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 semantically rich model for real-world applications that represent their data as graphs, 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 dimensions. For example, given a Q&A forum dataset, in order to study if there is a correlation between a poster'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, it may be interesting to ask what total sales of records is 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 substructures returned from the first step. However, in warehousing data analysis workloads, it is common to have repeating queries from time to time. Computing everything 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 maintenance techniques developed in traditional RDBMS have 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. The essential idea is to precompute and materialize some views wisely based on history workload, such that future workload processing can be accelerated.

We implemented a prototype system on top of Neo4j. Empirical studies over real-world property graph in different size scales show that, with a reasonable space cost constraint, our solution usually achieves 15-30x speedup over native Neo4j 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 ix

List of Figures x

1 Introduction 1

1.1 Property Graph Model . . . 1
1.2 OLAP over Property Graph . . . 4
1.3 Challenges of Graph OLAP . . . 5
1.4 Our Solution and Contributions . . . 7

2 Background and Related Work 9

2.1 OLAP over Property Graph Model . . . 9
2.1.1 Graph OLAP Examples . . . 10
2.1.2 Structure, Dimension, and Measure . . . 12
2.2 Graph Databases and Neo4j . . . 14
2.3 Related Work . . . 14

3 Problem Definition 17

3.1 Terminology . . . 17
3.1.1 Definition of Property Graph . . . 17
3.1.2 OLAP Query . . . 18
3.1.3 Materialization: Cuboid & Substructure . . . 19
3.2 Problem Definition . . . 20

4 Solution **22**
4.1 Solution Framework Overview . . . 23
4.2 Materialized View Selection . . . 24
4.2.1 Overview of Materialized View Selection . . . 24
4.2.2 Greedy Selection Framework . . . 26
4.2.3 CubePlanner . . . 27
4.2.4 Structure Planner . . . 32
4.2.5 ID and Property Selection . . . 35
4.2.6 Update on Materialized Views . . . 36
4.3 Query Processing . . . 36
4.3.1 Substructure Selection . . . 37
4.3.2 Decomposition and Join . . . 39

5 Evaluation **45**
5.1 Experiment Setup . . . 45
5.1.1 Datasets . . . 45
5.1.2 Query Workloads . . . 46
5.1.3 System Setting . . . 47
5.2 Aspects of Interest . . . 48
5.3 Results and Discussion . . . 49
5.3.1 Our System vs. Neo4j . . . 49
5.3.2 Frequency Threshold . . . 51
5.3.3 Space Cost Limit . . . 52
5.3.4 Storage Level for Materialized Views . . . 53
5.3.5 CubePlanner vs PMA . . . 54
5.3.6 StructurePlanner vs FPM . . . 56
5.3.7 Substructure Selection . . . 58
5.3.8 Decompose_Join . . . 58
5.3.9 Reflections on Neo4j . . . 61

6 Conclusion **63**

References **65**

APPENDICES **65**

A PDF Plots From Matlab **66**

A.1 Previous Workload 66

A.2 Future Workload 67

List of Tables

2.1 A summary of graph OLAP literature 16

3.1 Comparisons between Cuboid and Substructure. 20

List of Figures

1.1 A simple property graph modeling "users post posts"(data graph). 2
1.2 Meta graph containing User, Post and Tag. 3
1.3 A snapshot of data graph containing User, Post and Tag. 3
1.4 Neo4j's execution plans for executing "getting the average post score grouped by users upvotes" for the first time and fifth time. 6

2.1 *Structure* of Q1 12
2.2 Cube of properties {A,B,C,D}. 13
2.3 *Structure* of Q3 13

4.1 Solution framework. 23
4.2 Neo4j's execution plan for query *User-Badge, User-Post, Post-Tag: Tag.TagName.* 31
4.3 A substructure lattice with *Badge-User, User-Post, Post-Tag* as its root node. 33
4.4 "Border nodes" of structure *User-Post, Post-Tag.* 35

5.1 The meta graph of StackOverFlow used in experiments. 46
5.2 Time efficiency on the future workload: our solution vs Neo4j. 49
5.3 Substructure selected by StructurePlanner. 50
5.4 Total processing time under different settings of ω. 52
5.5 Efficiency vs Space Cost 53
5.6 Main memory storage vs hard disk storage 54
5.7 Time: CubePlanner vs PMA 55

