the second question as the *Composition Problem*, which performs basic relational operations such as join, projection and selection over views in order to get the result of q.

Composition Problem: Given a property graph dataset G, a future query q, materialized cuboids C' and substructures S', and remaining views R; find a composition plan $\mathit{compose}(G, q, C', S', R)$, so that estimated composition time $\mathit{compose}(G, q, C', S', R).\mathit{time}$ is minimized. Here $\mathit{compose}(G, q, C', S', R)$ returns result of query q by performing operations (join, selection, projection etc.) over C', S', R.

Decomposition Problem: Given a property graph dataset G, a future query q, materialized cuboids C and substructures S, a composition plan $\mathit{compose}(G, q, C, S, R)$; find $C' \subseteq C$,S'$\subseteq S$, and remaining views R, so that $\mathit{compose}(G, q, C', S', R).\mathit{time}$ is minimized.

Note that we define "Composition Problem" before "Decomposition Problem". The reason is that we need to consider a composition plan $\mathit{compose}(G, q, C', S', R)$ when making our selection policy of C', S' and R. In other words, these two problems are closely correlated.

Chapter 4

Solution

In this chapter, we present our complete solution towards efficient OLAP query processing over property graphs. For comprehensive presentation, we first illustrate the overall solution framework in Section 4.1. Then we present our strategy for materialized view selection, as well as the execution planning for query processing in Section 4.2 and 4.3, respectively.

4.1 Solution Framework Overview

Figure 4.1 describes the overall solution framework. Two dash line rectangles represents the major components of our solution: materialization and query processing. Materialization takes previous workload as input and performs materialization. We adopt a straightforward best effort approach for view selection. Intuitively, we first partition previous queries into “hot” queries and “less hot” queries based on the frequency count of their structures. Modules CubePlanner and StructurePlanner take “hot” queries and “less hot” queries as input and produce cuboids and substructures (in form of tables) for materialization respectively. More details are left to Section 4.2.1, where we will explain the intuition of categorization of “hot” and “less hot” queries, as well as the reason of passing them to different planners. Query processing component takes in-coming queries as input and returns results. Briefly, the work flow of query processing is the following. If a new query happens to contain a “hot” structure, we consult cuboid materializations to see if it can be directly answered by aggregation over a cuboid materialization. In this case, cuboid materialization will be used. Otherwise, if the query cannot be directly answered by any materialized cuboid, we consider available materialized substructures by decomposing the

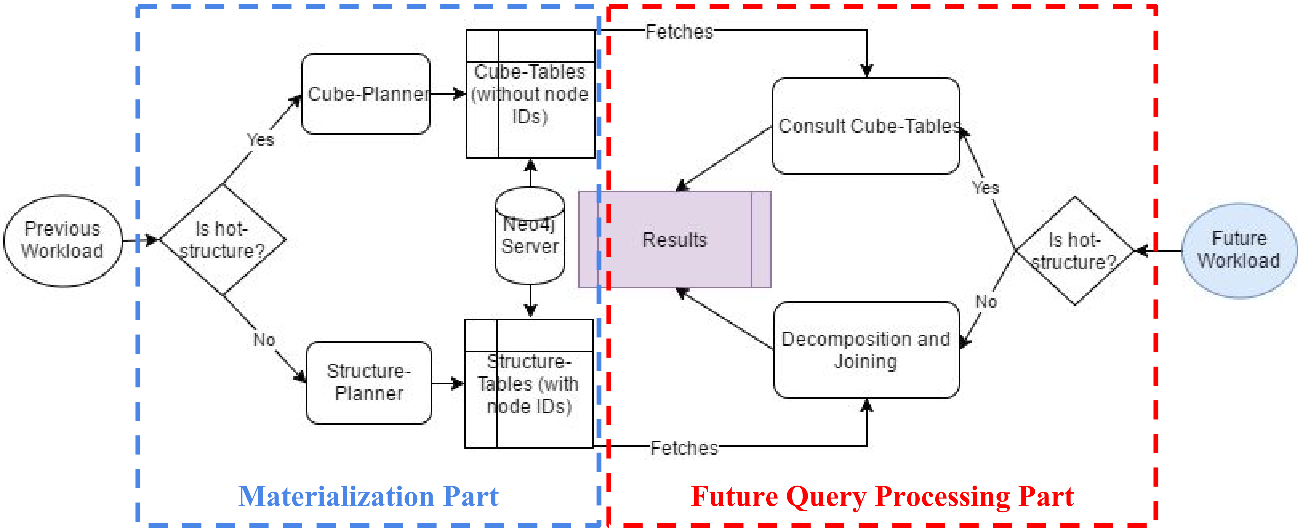

Figure 4.1: Solution framework.

query into substructures for join. Note that if required substructure is not materialized, on the fly data fetching from the graph database server is mandatory.

4.2 Materialized View Selection

Materialized view selection is a profound research problem that attracts enormous efforts in traditional RDBMS community. However, the fundamental difference between the relational model and the property graph model makes materialization of graph database an interesting topic to explore. As briefly illustrated before, we consider two types of materializations for efficient OLAP query processing: cuboids and substructures. In this section, we first elaborate the essential heuristic of selecting cuboids and substructures. Then we detail the approaches taken for different types of materializations.

4.2.1 Overview of Materialized View Selection

In Section 3.1.3, we have discussed about the trade-off between cuboids and substructures. We know that utilization of a cuboid materialization requires future queries to have exactly the same structure as the materialized cuboid. Therefore, it is only reasonable to materialize a cuboid when we are confident that the same structure is likely to be "hit" by future queries. Otherwise, it is simply a waste of space to materialize cuboids that would be rarely "hit". On the contrary, substructures do not have such strict structure match requirement. A substructure can be used as long as it appears in a future query.

Considering the different features of cuboids and substructures, we take the following strategy for materialized view selection. We first perform a frequency count of previous queries. If more than ω queries sharing the same structure, where ω is a predefined frequency threshold, this structure is considered as a *hot structure* and would be passed on to the *CubePlanner* module for cuboid selection. Queries do not have *hot structures* are passed to *StructurePlanner* for the substructure selection. Algorithm 1 describes the overall framework of the materialized view selection process.

Algorithm 1: Materialization Overview

```
System setting: $\omega$: frequency threshold for hot structures
Input: Q: a set of previous queries
Output: C: a set of materialized cuboids
S: a set of materialized substructures
$CInput \leftarrow \emptyset$;
$SInput \leftarrow \emptyset$;
foreach $q \in Q$ do
    if $structureFreq(Q,\ q) > \omega$ then
        $CInput \leftarrow CInput \cup \{q\}$;
    else
        $SInput \leftarrow SInput \cup \{q\}$;
    end
end
$C := materialize(CubePlanner(CInput))$;
$S := materialize(StructurePlanner(SInput))$;
```

As shown in Algorithm 1, function $structureFreq(Q, q)$ returns the frequency count of all queries' structures in Q. After *hot structures* are selected, two functions *CubePlanner* and *StructurePlanner* are called to select cuboids and substructures for materialization. Note that we use *materialize()* as a function to denote the materialization of selected cuboids and substructures. To elaborate, consider the following example. Assume we are aware of the six previous queries as shown below. We can group queries by structure and count the structure frequency.

Previous Workload:
#1 Badge-User, User-Post:Badge.Name,Post.Score,Post.PostTypeId=2
#2 User-Comment, Comment-Post: User.UpVotes, Comment.Score, (AVG)Post.Score, Post.PostTypeId=1
#3 User-Post, Post-Vote: User.UpVotes, Vote.VoteTypeId
#4 User-Post, Post-Tag: (AVG)User.CreationDate_Year, Tag.TagName
#5 User-Comment, Comment-Post: User.ActiveMonth, Post.CreationDate_Year=2016
#6 User-Comment, Comment-Post: User.Age, (AVG)Comment.Score, Post.PostTypeId=2
Future Workload:
#1 User-Comment, Comment-Post: User.UpVotes, (AVG)Post.Score, Post.PostTypeId
#2 User-Comment, Comment-Post: User.Age, Post.PostTypeId
#3 User-Post, Post-PostHistory: User.UpVotes, PostHistory.PostHistoryTypeId
#4 Badge-User, User-Post:(AVG)Post.Score,Post.PostTypeId=2

Structure	Frequency
User-Comment, Comment-Post	**3**
User-Post, Post-Tag	1
User-Post, Post-Vote	1

Apparently, *User-Comment, Comment-Post* is a *hot structure*. We materialize cuboids over structure *User-Comment, Comment-Post* by passing previous query #2, #5 and #6 to CubePlanner. CubePlanner will materialize cuboids that benefit processing of future query #1 and #2 (which have *User-Comment, Comment-Post* structure). Then, we pass the three remaining queries of less hot structures, query #1, #3, and #4 to StructurePlanner. StructurePlanner will discover and materialize most useful substructures. In this case StructurePlanner is likely to find *User-Post* as a useful substructure it can be used in joining the result of future query #3 and #4.

4.2.2 Greedy Selection Framework

Before diving into the details of the *CubePlanner* and *StructurePlanner* modules, we first illustrate the essential greedy heuristic employed for view selection.

In our solution framework, *CubePlanner* and *StructurePlanner* are responsible for materialized view selection (over cuboids and substructures, respectively). They both adopt the same greedy selection framework. In Section 3.2, we introduced the "Materialization Selection" problem, which aims at finding best materializations under a space limit σ. Materialization selection is known as a NP-complete problem [17]. The difficulty lies in that the overall benefit of materialized views is not a simple sum of the individual benefit of each view. A materialized view's marginal benefit may be deducted when another view is selected. For example, the marginal benefit of a substructure over "*User-Post, Post-Tag*" will be affected by selecting substructures over "*User-Post*" and "*Post-Tag*". A straightforward approach to solve the materialization selection problem is to enumerate over all possible combinations of cuboids C and substructures S within the space limit σ and find the best combination. But such a brute-force solution is infeasible in practice. In addition, assume that we obtain the optimal C' and S' in some way, it is not guaranteed that the actual total space cost of C' and S' is strictly lower than σ as we only made estimations in our calculation. Therefore, we turn to a greedy algorithm which is better than naive approach in terms of efficiency. Besides, it allows materializations to be done one by one having the space limit being strictly respected.

We will discuss this greedy selection framework first to give a high-level idea of our selection policy. We use a greedy algorithm for both cuboid and substructure selection, as shown in Algorithm 2. The idea is to always pick the next candidate with the highest ratio of margin benefit against the space limit. After a candidate is picked, we re-evaluate the benefit of remaining candidates. Re-evaluation is mandatory as the margin benefit of a candidate may be deducted owing to materialization of a selected candidate.

Algorithm 2: Greedy Selection

```
   System setting: $\sigma$: space limit
   Input: C: a set of candidates of cuboids or substructures in lattice structure
   P: A set of previous queries
   Output: Q: a queue of selected candidates to materialize
1  foreach $c \in C$ do
2      c.space := space(c);
3      c.benefit := estimateMarginBenefit(c, P, Q);
4      c.score := c.benefit/c.space;
5  end
6  while $Q.totalsize < \sigma$ do
7      selected := c in C with highest score;
8      Q.Enqueue(selected);
9      repeat Lines 1-5;
10 end
11
```

In this algorithm, we use a queue data structure for the output of Q. It is because the order of selection is helpful in later computations. Line 1-5 estimate the space cost, the marginal benefit for future workload, as well as the score for each candidate. We call this phase **score calculation**. Line 6-10 keeps picking up candidates with highest score one by one until space limit is hit. Notice that each time a candidate is selected, Line 9 refreshes scores for all candidates by repeating 1-5. We call this phase **pick-and-update**.

Note that *CubePlanner* and *StructurePlanner* apply this greedy selection framework with different implementations of score calculation and pick-and-update. Users can adjust the behavior of *CubePlanner* and *StructurePlanner* by plug in their own implementation of the score calculation function considering different database features. For the rest of this chapter, we will focus on our implement of *CubePlanner* and *StructurePlanner* in Neo4j.