5.8 Total cuboid space cost: CubePlanner vs PMA 55
5.9 Total processing time for future workload: StructurePlanner vs FPM . . . 56
5.10 Space cost: StructurePlanner vs FPM 57
5.11 Processing time for each query: StructurePlanner vs FPM 58
5.12 Processing time for Q10 - Q12 by three approaches. 59
5.13 Joining time in processing Q10 - Q12 by three approaches. 60
5.14 Total processing time for Q10 - Q12 by three approaches. 60
5.15 Total processing time vs "trial query" processing time. 61
5.16 Execution plans for *User-Post: User.Age* and *User-Post: ID(User), ID(Post).* 62

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 [**?**], PGX [**?**]. 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., business intelligence [**?**], risk management [**?**], trend monitoring [**?**]. 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. In this thesis, we study the efficient processing of OLAP queries over property graphs 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) [**?**], 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 an 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 "*has_tag*" 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.

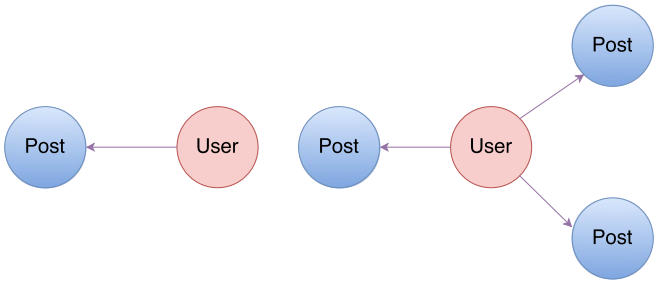

Figure 1.1: A simple property graph modeling "users post posts"(data graph).

To capture the rich semantics of connected real-world entities, property graph model [**?**] is becoming more and more popular due to 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. 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 "creates" connection. From the graph, we can clearly see that there is one user who has created one post while the other user has created 2 posts. In addition, a User node can have properties like the users Age, Views, UpVotes and etc. For clear presentation purpose, we shall use a property graph which contains information of an online Q&A forum `www.stackoverflow.com` through this thesis. We name this graph "StackOverFlow graph".

Note that although the property graph model does not enforce any restriction on what properties a node or edge can have, a high-level abstraction describing the property relations, named the "meta graph", is often 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. Figures 1.2 and 1.3 are the meta graph and a snapshot of the data graph of the StackOverFlow graph, respectively. As shown in Figure 1.2, there are three types of entities: User, Post, and Tag, which are colored in red, blue and green in the data graph. Each user has a property named "Age", each post has a property named "Score", and a property "TagName" associated with each tag. There are two types of edges being defined: "creates" and "has_tag". Note that figure 1.3 is a snapshot of the StackOverFlow data graph. The actual data graph contains millions of nodes and edges.

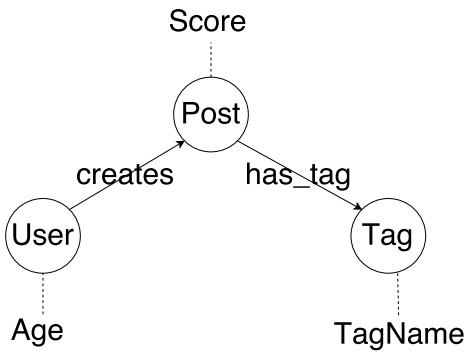

Figure 1.2: Meta graph containing User, Post and Tag.