4.2.3 CubePlanner

CubePlanner takes previous queries with *hot structures* as input and returns the selected cuboids for materializations. As mentioned in Section 3.1.3, a cuboid is only useful for queries sharing the exactly same structure. To put it another way, cuboids of different structures do not affect each other at all in terms of benefits for future queries. As a result even though the input queries for *CubePlanner* may have different structures, we can group queries by structure and treat them individually. For each group of input queries, we propose an algorithm named *SingleCubePlanner* to select top-n cuboids. After all groups are finished, we compute the final top-n cuboids by searching across all groups of queries. A good analogy for such process is to first hold regional competitions and then select national winners from regional winners. Next we will explain *CubePlanner* and *SingleCubePlanner* in details.

CubePlanner

As we mentioned above, *CubePlanner* performs cuboid selection in a holistic manner by one-by-one selection of cuboids from results of *SingleCubePlanners*. We first explain the work flow of *CubePlanner*, as shown in Algorithm 3. Intuitively, *CubePlanner* first groups Q by structure using the function $group(Q)$. Line 2-4 performs cuboid selection in each partition using *SingleCubePlanner*. A queue of ordered candidates is generated within each group of queries. Line 5-8 repeatedly checks the current top candidate of each partition to select the best candidate among them. n is a user defined parameter, denoting the most number of cuboids for materialization. Note that users may choose other ways, such as a space limit, as the bound for cuboid materialization.

Algorithm 3: CubePlanner

```
System setting: : maximum number of cuboids to be precomputed
Input: Q: a set of previous queries not necessarily with a same structure
Output: C: a queue of selected cuboids to be precomputed
Group:= group(Q);
foreach group ∈ Group do
    group.results := SingleCubePlanner(group);
end
for i=1 to n do
    group' := group in Group with highest group.results.top().score;
    C.offer(group'.Dequeue());
end
```

SingleCubePlanner

Now we elaborate the *SingleCubePlanner* function. As shown in Algorithm 4, *SingleCubePlanner* follows a greedy selection strategy to generate the top-n cuboids.

The algorithm starts with building a lattice over all combinations of dimensions of all attributes that appeared in previous query set P, using a classic lattice construction algorithm described in [19]. Line 2-4 initializes the best-so-far processing time for each previous query with its estimated naive database processing time. Line 5-12 performs score calculation following the greedy selection framework presented in Algorithm 1. For each cuboid, we estimates its space (line 6). Line 8-10 calculate the marginal benefit by iterating over previous queries that can be answered on scanning current cuboid. If the estimated scanning time is less than a previous query's current best-so-far processing time, we add the difference of two times to the cuboid's total marginal benefit (Line 9). Line 13-23 performs the pick-and-update, where line 15-17 terminate the selection process when there is no more extra marginal benefit, and line 19-22 update the best-so-far processing time for previous queries as a result of the current round of selection.

Now we explain the implementation details of the time estimation function employed in Algorithm 4. Function $time(query)$ estimates the naive time cost for processing a query in a graph database. Implementation of $time(query)$ is database specific as physical storage and execution plans vary among different databases. Since Neo4j provides APIs to show the execution plan as well as the estimated intermediate result size, we directly use the total size of intermediate results as an estimation of the time cost. For example, Figure

Algorithm 4: SingleCubePlanner

```
   System setting: n: as in “top-n”
   Input: P: a set of previous queries with a same structure
   Output: C: an queue of selected cuboids to precompute
1  $Lattice \leftarrow buildLattice(Q)$;
2  foreach query $Q \in P$ do
3      $q.time \leftarrow time(q)$;
4  end
5  foreach $cuboid \in Lattice$ do
6      $cuboid.space \leftarrow space(cuboid)$;
7      $cuboid.benefit \leftarrow 0$;
8      foreach query $Q \in P$ and $q.properties \subseteq cuboid.properties$ do
9          $cuboid.benefit += max(0, q.time - aggreTime(cuboid))$;
10     end
11     $cuboid.score \leftarrow cuboid.benefit/cuboid.space$;
12 end
13 for i=1 to n do
14     nextBestCube $\leftarrow$ cuboid in Lattice with highest score;
15     if $nextBestCube.score < 0$ then
16         break;
17     end
18     C.Enqueue(nextBestCube);
19     foreach cuboid $Q \in Q$ and $q.dimension \subseteq nextBestCube.dimension$ do
20         $q.time \leftarrow min(q.time, aggreTime(nextBestCube))$;
21     end
22     Repeat 5-12;
23 end
24
```

4.2 is an execution plan provided by Neo4j for query *User-Badge, User-Post, Post-Tag: Tag.TagName*. We can see that the number of estimated rows of intermediate results are provided. We use $\sum$ "*estimated_rows*$''$ to estimate the total processing time cost.

For graph databases which do not provide such APIs to see execution plans and estimated intermediate result sizes, users can construct different estimation function following the same intuition, which usually depends on specific database implementations. There are many studies on cost estimations for database operations (e.g., join operation). Users may consider joining (expanding) order [3] and estimation of intermediate result sizes [22] as two important factors.

Function $aggreTime(cuboid)$ estimates the time cost for scanning a materialized cuboid. Given a cuboid c, we estimate the space cost of c as follows:

$$spacePerRow := \sum_{p \in c.properties} sizeOf(p)$$

Thus,

$$SpaceCost(c) := spacePerRow \times numberOfRows(c)$$

Note that *sizeOf(property type)* refers to the standard size of data types. For example, the integer type in "C++" is 2 byte. *numberOfRows(c)* refers to the number of rows in c. A rough estimation is the size of the Cartesian product of all queried properties:

$$numberOfRows(c) := \prod_{p \in c.properties} |p|$$

4.2.4 Structure Planner

As mentioned above, *StructurePlanner* also adopts the same greedy selection strategy described in Algorithm 1. We detail the process of *StructurePlanner* in Algorithm 5. First, we build a lattice over all substructures contained in previous queries P, using the classic lattice construction algorithm (similar to the lattice construction algorithms adopted in *CubePlanner*). Figure 4.3 shows a substructure lattice originating from the root node *Badge-User, User-Post, Post-Tag*. Starting from a union of structures of previous queries as the root node, a lattice can be constructed recursively by populating descendants from parent nodes through edge removals.

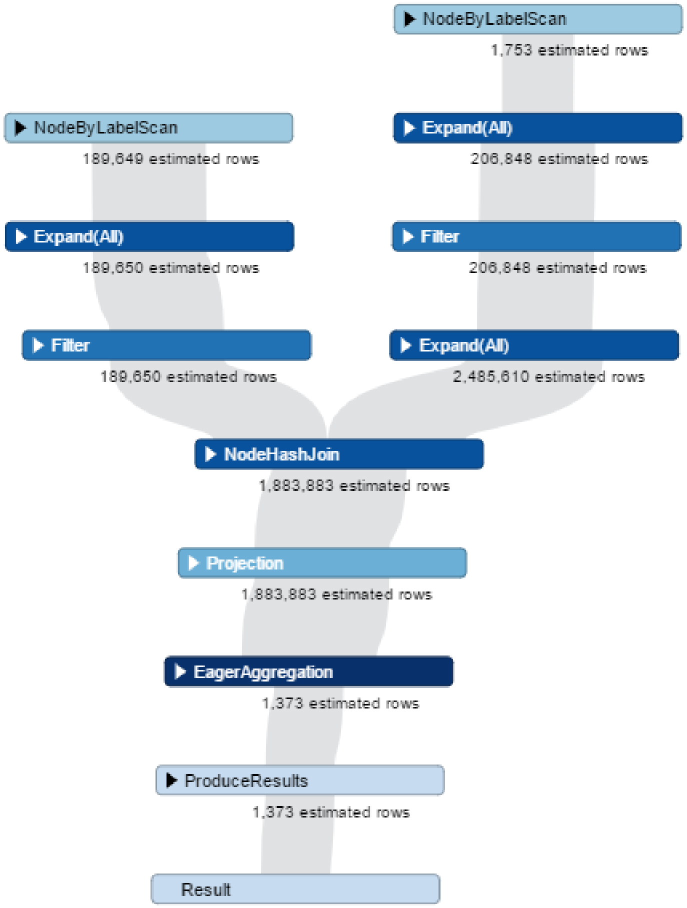

Figure 4.2: Neo4j's execution plan for query *User-Badge, User-Post, Post-Tag: Tag.TagName.*

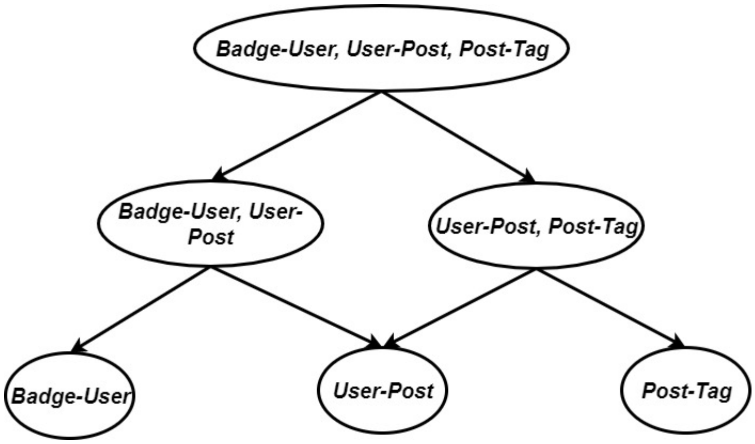

Figure 4.3: A substructure lattice with *Badge-User, User-Post, Post-Tag* as its root node.

Then, we initializes the covered substructures for each previous query as an empty set (line 2-4). For a previous query, *coveredSubstructure* keeps the information on what substructures have been selected so far. It is updated every time a new substructure is selected. Line 5-12 perform score calculation. For each substructure, Line 6 estimates its space. Line 8-10 iterate over all "favored" previous queries (favored by current substructure) and add on the marginal benefit (if any). Here marginal benefit refers to the time saved after adding current substructure to selected substructures (Line 9). Line 13-23 perform the pick-and-update, and line 19-22 updates the covered substructures for previous queries as a result of current round of selection. Such iteration will be terminated when space limit is exceeded (line 13) or when there is no more marginal benefit (line 15).

We now explain the detailed implementation of the estimation functions employed in Algorithm 5. Note that users can have these functions implemented differently on their specific database systems. Function *space(substructure)* returns the estimated space cost of a substructure materialization. We use Neo4j's execution plan API to get an estimated result size of a substructure. Function *benefit(q, substructure, q.coveredSubstructres)* evaluates the marginal benefit of materializing a substructure for Q. We know that execution plan and estimated intermediate result size are provided by Neo4j's API. But such information is on database's naive processing plan. When substructure materialization is used, execution plan (intermediate result) becomes different from naive processing plan. As a result, estimation on marginal benefit of a substructure is tricky. We estimate the marginal benefit of a substructure as follows:

$$time(q.coveredSubstructres \cup substructure) - time(q.coveredSubstructres)$$

Algorithm 5: StructurePlanner

```
   System setting: $\sigma : spacelimitformaterializedviews$
   Input: Q: a set of previous queries
   Output: S: an queue of selected substructures to precompute
1  $Lattice \leftarrow buildSubstuctureLattice(Q)$;
2  foreach $q \in Q$ do
3      $q.coveredSubstructres := \emptyset$;
4  end
5  foreach $substructure \in Lattice$ do
6      $substructure.space \leftarrow space(substructure)$;
7      $substructure.benefit \leftarrow 0$;
8      foreach $q \in Q$ and $q.structure \subseteq substructure.structure$ do
9          $cuboid.benefit+ = max(0, benefit(q, substructure, q.coveredSubstructres))$;
10     end
11     $substructure.score \leftarrow substructure.benefit/substructure.space$;
12 end
13 while $System.memoryUsage < \sigma$ do
14     $nextBestSubstructre \leftarrow$ substructure in Lattice with highest substructure.score;
15     if $nextBestSubstructre.score < 0$ then
16         break;
17     end
18     S.offer(nextBestSubstructre);
19     foreach $q \in Q$ and $q.structure \subseteq nextBestSubstructre.structure$ do
20         $q.coveredSubstructres \leftarrow q.coveredSubstructres \cup \{nextBestSubstructre\}$;
21     end
22     repeat 5-12;
23 end
24
```

, which would serve as an indication on the overall improvement of adding *substructure* to *coveredSubstructres* as materializations.