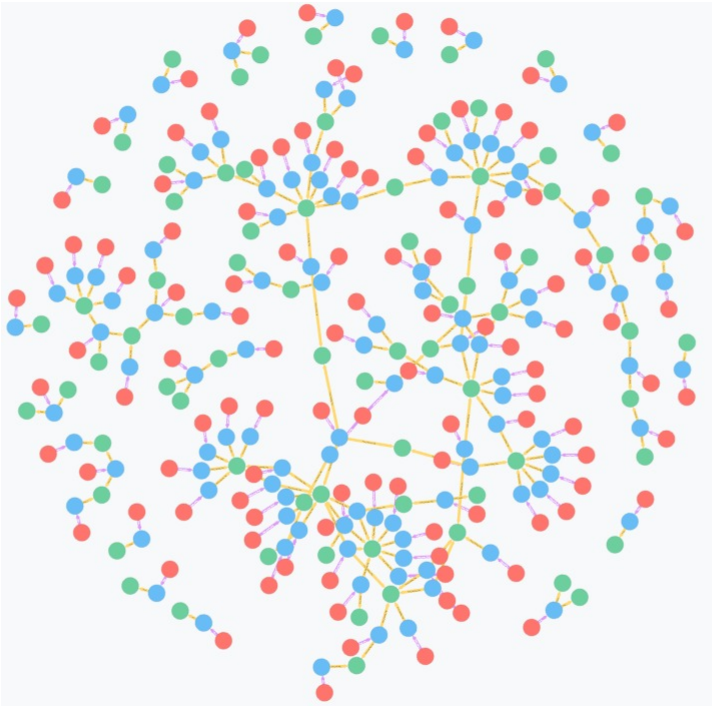

Figure 1.3: A snapshot of data graph containing User, Post and Tag.

1.2 OLAP over Property Graph

In traditional databases and data 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 users to view data in more detail, while drill-down operation does the opposite. Slicing enables filtering on data. For example, we can perform OLAP to analyze earning performance of an international company by different branches. 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. Then we can proceed and perform a roll-up operation by discarding the branch dimension and only look into profit performance by seasons. Finally, we can perform a slicing operation by adding a condition, say summer, on season dimension. By this we only forcus on profits in the summer season. 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 StackOverFlow 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 scenarios 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 relational database management systems (RDBMS for short) [**?**] or warehousing applications is a well studied topic. There is abundant literature attacking this problem from various different perspectives, e.g. data partition [**?**], view selection [**?**], partial materialization [**?**]. However, there is very few research efforts on the Graph OLAP. Existing literature concerning OLAP workload over graph data either targets accelerating graph OLAP over a special subset of property graphs [**?**], or addresses generic high-level topics, such as an expressive graph aggregation operator for RDF graphs proposed in [**?**].

Our empirical study shows that existing graph databases do not provide efficient support for graph OLAP, especially when the graph size scales to real-word graph sizes, 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. Figure 1.4 reveals that even if we repeatedly executed the same query using Neo4j, 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 StackOverFlow 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 real-time analytics.

As a matter of fact, previous workload provides useful information for future workloads. This is because in real case users do not generally 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 workload history and can serve as an insightful hint on future workloads. Our claim is that by sacrificing some memory space and materialize "hot" structures and properties even before future queries arrive, future queries can be faster processed.

The real challenge is how to design a score function to evaluate the trade-off between such benefit and cost so that we can select best queries for materialization. Here best materialization refers to the case where we achieve best future workload acceleration with a given memory constraint for materialization.

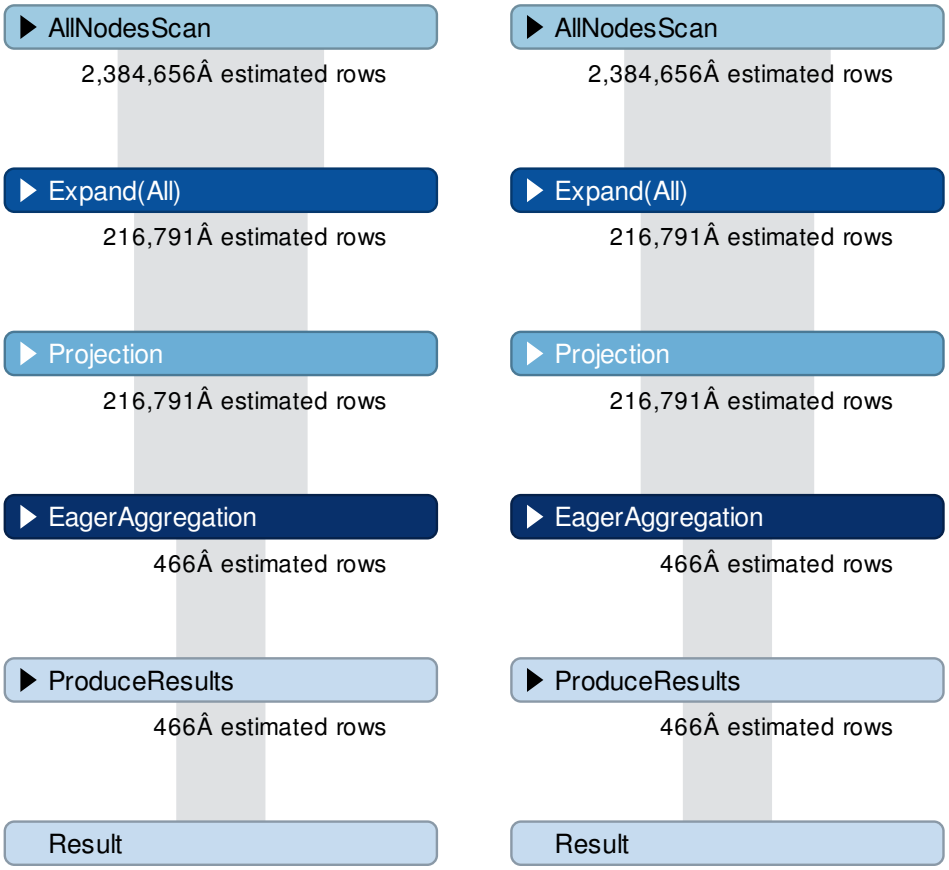

Figure 1.4: Neo4j's execution plans for executing "getting the average post score grouped by users upvotes" for the first time and fifth time.

1.4 Our Solution and Contributions

To address the challenges discussed above, we propose an end-to-end solution to support efficient OLAP over large property graphs in graph databases.

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 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 summarize 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, which is followed by the background knowledge about OLAP, graph databases, and Neo4j. Besides 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 discussion 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

As noted in the previous chapter, 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 IDs of nodes or edges of a certain type as ID([NodeType]) or ID([EdgeType]). For example, ID(User) refers to unique IDs of "User" nodes. Note that in some scenarios, a unique ID can be considered as a special kind of property for its corresponding node or edge.

OLAP (On-Line Analytical Processing) [**?**, **?**, **?**] 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 in this chapter.

2.1.1 Graph OLAP Examples

Graph OLAP was first proposed by Graph Cube [**?**] to refer to OLAP over graphs, although no formal definition of the notion Graph OLAP was given. In order to better elaborate how "Graph OLAP" is interpreted in this 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?

Q1: 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 user's post quality. Suppose we would like to propose suggested posts based on scores. When a post is freshly posted and the score of the post has not been voted, we may use the authors upvotes as a factor to estimate the quality of his or her post.

Example 2 Following the context of Q1, but this time we want to take a closer look at how upvotes are correlated with post scores for different types of posts. It is reasonable to guess that authors' upvotes make more difference in scores of their answers (posts with PostTypeId 2) than their questions (posts with PostTypeId 1), because a user's level of expertise is usually better reflected by his or her answers.

Q2: 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 reveal that users with higher upvotes not only provide good answers but ask valuable questions as well. However, there is a subtle difference on how upvotes are correlated with scores of 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 tolerant of a naive question rather than with a wrong answer.

Q1 and Q2 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, what is the weighted average age of users? For instance is python more trendy than c among young users?

> Q3: For each tag, get the weighted average age of users who have posted posts on the tag in 2017. In other words, we group all the posts that were posted in 2017 by their tag names and calculate the weighted average users' age under each tag.

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 users' age with respect to each tag clearly and easily compares them. It reveals some interesting insights. We know that Python may

have a relatively young user group, compared with C. Similarly, "Router" is a topic that is more often discussed among youngsters than the general topic of "Internet". Note that we are led to the above findings based on the recent posts (posted in year 2017).

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 Q1 as an example.

Q1 concerns the structure (colored in blue) shown in Figure 2.1.