4.2.5 ID and Property Selection

Given a substructure picked by the *StructurePlanner*, we need to decide on which IDs and properties should be stored. Keeping all IDs and attributes makes a substructure materialization more informative but increases the space cost. Thus, a selection on IDs and properties is an important issue. We will use substructure *User-Post, Post-Tag* as an example and discuss different ID and property selection policies.

For IDs, we consider the following two policies.

- Policy #1 keeps IDs of all nodes and edges. This enables "overlap" join with other substructures but increases space cost. For *User-Post, Post-Tag*, if we keep IDs of all nodes and edges, then we can perform join operation with *Badge-User, User-Post*. We call such join an "overlap" join as the two substructures have an overlap part which is *User-Post*. Note that we can join the two substructures only when IDs of nodes (User and Post), and edge (edge between User and Post) are stored in both substructures.

- Policy #2 only keeps IDs of "border nodes" which are on the border of the substructure's *structure*. Figure 4.4 highlights "border nodes" of structure *User-Post, Post-Tag*. In this example we only save IDs of User and Tag. We do not keep IDs of Post as node Post is not located on the border of the *structure*. Compared to Policy #1, this saves space cost but "overlap join" with other substructures is not enabled. Policy #2 only enables joins on border nodes. For example we may join *User-Post, Post-Tag* with *User-Badge* on their common border node User.

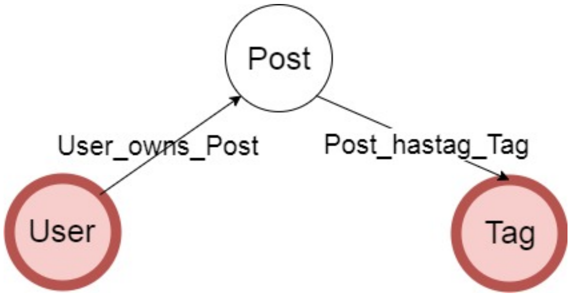

Figure 4.4: "Border nodes" of structure *User-Post, Post-Tag*.

We use Policy #1 in our implementation. However if keeping IDs of inner nodes and edges overwhelmingly increases result length, it's wise to choose Policy #2 as space cost becomes too high.

For properties, we consider the following two policies.

- Policy #1 keeps all properties.
- Policy #2 only keeps properties that were queried in previous workloads.

Our suggestion is to consider the proportion of properties which were queried in previous workload over all properties in the data schema. For example, in our experiment only a small proportion of properties were queried. We choose Policy #2 as it is a waste of space to keep all properties.

4.2.6 Update on Materialized Views

As a matter of fact, the solution we have discussed so far is applicable for static scenarios (with fixed previous workload). Now let's expand our solution to dynamic scenarios: Along the execution of more and more queries there could be change of "hot structures" and "properties of interest". Thus updates on materialized views are necessary. Remember that we have a memory budget therefore in some cases obsolete materialized views need to be swapped out for new ones. To achieve this, we maintain a sliding window over previous queries. Each time after finishing a certain number of queries, we perform materialized view selection over only "recent queries" which are in the sliding window. For eliminated old views that need to be swapped out we simply release their memory. For newly selected views we materialize them. As for survived old views we do nothing as they are already materialized in memory. In this way periodical update on materialized views can be realized.

4.3 Query Processing

Query processing aims at processing in-coming queries efficiently using materialized substructures and cuboids. When a query Q arrives, we first consult cuboids materialization. If Q can be answered with an aggregation over any materialized cuboid, we select the cuboid with the minimum space and directly scan over it to produce result of Q. If Q

cannot be answered by any cuboid, we decompose Q and use substructures as much as possible to compute the result of Q.

Algorithm 6: Query Processing

```
   System: C: a set of materialized cuboids
   S: a set of materialized substructures
   Input: q: a query
   Output: r: result of q
1  $minspace := \infty$;
2  mincuboid := NULL;
3  foreach $cuboid \in C$ do
4      if $cuboid.structure = q.structure$ and $q.dimension \subseteq cuboid.dimension$ then
5          if $cuboid.space < minspace$ then
6              $minspace := cuboid.space$;
7              $mincuboid := cuboid$;
8          end
9      end
10     if $mincuboid \neq NULL$ then
11         $r := aggregate(mincuboid, q)$;
12     else
13         $r := Decompose\_Join(q)$;
14     end
15 end
```

Algorithm 6 describes the generic work flow of query processing. Given an in-coming query q, we first look up materialized cuboids and find if any cuboid can be used to answer q (line 4-9). If there are multiple useful cuboids we select the cuboid with the smallest scanning cost (i.e., the one with the minimum *cuboid.space*). Note that *cuboid.space* was computed in line 9 in Algorithm 4. Then, we check if q can be answered by cuboid materialization. If positive, we perform aggregation operation over the cuboid (line 11). Otherwise, we need to decompose q into substructures and compose the result (line 13). Function $aggregate(mincuboid, q)$ is the classic aggregation operation. We will discuss how function $Decompose\_Join(q)$ is implemented at the end of this section.

4.3.1 Substructure Selection

Before discussion on $Decompose\_Join(q)$, we need to first solve a "Substructure Selection" problem. In order to decompose a query q, we need to consider which materialized substructures we need to use. We need to make decision when candidate substructures in S overlap. For example suppose q has structure *Badge-User, User-Post, Post-Tag.*

And S consists of substructures

(1)Badge-User

(2)Badge-User, User-Post

(3)User-Post, Post-Tag

(4)Post-Tag

(5)User-Post.

We can get structure of q by joining structures of (1) and (3). Thus (1) and (3) seems to be a possible combination for substructure selection in this case. Actually we may have at least three ways of substructure selection: (1) and (3); (2) and (4); (1), (4) and (5). The key question is which selection will result in fastest processing time on q? Here are some intuitions to solve this tricky question. First, when we select substructures one by one, we do not select a substructure when it is covered by selected substructures. For example we will not consider (1) if (2) has been selected as (1) is covered by (2). Second, we prefer to minimize total size of selected substructures as we need to at least access each selected view once. We prefer less memory access. Third, we prefer smaller number of selected substructures as intuitively this causes less times of joins.

We propose a greedy algorithm for substructure selection based on user defined heuristics. Users may define heuristic functions based on intuitions (like the three intuitions mentioned above). The idea of the greedy algorithm is to always pick up next substructure with highest score of user defined heuristic function $h(s)$, which returns heuristic score for a substructure s. Some exampling heuristics are #edges of substructure, score calculated in StructurePlanner (Line 11 in Algorithm 5), table size etc.

```
Algorithm 7: SelectSubstrucre
   System: S: a collection of materialized substructures
   h(s): user defined function. It returns the heuristic score of a substructure $s$.
   Input: q: a future query
   Output: V : selected views for future joining
   uncoveredStruc: structure not covered by selected views
   uncoveredProp: properties not covered by selected views
 1 uncoveredStruc := q.structure;
 2 uncoveredProp:= q.properties;
 3 $coveredStruc := \emptyset$;
 4 $V := \emptyset$;
 5 foreach $s \in S$ ordered by h(s) do
 6     if $s \subseteq uncoveredStruc$ and $s \not\subseteq coveredStruc$ then
 7         $V := V \cup \{s\}$;
 8         $coverdStruc := coveredStruc \cup s.structure$;
 9         uncoveredStruc := uncoveredStruc - s.structure;
10         uncoveredProp := uncoveredProp -s.properties;
11     end
12 end
```

Line 1-2 initializes *uncoveredStruc* and *uncoveredProp*, which keeps track of structures and properties which have not been covered by selected substructures. Such uncovered structures and properties will need to be fetched from database. Line 3 initializes *coveredStruc*, which keeps union of selected substructures. Line 5 starts iteration over substructures ordered by user-defined heuristics $h(s)$. Line 6 assures that a candidate substructure that is totally covered by selected substructures will be disqualified. In the above example, suppose we have already selected (2), there is no need to select (1) since (1) is totally covered by (2).

4.3.2 Decomposition and Join

We have talked about how to select substructure materializations in last subsection. In this part, we will finally discuss how to implement function $Decompose\_Join(q)$ (as in Algorithm FutureQueryProcessing in subsection 4.3). Besides $Decompose\_Join(q)$, we shall discuss two other variations of implementation: $Decompose\_Join^*$ and $Decompose\_Join^+$.

#1 *Decompose_Join*

Given a query q, we use the previously discussed algorithm "SelectSubstrucre" to select a set of substructure materializations V. However, substructures in V may not completely covers the structure of V. If there is any remaining structure (*uncoveredStruc*) and properties (*uncoveredProp*) that V does not cover, we need to retrieve them from database. We call such remaining structure and properties fetched from database "complementary components". After all these components (both materializations and "complementary components") are finally ready, we join and aggregate them together to produce final results.

Algorithm 8: Decompose_Join

```
System: S: a collection of materialized substructures
Input: q: a future query
Output: r: result of q
1 $\Sigma \leftarrow \emptyset$;
2 $V, uncoveredStruc, uncoveredProp \leftarrow SelectSubstrucre(q)$;
3 $\Sigma \leftarrow \Sigma \cup V$;
4 $Splits := split(uncoveredStruc, uncoveredProp)$;
5 foreach $s$: $Splits$ do
6     $\Sigma \leftarrow \Sigma \cup \{retrieve(s)\}$;
7 end
8 $r := join\_aggregate(\Sigma, q)$;
```

Line 1 initializes Σ, which maintains a set of all components (materializations and "complementary components") that are needed. Line 2 selects substructures using SelectSubstructure algorithm. *uncoveredStruc* and *uncoveredProp* refer to structures and properties which are not covered by selected substructures. They are "complementary components" and will be retrieved from database servers. Line 4 splits *uncoveredStruc* and *uncoveredProp* into connected components. We will retrieve each connected component from database server. Note that splitting is necessary since *uncoveredStruc* may not be exactly one connected component. Line 8 joins and aggregates all materials together to produce results.

Function *split(uncoveredStruc, uncoveredProp)* is implemented by classic connected components detection algorithms. It splits *uncoveredStruc* and *uncoveredProp* into connected components (structures). We want to retrieve each connected structure separately from database because otherwise it may result in unnecessarily large results of Cartesian products of several disconnected structures. Function $materialize(s)$ retrieve "complementary components" s from database server. Function *join(*Σ*, q)* join tables of Σ together and aggregate over properties based on q. Joins over multiple tables has been a well-studied topic. Joining order and join technique (hash join etc) are two important aspects on this topic. In our implementation we use hash join and our joining order policy is to keep joining two tables which have minimum sum of table sizes and have common column(s). That is, we tend to select two smaller tables to join.

#2 *Decompose_Join**

Decompose_Join retrieve "complementary components" from database in a naive manner. We adopt the idea of Semi-Join [20] and propose another way of implementation: *Decompose_Join**. Semi-join takes advantage of "selection effect" of natural joins. In *Decompose_Join**, we first perform joins over substructures of V even before fetching "complementary components" from database. Line 3 performs $join(V)$ before *retrieve* in Line 7. We call this phase "first round of joins". Note that substructures in V may reside in multiple connected components. Thus V^* may consist of multiple intermediate tables. The purpose for first round of joins is that it provides "candidate" node and edge IDs for future joins (thanks to 'selection effect" of natural joins). When fetching "complementary components" from database server, we inform database server such candidate node and edge IDs so that search space for "complementary components" is narrowed down ($retrieve^*(s, V^*)$ in Line 7). We name this approach *Decompose_Join**.

Algorithm 9: *Decompose_Join**

```
   System: S: a collection of materialized substructures
   Input: q: a future query
   Output: r: result of q
1  $\Sigma \leftarrow \emptyset$;
2  $V, uncoveredStruc, uncoveredProp \leftarrow SelectSubstrucre(q)$;
3  $V^* := join(V)$;
4  $\Sigma \leftarrow \Sigma \cup V$;
5  $Splits := split(uncoveredStruc, uncoveredProp)$;
6  foreach s: Splits do
7      $\Sigma \leftarrow \Sigma \cup \{retrieve^*(s, V^*)\}$;
8  end
9  $r := join\_aggregate(\Sigma, q)$;
```

We explain implementation of function $retrieve^*(s, V^*)$. It fetches results from databases by passing candidate IDs information (from join result V^*). Syntax to achieve this varies by database. In SQL and Cypher we may pass candidate IDs using a “WHERE” statement. Besides Neo4j driver supports passing lists of integers as arguments in a query.

We compare *Decompose_Join** vs. *Decompose_Join* and summarize following pros and cons of *Decompose_Join**. *Pros*: *Decompose_Join** helps accelerate retrieval process from back end databases in two aspects. First, since screened out candidate IDs are provided, database back end only needs to iterate through a portion of nodes and edges. This saves database processing time. Second, candidate IDs have a “selection” effect thus size of retrieval results is deducted. Thus time caused by result transmission will be reduced. *Cons:* First, *Decompose_Join** has an transmission overhead of IDs. Second, *Decompose_Join* performs one round of joins, after all components are ready. While *Decompose_Join** performs first round of joins on V before “complementary components” are ready and then followed by second round of joins. In terms of optimization on joining orders, *Decompose_Join* is better as its one-round joining order is considered based on all components (with all possible orders for joining).

#3 *Decompose_Join*$^+$

We have mentioned two advantages of $retrieve^*(s, V^*)$. However a disadvantage of $retrieve^*(s, V^*)$ is an overhead of transmission of candidate IDs. We propose a decisive way to evaluate the trade-off between overhead and benefits of $retrieve^*(s, V^*)$ and choose between $retrieve^*(s, V^*)$ and $retrieve(s)$. *Decompose_Join*$^+$ origins from *Decompose_Join**. The trick is to take a second thought in Line 7 by using $decide(s, V^*)$ to choose the better way between $retrieve^*(s, V^*)$ and $retrieve(s)$.

Algorithm 10: *Decompose_Join*$^+$

```
System: S: a collection of materialized substructures
Input: q: a future query
Output: r: result of q
$\Sigma \leftarrow \emptyset$;
$V, uncoveredStruc, uncoveredProp \leftarrow SelectSubstrucre(q)$;
$V^* := join(V)$;
$\Sigma \leftarrow \Sigma \cup V$;
Splits:=split(uncoveredStruc, uncoveredProp);
foreach s: Splits do
    if decide(s, $V^*$) then
        $\Sigma \leftarrow \Sigma \cup \{retrieve^*(s, V^*)\}$;
    else
        $\Sigma \leftarrow \Sigma \cup \{retrieve(s)\}$;
    end
end
$r := join\_aggregate(\Sigma, q)$;
```

Implementation of function $decide(s, V^*)$ is detailed as follows. We first estimate result sizes two retrieval methods: $retrieve^*(s, V^*).estSize$ and $retrieve(s).estSize$. $retrieve(s).estSize$ can be returned by $space(substructure)$ in Algorithm 5. Now the tricky one is $retrieve^*(s, V^*).estSize$. We conduct estimation in the following way:

(1). Randomly sample a small number of candidate IDs.

(2). Do $retrieve^*$ but passing only sampled candidate IDs in (1). We call this a “trial query”. We want to use “trial query” to estimate result length of actual $retrieve^*(s, V^*)$. Since we only pass a small number of IDs, time cost of “trial query” is acceptably small.

(3). Calculate $retrieve^*(s, V^*).estSize$ proportionally using the result of “trial query”,.

With result sizes of the two ways being estimated, we compare $retrieve^*$’s benefit in result size reduction against its cost in passing candidate IDs by evaluating $(retrieve(s).estSize - retrieve^*(s, V^*).estSize)/sizeOf(candidateIDs)$ and compare the ratio with a threshold γ and make a decision. γ is important as it directly affects decision making. We suggest that value of γ should be determined based on understanding of specific database. To make it even better, we may consider running some prior tests in order to adjust γ to a proper value.

We have compared *Decompose_Join** vs. *Decompose_Join*. Now we give comparison on *Decompose_Join*$^+$ vs. *Decompose_Join**: We see that *Decompose_Join*$^+$ performs two rounds of joins like *Decompose_Join**. The major difference is that *Decompose_Join*$^+$ plays "trial query". The principle behind "trial query" is to pay an acceptable price of time cost so that we make wise decision on "complementary components" retrieval. From our experiment, we summarize that a good decision making on "complementary components" retrieval often saves much more time than time cost of "trial queries", especially when a dataset is large.

Chapter 5

Experiments

In this chapter, we valid our proposed solution with a set of meaningful queries over a real world dataset. To demonstrate the time efficiency of our approach, we select Neo4j community version 3.1.3 as the baseline. In the experiment, we test query processing time and space cost by running a set of queries using both our approach and Neo4j implementation. And the results show that our approach is about 20 times faster than the Neo4j implementation on average under the default settings (to be covered in 5.2). Moreover, we assess and explain how each aspect in 5.2 in our system affects processing efficiency. Last but not the least, we discuss our reflection on the Neo4j system.

5.1 Experiment Setup

There are two purposes of our experiments. First, it is to demonstrate the query processing efficiency of our proposed solution by comparing against the native Neo4j system. Second, we would like to test how different settings (of parameters and choices of strategies) in our solution affect query processing efficiency. Experiments are run on a set of self-designed meaningful queries on a real-world dataset.

5.1.1 Datasets

The StackOverFlow dataset used in our experiments is generated by using raw information from https://archive.org/details/stackexchange. The dataset contains user-contributed content (such as user information, posts and etc.) on www.stackoverflow.com. The dataset contains 10 different node labels and 12 different edge labels. Figure 5.1 shows the meta graph of 8 node labels and 8 edge labels which are involved in our experiments. The data graph overall contains over 300 million nodes and more than 400 million edges. Its size on hard disk is 44.5GB.

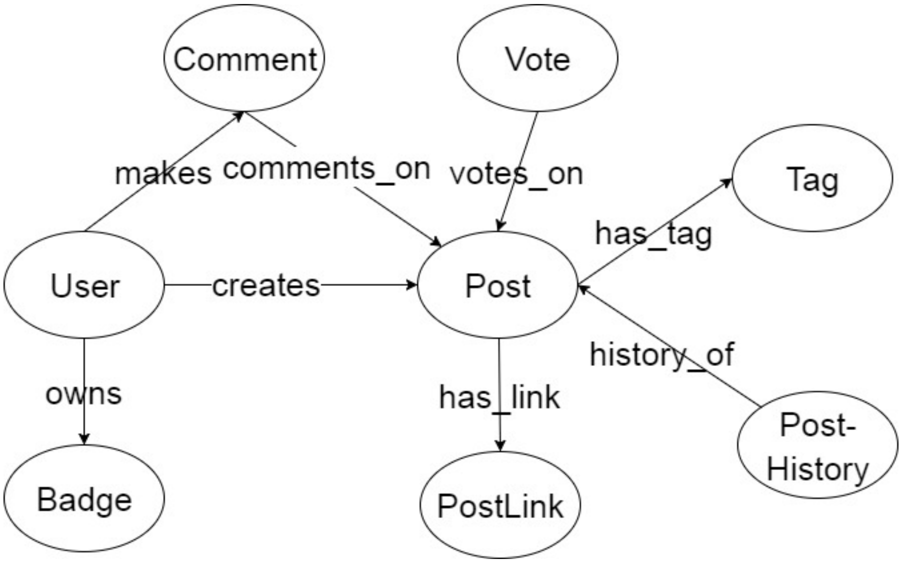

Figure 5.1: The meta graph of StackOverFlow used in experiments.

5.1.2 Query Workloads

We design 24 meaningful queries against the StackOverFlow dataset. We randomly select 12 queries as the previous workload, leaving the rest 12 ones as the future workload. For completeness, we include all the 24 queries as follows.

Previous WorkLoad:

P1 User-Post: User.UpVotes, Post.Score=10

P2 User-Post: User.UpVotes, (AVG)Post.Score

P3 User-Post: User.Age, (SUM)Post.ActiveMonth

P4 User-Post: User.CreationDate_Year, Post.PostTypeId=1

P5 Badge-User, User-Post, Post-Tag: Tag.TagName, User.CreationDate_Year=2017

P6 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Name

P7 Badge-User, User-Post:Badge.Date_Year, (AVG)Post.Score

P8 Badge-User, User-Post:Badge.Class, (AVG)Post.ActiveMonth

P9 User-Post, Post-Tag: (AVG)User.Age, Tag.TagName

P10 User-Post, Post-Tag: (AVG)User.UpVotes, Tag.TagName=Java

P11 User-Post, Post-Vote: (AVG)User.UpVotes, Vote.VoteTypeId

P12 Post-Comment, Post-PostLink: PostLink-LinkTypeId, (AVG)Comment-Score

Future WorkLoad:

Q1 User-Post: User.CreationDate_Year=2017, Post.PostTypeId

Q2 User-Post: (AVG)User.UpVotes,Post.Score

Q3 User-Post: Post.ActiveMonth, (AVG)User.Age

Q4 User-Post: User.CreationDate_Year

Q5 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Class

Q6 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Date_Year

Q7 Badge-User, User-Post:Badge.Name, Post.PostTypeId

Q8 User-Post, Post-Tag: User.UpVotes, Tag.TagName, Post.PostTypeId=2

Q9 User-Post, Post-Tag:User.CreationDate_Year, Tag.TagName

Q10 Badge-User, User-Comment: Badge-Class, (AVG)Comment-Score

Q11 Badge-User, User-Comment: Badge-Name, (AVG)Comment-Score

Q12 Post-PostHistory, Post-Tag: Tag-TagName, PostHistory-PostHistoryTypeId

5.1.3 System Setting

We run all the experiments on a Linux server with 256GB main memory. Our system is implemented in Java. We set the initial JVM memory as 100GB, and topped by 200GB.

Our solution is developed on top of Neo4j Community v3.1.3. In the experiment, we set Neo4j's initial memory as 60GB and 200GB for the maximum usage. We use Neo4j's official Java driver (https://neo4j.com/developer/java/neo4j-java-driver) to interact with Neo4j server.

5.2 Aspects of Interest

As we mentioned, there are two purposes of our experiments. In addition to the efficiency test, we would like to study how different settings in our system could affect query processing efficiency. We list the following aspects of interests to be tested. Default setting for each aspect is also presented. For variable control purpose, in each test all other variables are set to default settings.

Materialization

- Space cost limit, i.e. σ in Algorithm 5 in Section 4.2.4, 6GB by default. Results and discussion are presented in 5.3.3.

- Algorithms in materialized view selection described in 4.2. For cuboid selection, we compare CubePlanner (in 4.2.3) with the "Partial Materialization" algorithm (PMA for short) proposed in Graph Cube [25]. For substructure selection we will compare StructurePlanner (in 4.2.4) with the well-known frequent pattern mining algorithm (FPM for short). In other tests, CubePlanner and StructurePlanner are used by default. Results are discussed in Section **??** and Section **??**, respectively.

- Frequency threshold for identification of hot structures, i.e. ω in Algorithm 1. We set the default value to be 4, while the intuition of doing so, together with results and discussion are elaborated in Section 5.3.2.

- Storage level for materialized views. We will compare main memory storage v.s. hard disk storage. Note that memory based materialization is set as the default in all other tests. Detailed discussion is provided in Section 5.3.4.

Future Query Processing

- Choice of score functions in ranking substructures during the "Substructure Selection" (discussed in 4.3.1), i.e. $h(s)$ in Algorithm 7. By default, $h(s)$ returns the score of s calculated in StructurePlanner (as result of line 11 in Algorithm 5). Results and explanations can be found in Section **??**.

- Choice among using *Decompose_Join*, *Decompose_Join** and *Decompose_Join*$^+$ in "Decomposition and Join" 4.3.2. We use *Decompose_Join* as the default method. For detailed discussion please refer to Section **??**.

5.3 Results and Discussion

Following the interests of study listed above, we now present our experimental results.

5.3.1 Our System vs. Neo4j

We first show the time efficiency of our solution running in default settings against the native Neo4j implementation for the future workload.