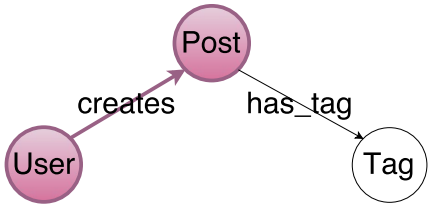

Figure 2.1: *Structure* of Q1

We say that (User)-[creates]->(Post) is the structure of Q1. The query is first aggregated on users' upvotes. We say that User.Upvotes is the dimension of Q1, and the output of the query is an aggregation function on posts score. We say that AVG(Post.Score) is the measure of Q1. Similarly, consider Example 2, which shares the same structure as shown in Figure 2.1. The dimensions of Q2 is User.Upvotes, Post.PostTypeId, and the measure is AVG(Post.Score). Note that Q2 adds Post.PostTypeId to Q1's dimensions. In other words, Q2 asks for a more detailed partitions over dimensions. We call Q2 a drill-down from Q1, and Q1 is a roll-up from Q2. 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.

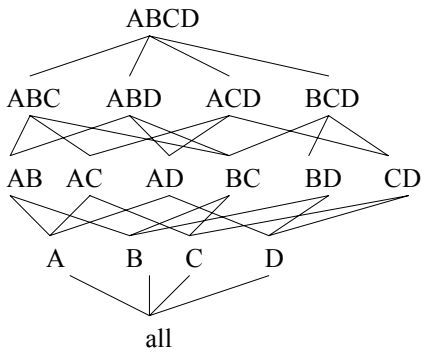

Figure 2.2: Cube of properties {A,B,C,D}.

Q3: Get average user age grouped by users 2017 posts tags.

Structure: (User)-[creates]-(Post)-[has_tag]-(Tag)

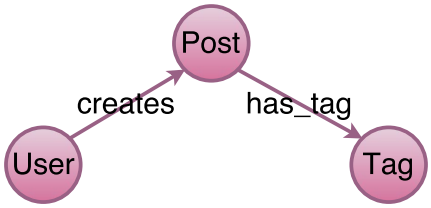

Figure 2.3: *Structure* of Q3

Dimensions: Tag.Tagname

Measure: AVG(User.Age)

Note that Q3 has a different *structure* than Q1 and Q2, as shown in Figure 2.3. Q3 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 constraints on selected properties. In this thesis we call these constraints (e.g., {Post.Year=2017} in Q3) *"slicing conditions"*.

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 [**?**] [**?**]. Relational and graph database systems both have their own strengths in term of query processing. In this thesis, we work with graph database systems.

In graph databases, the storage structure that is most commonly used in an adjacency list. An adjacency list is a list that stores all the edges and neighbor nodes for a given node. A popular graph database system is Neo4j. Like most traditional RDBMS, Neo4j holds atomicity, consistency, isolation and durability (ACID). Neo4j's native graph processing engine enables SQL like queries over property graphs which are managed by its own graph storage. Common features in RDBMS such as indexes building and constraints are also supported. REST API plus official drivers for programming languages such as Java are provided for the sake of easy interactions with other programs. One most special feature about Neo4j is that it allows multiple-labeled nodes and edges; for example, a node referring to a male student could have various labels such as "student" and "male" and etc.

Cypher, Neo4j's query language, is a powerful SQL like 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:creates]->(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 [**?**] 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 [**?**] introduces an RDF graph aggregation operator that is both expressive and flexible. It defines its operational semantics on top of SPARQL algebra, and proposes an algorithm to answer graph aggregation queries. Gagg achieves significant improvements in performance compared to plain-SPARQL graph aggregation.

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

Graph Cube [**?**] introduces a data warehousing model that supports OLAP queries effectively on large multidimensional networks. It takes account of both attribute aggregation and structure summarization of the networks. In order to deal with "curse of dimensions", a greedy algorithm framework is introduced for partial materialization of cuboids. Moreover, it addresses and defines two most important notions in graph OLAP scenarios as *dimension* and *measure*. Note that in our work, *structure* (of meta graph) is addressed as a third important notion (as discussed in 2.1.2). Graph Cube [**?**] focuses on OLAP senerios over a fixed *structure*, with *dimension* and *measure* varied, whereas our work deals with OLAP workloads over various *structures*.

Graph OLAP [**?**] 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 (e.g., distributive measures and holistic measures) 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 [**?**], 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 [**?**] that as long as the domain of measure is a subset of {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 means that once a cuboid is materialized, all roll-up operations from this

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

Table 2.1: A summary of graph OLAP literature

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 the number of dimension is large, number of cuboids grows exponentially, making total materialization hard due to overwhelming space cost. To solve this Graph Cube [**?**] 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.