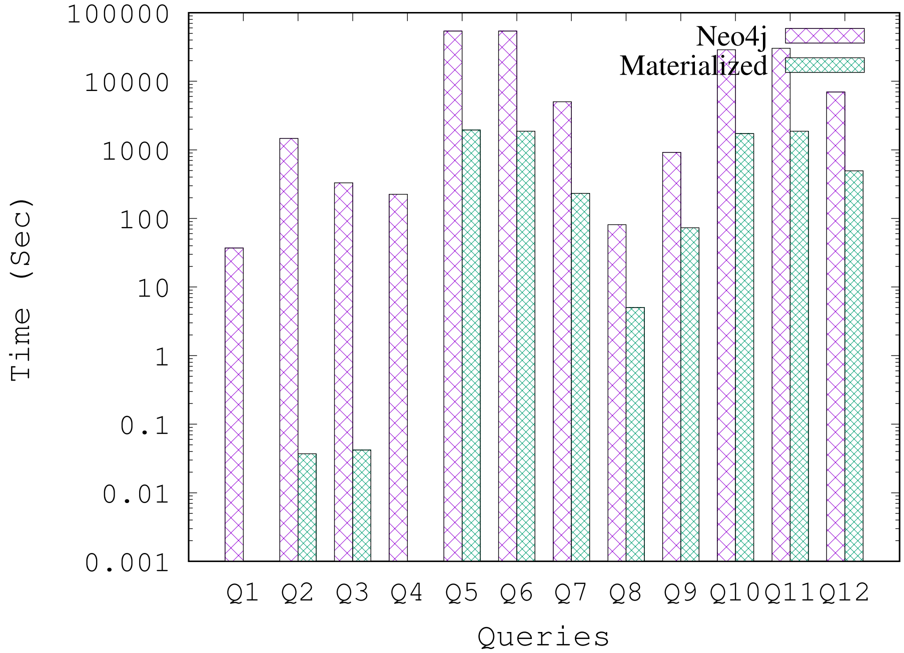

Figure 5.2: Time efficiency on the future workload: our solution v.s. Neo4j.

Figure 5.2 shows the processing time for 12 future queries by both our system and Neo4j. It worth pointing out that with an extra 5.7GB space cost for materialized views, which is acceptable considering the size of the dataset, our solution achieves remarkable efficiency improvement.

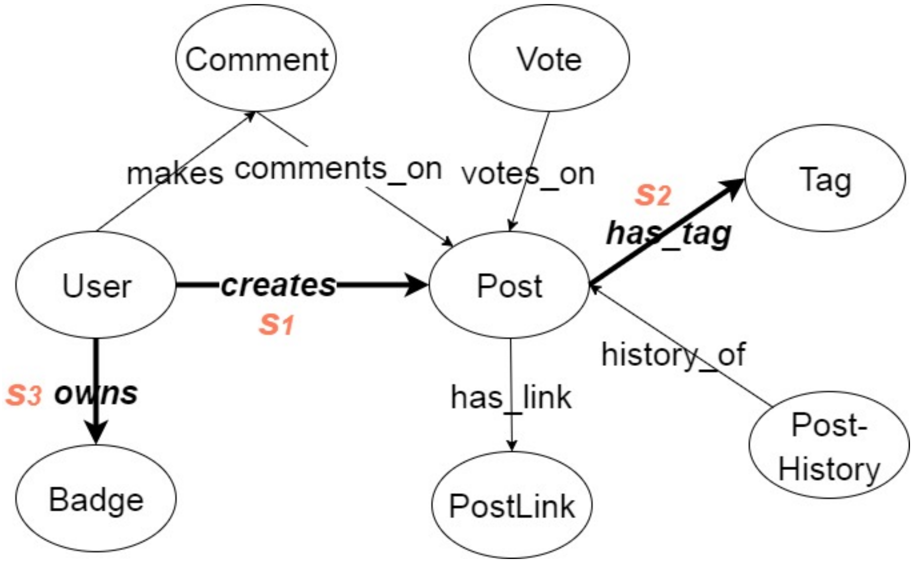

Figure 5.3: Substructure selected by StructurePlanner.

We detail how our system works. First, *User-Post* is identified as a "hot structure" as its frequency count is larger than or equal to the frequency count threshold. As a result, P1 - P4 are passed to the CubePlanner and P5 - P12 are passed to the StructurePlanner. Cuboids selected by CubePlanner are {User.Age, Post.ActiveMonth}, {User.CreationDate_Year, Post.PostTypeId}, and {User.UpVotes, Post.Score}. Figure 5.3 highlights substructures S that StructurePlanner selects: *User-Post* (s_1), *Post-Tag* (s_2) and *Badge-User* (s_3). From the observation of the previous workload, we can tell that the StructurePlanner makes a good decision as these three substructures are able to cover most of previous queries.

We further explain the performance variance of the future workload. Overall improvement rate for Q1 - Q4 is more than 20000. This is because Q1 - Q4 are answered with a "cuboid hit". As a result, the time saving for these 4 queries are much great than for Q5 - Q12 (which do not get a "cuboid hit"). It worth pointing out that time complexities of cuboid aggregation and substructure joins are much different. The time complexity of a cuboid aggregation is bounded by the size of the Cartesian product of its involving properties. When it comes to substructure joins, however, the time complexity is related to the actual data sizes. Note that Q5 - Q9 are totally covered by S. In these cases, no "complementary components" are fetched from the Neo4j database. Q10 - Q12 are partially covered by S. Therefore, "complementary components" are fetched from the database, where a series of I/O swapping increases the total time cost. Overall, the improvement rate for Q5 - Q9 is around 28, while improvement rate for Q10 - Q12 ranges around 15. Clearly, our system could greatly improve query processing efficiency with an acceptable extra space cost. While the improvement ratio varies in different scenarios of cuboid and structure "hits".

5.3.2 Frequency Threshold

Frequency threshold ω can significantly affect the query processing efficiency. A change of ω may result in different "hot structures", followed by varied inputs for the CubePlanner and the StructurePlanner. It would further lead to different materialized view selection and overall query processing performance. As indicated in Section 4.2.1, ω serves as a minimum bar to convince that building a cuboid over a "hot structures" is worthy ("cuboid hit" assured). We think 4 is an appropriate choice for ω in our test case considering the frequency count of structures in previous workload (as shown in the table below).

Structure	Frequency
User-Post	**4**
Badge-User, User-Post, Post-Tag	2
Badge-User, User-Post	2
User-Post, Post-Vote	2
Badge-User, User-Comment	1
Post-Comment, Post-Vote	1

Suppose we lower ω to 2. *Badge-User, User-Post, Post-Tag* will be "hot structure". As a result, cuboid selection over *Badge-User, User-Post, Post-Tag* will be considered based on merely two queries (P5 and P6). Notice that Q5 and Q6, which have structure *Badge-User, User-Post, Post-Tag*, will never get "cuboid hit". This is because properties Badge-Class in Q5 and Badge-Date_Year in Q6 did not even appear in P5 and P6. That is to say, in this case any cuboid selected over *Badge-User, User-Post, Post-Tag* will be useless. In addition, when ω is set to 2, input for StructurePlanner will be only two queries (Q11 and Q12). Note that structure frequency counts of these two queries are both 1. In other words, these are the most "random" queries. Note that the idea of StructurePlanner is to discover of useful substructures based on a sufficient number of "less hot" queries. In this case, two "random" queries are not the ideal input for the StructurePlanner.

On the other side, suppose we increase ω to 5, then there will be no cuboid materialized in our test case. As a result Q1 - Q4 will be processed using substructure materialization

(s_1). Although it is still faster than the native Neo4j system, the outstanding improvement ratio of "cuboid hit" cannot be achieved.

Figure 5.4 presents the total processing time under different settings of ω. Note that no structure has frequency count of 3, therefore setting ω as 3 and 4 would categorize the same set of "hot structures and thus yield the same materialized views. We reach our conclusion that ω does have a significant effect over the system performance. Actually, determination on the value of ω is an interesting classification problem (based on the frequency count). However we will leave this topic to future work since it is not the main focus of this thesis.

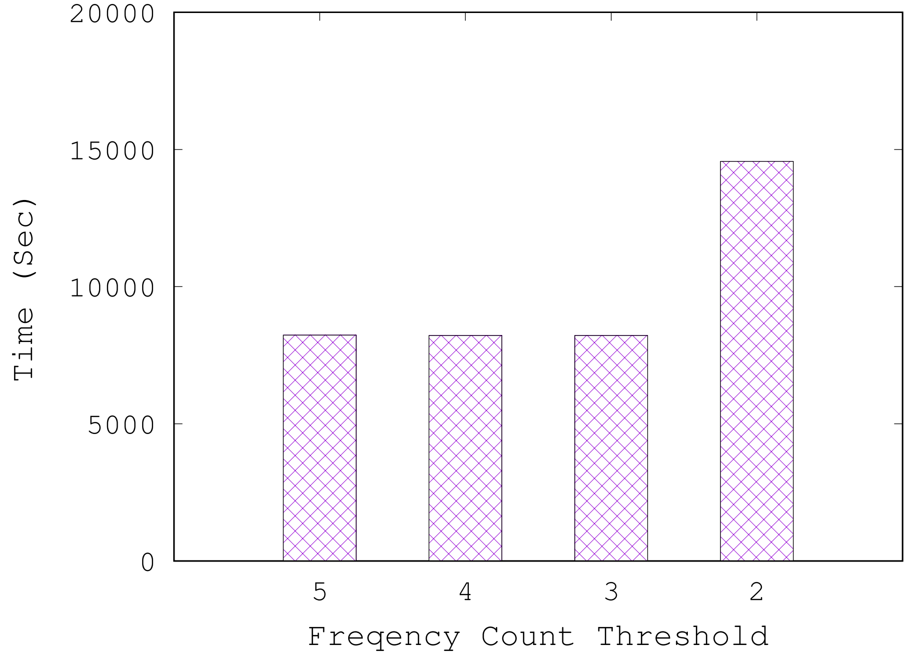

Figure 5.4: Total processing time under different settings of ω.

5.3.3 Space Cost Limit

As pointed out in Section 5.3.1, the space cost of our materialized views is 5.7GB, while the default space cost, σ, is set to 6GB. In this section, we study the effect of σ on the query processing efficiency. Figure 5.5 shows how the total processing time varies with different space costs (which were caused by setting σ to 6GB, 4GB, and 2.5GB, respectively). Apparently, with more views being materialized, the total processing decreases. However, the marginal benefit from materializing more views decreases. It indicates that our Greedy

Selection Framework (in Section 4.2.2) successfully picks the proper candidates according to marginal benefits they would bring.

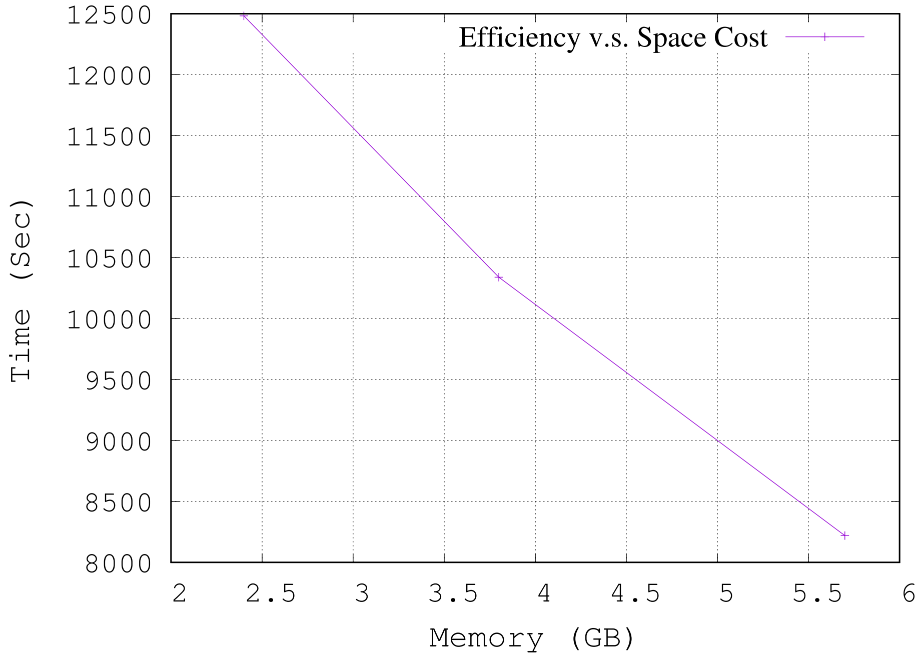

Figure 5.5: Efficiency v.s. Space Cost

5.3.4 Storage Level for Materialized Views

As listed in 5.2, by default, materialized views are stored as objects in main memory. It guarantees fast data access on the cost of extra memory consumption. Alternatively, materialized views can be serialized and stored as files on hard disks. Figure **??** shows a comparison of the total query processing time using memory-based views and disk-based views. As expected,hard disk storage does not perform as fast as main memory storage. However, it worth noticing that the drop in efficiency is acceptable. Such a drop in efficiency is owing to the I/0 overhead in loading materialized views from disk files.