From the summary, we categorize the existing work into two lines. The first focuses 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 (e.g., Graph Cube [**?**]). Although attribute-aware optimizations are proposed in some literature, structure-aware optimizations are not studied. Second focuses on a relatively high-level framework that processes generic queries over generic property graphs (e.g., Gagg [**?**]). However, query processing efficiency is not the main focus.

To conclude, there is not work 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, we investigate faster structure-aware OLAP processing over general property graphs.

Chapter 3

Problem Definition

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

3.1 Terminology

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

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;
- $f_{Vid} : v_i \rightarrow vid_i, v_i \in V, vid_i \subseteq Vid$, maps each node to its unique ID;

- $f_{EA} : e_i \rightarrow A_i, e_i \in E, L_i \subseteq A$, maps each edge to its properties;
- $f_{EL} : e_i \rightarrow L_i, e_i \in E, L_i \subseteq L$, maps each edge to its labels;
- $f_{Eid} : e_i \rightarrow eid_i, e_i \in V, eid_i \subseteq Eid$, 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 Q3 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, Q3's *structure* as shown in Figure 2.3 is written as *"User-creates-Post, Post-has_tag-Tag"*.

Dimension: A *Dimension* is written by listing all properties that act as dimensions in an OLAP query. For example, Q3's *dimension* is written as *"Tag.Tagname"*.

Measure: We focus on three most common types of *measure*: *COUNT, SUM and AVG.* Q3's *measure* is written as *"AVG(User.Age)"*.

Slicing Condition: A *Slicing Condition* is written as "*Property = value*". Q3's *slicing condition* 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 Q3 is written as

User-creates-Post, Post-has_tag-Tag: Tag.Tagname, AVG(User.Age), Post.Year=2017

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

We define some notations adopted in this thesis. Given an OLAP query q, we use *"q.properties"* to refer to the set of **all properties** that appear in *Dimension, Measure, and Slicing Condition* of q. We use *"q.structure"* to refer to the structure of q. For example, *Q3.properties=*{*Tag.Tagname, User.Age, Post.Year*}, and *Q3.structure=*{ *User-creates-Post, Post-has_tag-Tag*}.

3.1.3 Materialization: Cuboid & Substructure

As previously discussed, a key issue of our work is to to find the most useful common substructures and properties from the previous queries, which are assumed to appear in the future workload frequently. For fast access to results of these useful queries, we store their results (preferably in main memory), and this is known as the *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 the 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 or not unique IDs of nodes (or edges) are kept. To better elaborate the differences between cuboid and substructure, we consider the following example. Suppose we have the following query workload containing two history queries (P1 and P2) and two incoming queries (Q1 and Q2):

P1 User-creates-Post: User.Age
P2 User-creates-Post: User.Age, (AVG)Post.Score
Q1 User-creates-Post: (AVG)User.Age, Post.Score
Q2 User-creates-Post, Post-has_tag-Tag: User.Age, Tag.TagName

We can tell that users are most interested in the *User-creates-Post* structure. {*User.Age, Post.Score*} is the set of properties being involved in queries over *User-creates-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-creates-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. The above materialized view is useful for Q1 since we can process Q1 by aggregation over M_1. We call such materialization a **cuboid**.

However, M_1 is not useful for Q2 since they have different *structures*. On the contrary, if we add *ID(Post)* to *dimension* and materialize

M_2: *$User-creates-Post: User.Age, Post.Score, ID(Post), COUNT(*)*

so that *Post* is "activated" to be able to join with other materializations containing *Post* and produce results for OLAP over more complicated *structures*. For example, Q2 can be processed through the following steps:

	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.

Step 1: joining *$User-creates-Post: User.Age, Post.Score, ID(Post) COUNT(*)* and *$Post-has_tag-Tag: ID(Post), Tag.TagName, COUNT(*)* on *ID(Post)*;

Step 2: perform aggregation on {*User.Age, Tag.TagName*}.