One interesting question is that since eventually materialized views are to be read into the main memory, what is the point of storing them in files Our answer is that in cases when main memory is far too small for holding all selected views, hard disk materialization provides a lower level of storage of sufficient volume.

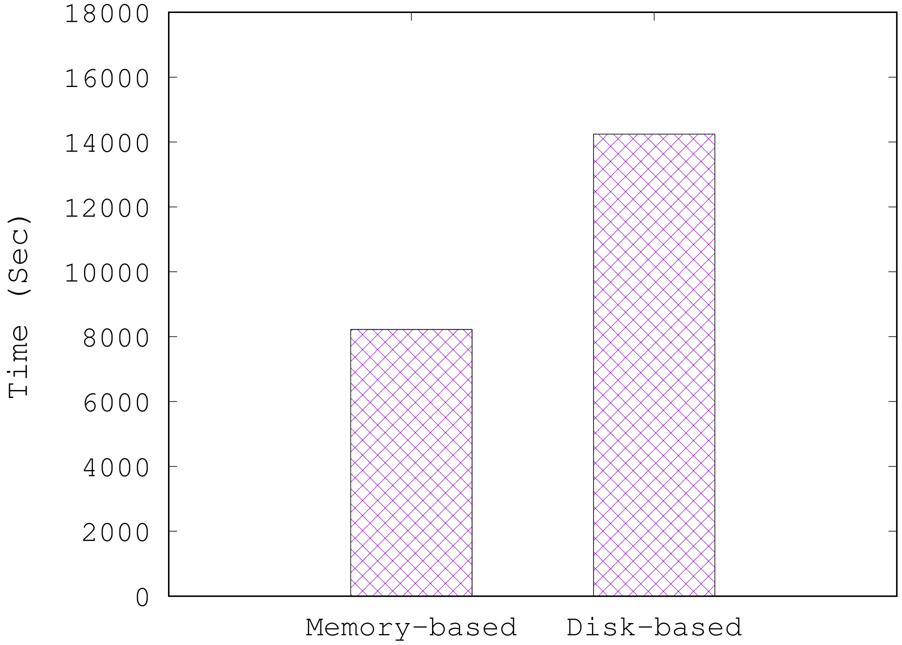

Figure 5.6: Main memory storage v.s. hard disk storage

5.3.5 CubePlanner v.s. PMA

We compare CubePlanner 4.2.3 in our solution with PMA in Graph Cube [25]. Figure **??** and **??** show that CubePlanner outperforms PMA in both the query processing efficiency and the space cost. Cuboids selected by CubePlanner are {User.Age, Post.ActiveMonth}, { User.CreationDate_Year, Post.PostTypeId, and {User.UpVotes, Post.Score}. While PMA selected {User.Age, User.UpVotes, User.CreationDate_Year, Post.PostTypeId, Post.Score, Post.ActiveMonth}, {User.Age, User.UpVotes, User.CreationDate_Year, Post.PostTypeId, Post.ActiveMonth}, {User.Age, User.UpVotes, User.CreationDate_Year, Post.PostTypeId, Post.Score}.

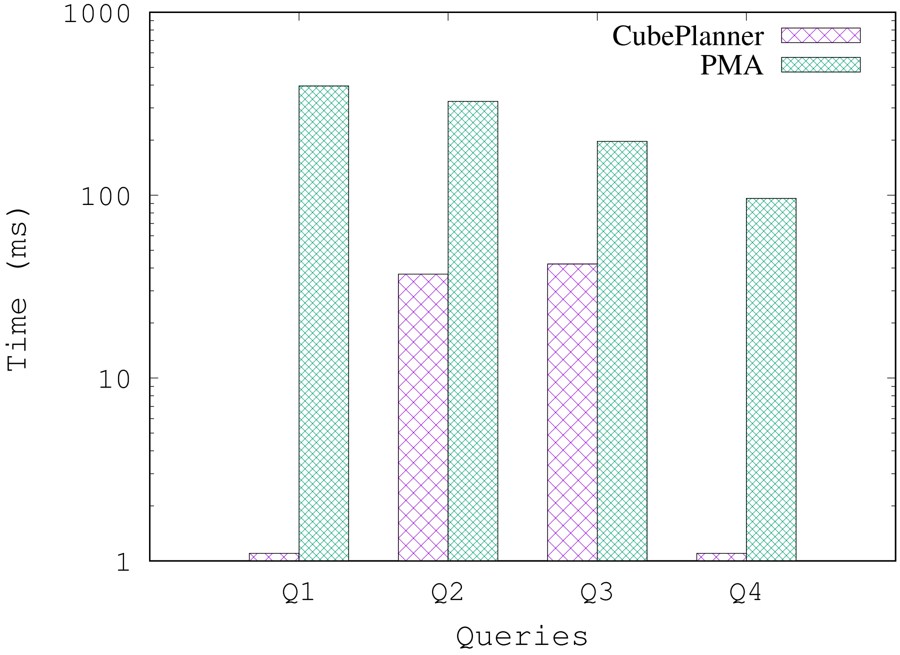

Figure 5.7: Time: CubePlanner v.s. PMA

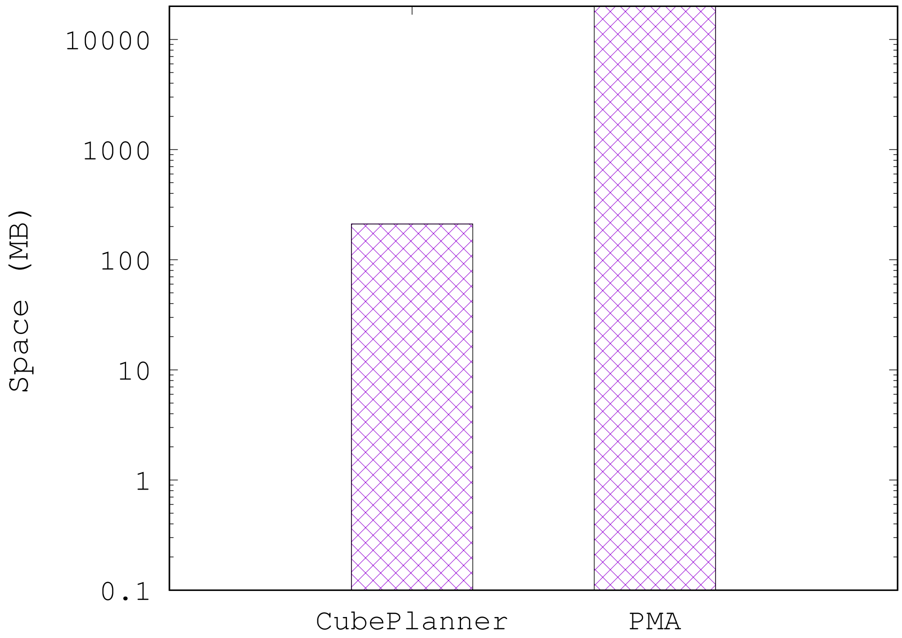

Figure 5.8: Total cuboid space cost: CubePlanner v.s. PMA

Actually both two approaches are considered as implementations of the Greedy Selection Framework given in 4.2.2. We reveal differences between the two implementations. CubePlanner uses the ratio of marginal benefit over space cost as a score for candidate ranking (line 11 in Algorithm 4). While PMA only considers marginal benefit. That is to say, space cost is not taken into account in PMA. Moreover, PMA treats each combination of properties with an equal weight, regardless of how many times a combination appeared in previous queries. For example, in our test case, the combination of User.UpVotes, Post.Score appears twice in previous workload. But PMA would treat User.UpVotes, Post.Score with the same weight as those combinations are not even queried in previous workload ({User.UpVotes, Post.PostTypeId} etc). As a result, CubePlanner adopts more information from previous workload and thus makes a better selection.

5.3.6 StructurePlanner v.s. FPM

We compare our algorithm 5 in StructurePlanner with FPM. In FPM we set the minimum support to 2 considering the frequency count as listed in the table in Section 5.3.2. These two ways provide different substructure selections which lead to different processing efficiency. Figure **??** and **??** show that our StructurePlanner outperforms FPM in both efficiency and space cost.

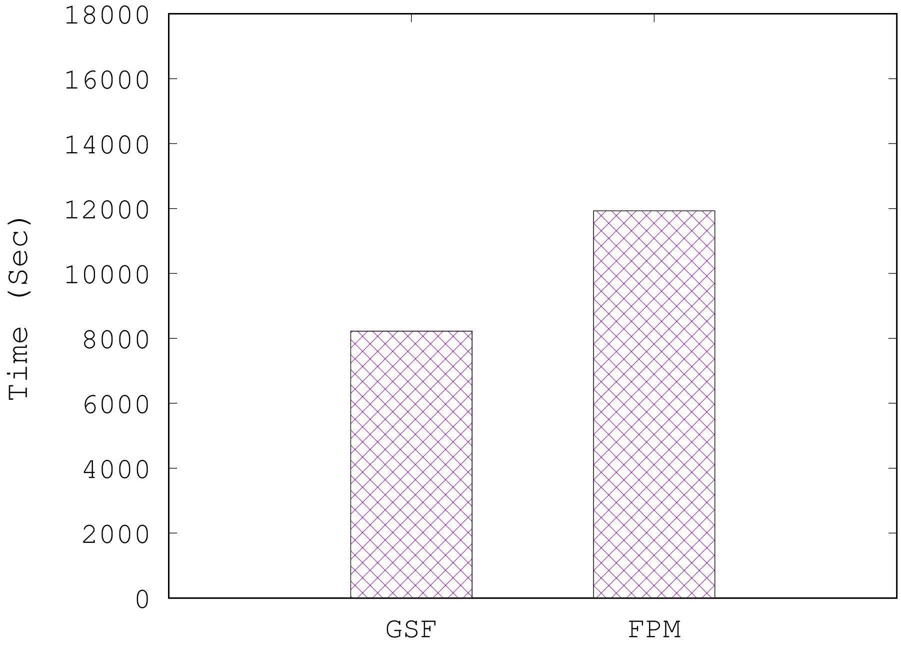

Figure 5.9: Total processing time for future workload: StructurePlanner v.s. FPM

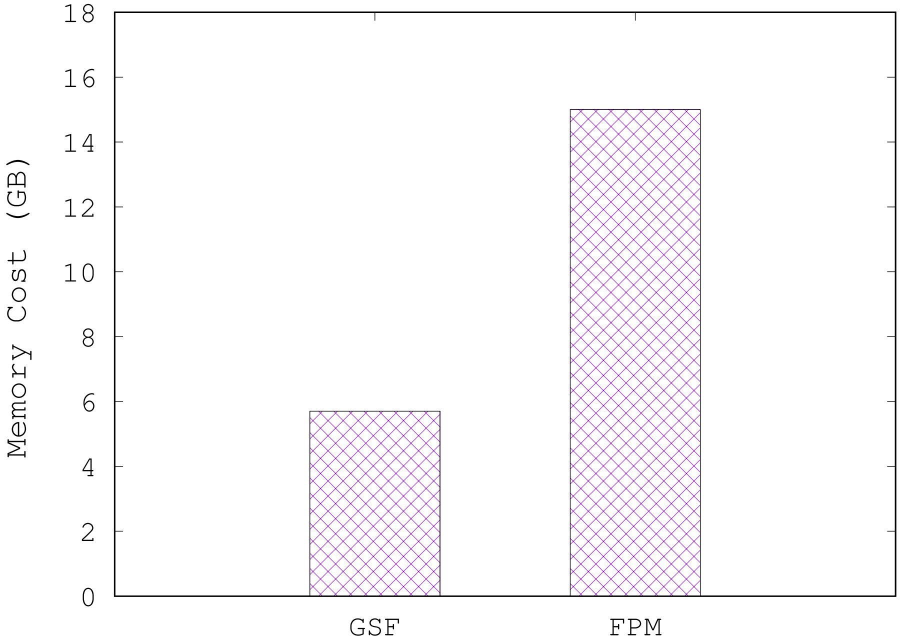

Figure 5.10: Space cost: StructurePlanner v.s. FPM

Figure 5.3 highlights three selected substructures by StructurePlanner. As mentioned, it is a good selection as these three substructures are able to cover most of previous queries. However, FPM selects *Badge-User, User-Post, Post-Tag.* It is a bad selection because it is useful only for Q5 and Q6. In addition, materialization of *Badge-User, User-Post, Post-Tag* results in even more space cost than materialization of the three edges separately (as selected by StructurePlanner). Figure **??** details processing time for each query. FPM only outperforms StructurePlanner on Q5 and Q6. This is because Q5 and Q6 would be able to perform aggregation over the materialization of *Badge-User, User-Post, Post-Tag* when FPM is applied. While table joins of s_1, s_2 and s_3 are required if StructurePlanner is applied, which clearly is more time consuming. However for Q7 - Q12, StructurePlanner is the winner as it gets at least a partial “substructure cover”, while the FPM-selected structure is not helpful at all.

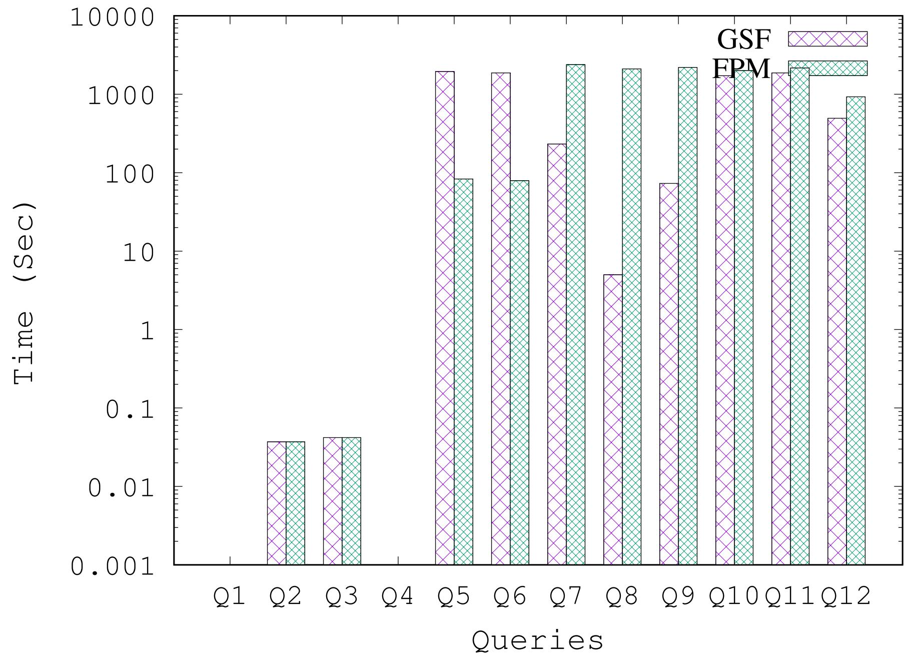

Figure 5.11: Processing time for each query: StructurePlanner v.s. FPM

5.3.7 Substructure Selection

In our experiment, $h(s)$ in Algorithm 7 does not make any difference during "Substructure Selection" 4.3.1. This is because the three selected substructures do not share any common edge. Note that scenarios in Section 4.3.1 where multiple valid combinations of materialized substructures exist only happen when materialized substructures have overlaps.

5.3.8 Decompose_Join

We now present the experiments comparing on *Decompose_Join*, *Decompose_Join** and *Decompose_Join*$^+$ in "Decomposition and Join" (presented in 4.3.2). Such three different implementations in "Decomposition and Join" would lead to different processing efficiency for Q10 - Q12, as they are partially covered by S and fetching "complementary components" from Neo4j is necessary. Figure **??** provides the processing time for Q10 - Q12 using the three approaches respectively. *Decompose_Join** performs better than *Decompose_Join* in Q10. This is because *Decompose_Join** passes candidate IDs of users who have badges to Neo4j, which provides a considerable "filtering effect" when fetching *User-Comment*. As a result, *Decompose_Join**'s processing time for fetching *User-Comment* is reduced. Besides, its time for joining *Badge-User* and *User-Comment* also decreases because the table size of *User-Comment* is smaller than that in *Decompose_Join*, thanks to the "filtering effect". This is reflected in Figure **??**, where the time for join is saved in *Decompose_Join**. However *Decompose_Join** performs badly for Q12. This is because the "filtering effect" of *Post-Tag* in fetching *Post-PostHistory* is small as most posts have tags. In addition, *Decompose_Join** has an overhead of scanning the *Post-Tag* table in order to get the set of candidate IDs. It explains why *Decompose_Join** takes longer time in processing Q12. Figure **??** gives the total processing time for Q10 - Q12 using the three approaches. We see that *Decompose_Join*$^+$ has best overall performance. To explain, *Decompose_Join*$^+$ is able to choose the faster approach when fetching "complementary components" from Neo4j in scenarios like Q10 and Q12 with the cost of a cheap trial query (as discussed in 4.3.2). Note that the time cost for a "trial query" is bounded by a constant sampling size (which is set to 100 in our experiments). It is not proportional to actual data size in the dataset. Figure **??** shows that the time cost "trial query" is negligible comparing to the overall processing time.

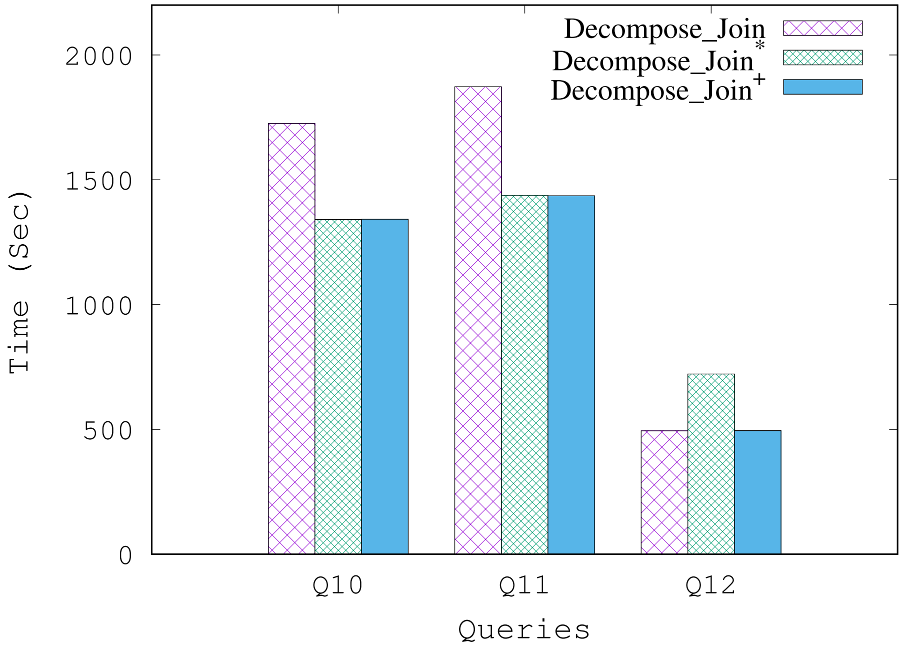

Figure 5.12: Processing time for Q10 - Q12 by three approaches.

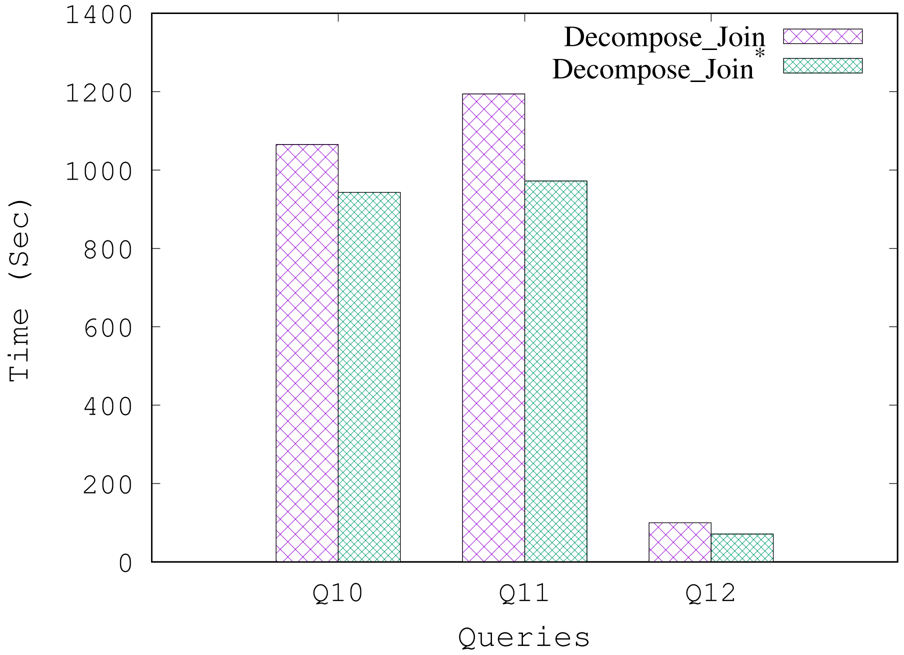

Figure 5.13: Joining time in processing Q10 - Q12 by three approaches.

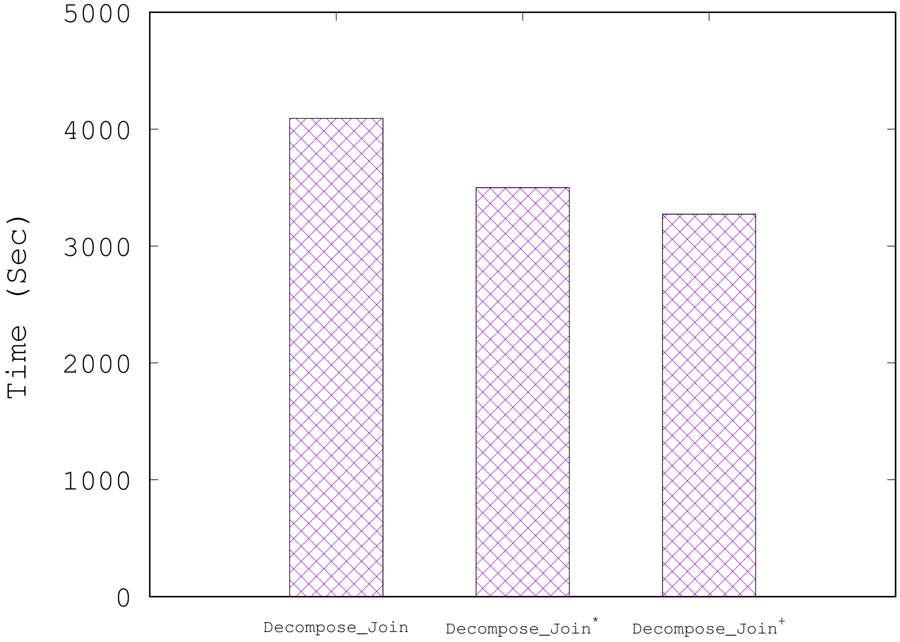

Figure 5.14: Total processing time for Q10 - Q12 by three approaches.

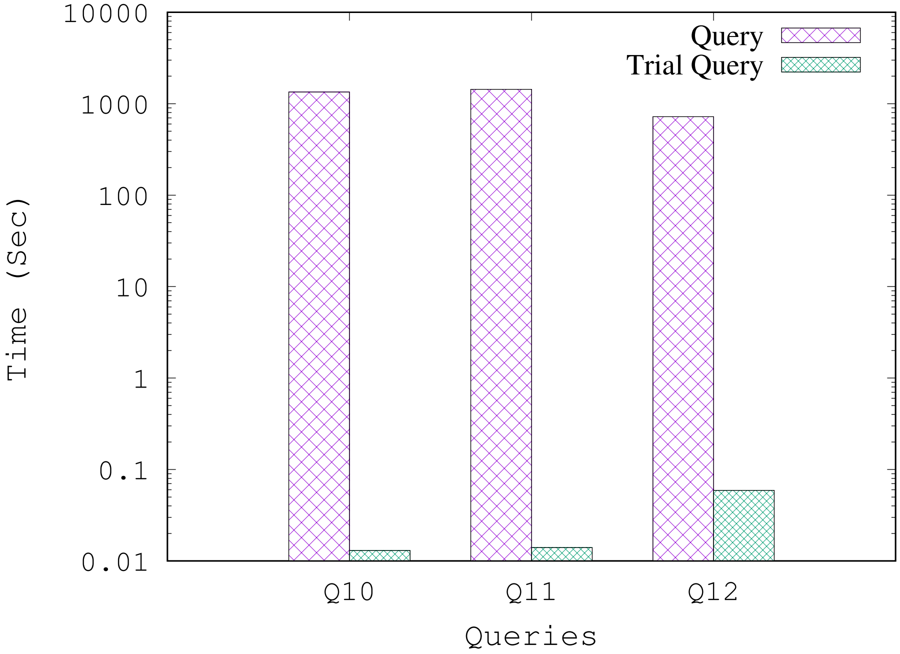

Figure 5.15: Total processing time v.s. "trial query" processing time.

To conclude, "filtering effect" is an important factor in performance of these three different implementations of "Decomposition and Join". In general, *Decompose_Join*$^+$ is the recommended approach as it is able to select the better solution with a small cost of "trial query".

5.3.9 Reflection on Neo4j

During the experiments, we found that Neo4j uses a naïve way of result size estimation for aggregation queries. It simply takes the square root of table length before aggregation as the estimated size for the aggregated result, regardless of which properties are being aggregated. Such a method leads to a huge bias in the estimation.

For example, Figure **??** and figure **??** are Neo4j's execution plans for queries *User-Post: User.Age* and *User-Post: ID(User), ID(Post).* Obviously the latter query should have much larger estimated result size than the former one. However Neo4j returns the exactly same estimated result size by simply taking the square root of table length (216791 estimated rows in "Projection" step) in the previous step.

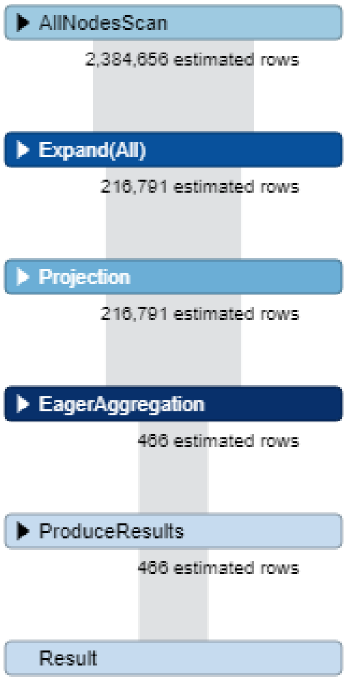

Figure 5.16: Execution plan for *User-Post: User.Age*.

This gives reason to why in “Single CubePlanner” (covered in 4.2.3), we use the Cartesian product of involved properties to estimate cuboid sizes instead of directly using Neo4j’s estimation.

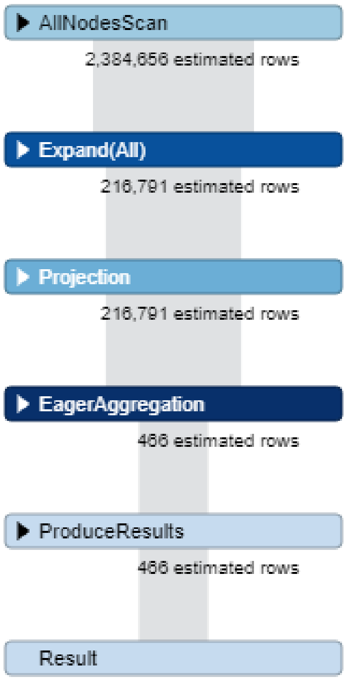

Figure 5.17: Execution plan for *User-Post: ID(User), ID(Post)*.

Chapter 6

Conclusion

Our work addresses on the urgent need for efficient OLAP processing over large property graphs. To model the problem mathematically, we provided formal definitions of “Materialization Selection”, “Execution Planning”, “Decomposition Problem” and “Composition Problem”. We proposed an end-to-end system to tackle these problems and implemented the system for Neo4j. The main idea of our solution is to accelerate future query processing with materialized views which were selected based on their benefits to previous workload. We conducted experiments on our system and it was proved to be a feasible solution that achieves efficient OLAP queries processing on large graph datasets.

We summarize future work as follows. First, as reflected in experiment results 5.3, efficiency performance of our system is affected by system settings, hence study on appropriate system settings would be a valuable follow-up for our work. Second, the system we implemented so far supports SPARQL like queries over schema graph, instead of data graph. It is realizable to enable processing SPARQL like queries over data graph within our proposed solution framework. The key issue is take isomorphism into consideration during query decomposition and composition. Last but not least, greedy selection framework is with no doubt not a perfect solution for “Materialization Selection” problem. We look forward to better solutions for this hard but valuable problem.

References

[1] The internet of things. *Commun. ACM*, 60(5):18–19, 2017.

[2] Kevin S. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 359–370, 1999.

[3] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43, 1998.

[4] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph OLAP: towards online analytical processing on graphs. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pages 103–112, 2008.

[5] Tsan-Ming Choi, Hing Kai Chan, and Xiaohang Yue. Recent development in big data analytics for business operations and risk management. *IEEE Trans. Cybernetics*, 47(1):81–92, 2017.

[6] Alfredo Cuzzocrea and Carson Kai-Sang Leung. Efficiently compressing OLAP data cubes via r-tree based recursive partitions. In *Foundations of Intelligent Systems - 20th International Symposium, ISMIS 2012, Macau, China, December 4-7, 2012. Proceedings*, pages 455–465, 2012.

[7] Grzegorz Drzadzewski and Frank Wm. Tompa. Partial materialization for online analytical processing over multi-tagged document collections. *Knowl. Inf. Syst.*, 47(3):697–732, 2016.

[8] Shifeng Fang, Li Da Xu, Yunqiang Zhu, Jiaerheng Ahati, Huan Pei, Jianwu Yan, and Zhihui Liu. An integrated system for regional environmental monitoring and management based on internet of things. *IEEE Trans. Industrial Informatics*, 10(2):1596–1605, 2014.

[9] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[10] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[11] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 58:1–58:12, 2015.

[12] Wararat Jakawat, Cécile Favre, and Sabine Loudcher. OLAP cube-based graph approach for bibliographic data. In *Proceedings of Student Research Forum Papers and Posters at SOFSEM 2016 co-located with 42nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2016), Harrachov, Czech Republic, January 23-28, 2016.*, pages 87–99, 2016.

[13] Donald Knuth. *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[14] Wing-Kit Sunny Lam, Tony R. Sahama, and Randike Gajanayake. Constructing a traditional chinese medicine data warehouse application. *CoRR*, abs/1606.02507, 2016.

[15] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[16] Michael Lawrence and Andrew Rau-Chaplin. Dynamic view selection for OLAP. In *Strategic Advancements in Utilizing Data Mining and Warehousing Technologies: New Concepts and Developments*, pages 91–106. 2010.

[17] Wen-Yang Lin and I-Chung Kuo. A genetic selection algorithm for OLAP data cubes. *Knowl. Inf. Syst.*, 6(1):83–102, 2004.

[18] Fadi Maali, Stéphane Campinas, and Stefan Decker. Gagg: A graph aggregation operator. In *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, pages 491–504, 2015.

[19] Lhouari Nourine and Olivier Raynaud. A fast algorithm for building lattices. *Inf. Process. Lett.*, 71(5-6):199–204, 1999.

[20] Z. Meral Özsoyoglu. Review - using semi-joins to solve relational queries. *ACM SIGMOD Digital Review*, 1, 1999.

[21] André Petermann, Martin Junghanns, Robert Müller, and Erhard Rahm. Graph-based data integration and business intelligence with biiig. *Proc. VLDB Endow.*, 7(13):1577–1580, August 2014.

[22] Arun N. Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, pages 287–300, 1994.

[23] Zhengkui Wang, Qi Fan, Huiju Wang, Kian-Lee Tan, Divyakant Agrawal, and Amr El Abbadi. Pagrol: Parallel graph olap over large-scale attributed graphs. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 496–507, 2014.

[24] Jim Webber. A programmatic introduction to neo4j. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 217–218, 2012.

[25] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and OLAP multidimensional networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 853–864, 2011.

[26] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 159–170, 1997.

APPENDICES

Appendix A

Meaning for Each Query in Our Experiments

A.1 Previous Workload

Meaning for each previous query is listed below.

P1 User-Post: User.UpVotes, Post.Score=9

To see the distribution of users' upvotes for high score (score=9) posts.

P2 User-Post: User.UpVotes, (AVG)Post.Score

To see average post scores by different upvotes.

P3 User-Post: User.Age, (SUM)Post.ActiveMonth

To see each age group's contribution to total posts' active time. What is the main stream users' age range in stackoverflow.com?

P4 User-Post: User.CreationDate_Year, Post.PostTypeId=1

To see numbers of questions (Post.PostTypeId=1) posted by different years when joining the forum.

P5 Badge-User, User-Post, Post-Tag: Tag.TagName, User.CreationDate_Year=2017

In 2017, how many badges are "involved" in each tag?

P6 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Name

For each tag, what is the distribution of different types of “involved” badges?

P7 Badge-User, User-Post:Badge.Date_Year, (AVG)Post.Score

How average post score varies by years that badges are honored? Has the “value” of badges changed by time?

P8 Badge-User, User-Post:Badge.Class, (AVG)Post.ActiveMonth

Does high class badge indicate long post active month?

P9 User-Post, Post-Tag: (AVG)User.Age, Tag.TagName

Which topics are trendy among youngster users and which ones are popular among middle-aged users?

P10 User-Post, Post-Tag: (AVG)User.UpVotes, Tag.TagName=Java

What’s the average upvotes (weighted) for users who involve in tag “Java”.

P11 User-Post, Post-Vote: (AVG)User.UpVotes, Vote.VoteTypeId

For different type of votes, is there a difference in the voters average upvotes?

P12 Post-Comment, Post-PostLink: PostLink-LinkTypeId, (AVG)Comment-Score

Is there a connection between post’s link type and post’s average comment score?

A.2 Future Workload

Intuition for asking each future query is listed as follows.

Q1 User-Post: User.CreationDate_Year=2017, Post.PostTypeId

For users who join recently in year 2017, how many posts are posted for each type of posts? How many are questions and answers?

Q2 User-Post: (AVG)User.UpVotes,Post.Score

What is the average users’ upvotes for each post score?

Q3 User-Post: Post.ActiveMonth, (AVG)User.Age

For posts of different active timespan, is there a difference in average users’ age?

Q4 User-Post: User.CreationDate_Year

How many posts are posted for users who joined in different years?

Q5 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Class

Do users of different classes of badges have different topics of interest?

Q6 Badge-User, User-Post, Post-Tag: Tag.TagName, Badge.Date_Year

Is there a major shift in topics of interest for users who receive badges in different years?

Q7 Badge-User, User-Post:Badge.Name, Post.PostTypeId

How many posts are posted for different types of posts and badges?

Q8 User-Post, Post-Tag: User.UpVotes, Tag.TagName, Post.PostTypeId=2

What is the distribution of users' upvotes for each topic of answers (Post.PostTypeId=2)?

Q9 User-Post, Post-Tag:User.CreationDate_Year, Tag.TagName

Do users who join in different years have different topics of interests?

Q10 Badge-User, User-Comment: Badge-Class, (AVG)Comment-Score

Do users of higher classes of badges tend to be more "picky with comments?

Q11 Badge-User, User-Comment: Badge-Name, (AVG)Comment-Score

Do users of different types of badges give different comment scores? For example, do "masters" tend to give lower comment scores than "students"?

Q12 Post-PostHistory, Post-Tag: Tag-TagName, PostHistory-PostHistoryTypeId

For different tags, is there a difference in post histories related to the tags? For example, posts of which tags are more often re-edited?