In this case, we only need to fetch *$Post-has_tag-Tag: ID(Post), Tag.TagName, COUNT(*)* from database to produce result for Q2. 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 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

σ. The query processing step involves "Execution Planning" (EP for short). We formally define these two problems as follows.

Materialization Selection Problem: Given a property graph G, a set of previous queries P on G, space limit σ, find cuboids C and substructures S, such that: 1) $\sum_{c_i \in C} c_i.space + \sum_{s_i \in S} s_i.space \leq \sigma$; and 2) $\sum_{p_i \in P} T(G, p_i, C, S)$ 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 $process(G, q, C, S)$, such that the processing time of q, denoted by $process(G, q, C, S).time$, is minimized.

As a matter of fact, execution planning can be further divided into two sub-problems. 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 the 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 G, a future query q, materialized cuboids C' and substructures S', and remaining views R, find a composition plan $compose(G, q, C', S', R)$, so that estimated composition time $compose(G, q, C', S', R).time$ is minimized. Here $compose(G, q, C', S',$ returns result of query q by performing operations (join, selection, projection etc.) over C', S', R.

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

Note that we define "Composition Problem" before "Decomposition Problem". The reason is that we need to consider a composition plan $compose(G, q, C', S', R)$ 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

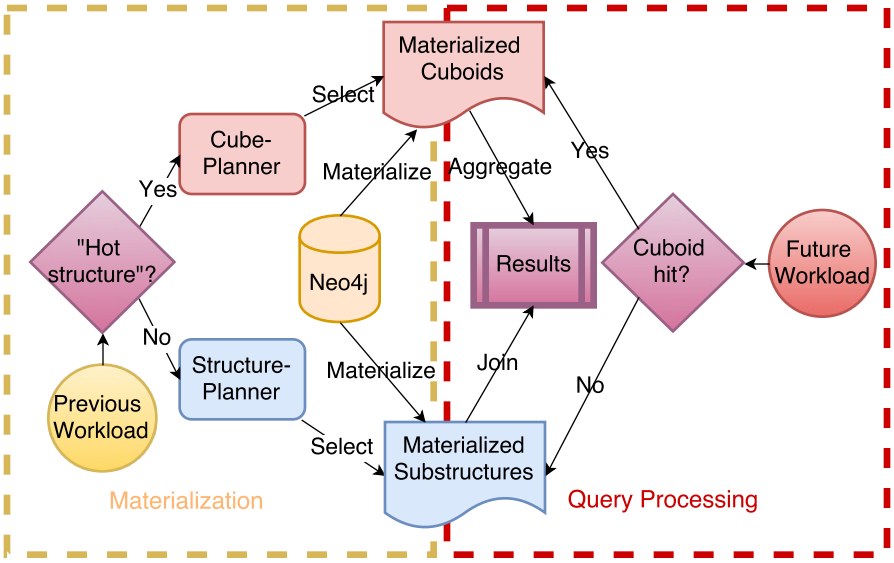

Figure 4.1: Solution framework.

Figure 4.1 describes the overall solution framework. Two dashed line rectangles represents the major components of our solution: materialization and query processing. Materialization takes previous workload as input and materializes cuboids and substructures which are beneficial for future workload processing. We adopt a straightforward best effort approach for materialization. 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 incoming queries as input and returns results. Briefly, the workflow 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 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 research problem that has attracted significant attention 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. This is mainly because *structure* plays an important role in a query over a property graph. One materialized view may affect the benefits of other ones (e.g., when their *structures* overlap). Therefore combinational benefit for a set of materialized views is not a simple sum of benefits for each one of them. 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 discussed 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 structural 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 shares the same structure, where ω is a predefined frequency threshold, this structure is considered to be a *hot structure* and would be passed on to the *CubePlanner* module for cuboid selection. Queries that 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: ω: 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

```
1  CInput ← ∅;
2  SInput ← ∅;
3  foreach q ∈ Q do
4  |   if structureFreq(Q, q) > ω then
5  |   |   CInput ← CInput ∪ {q};
6  |   end
7  |   SInput ← SInput ∪ {q};
8  end
9  C←materialize(CubePlanner(CInput));
10 S←materialize(StructurePlanner(SInput));
```

As shown in Algorithm 1, function $structureFreq(Q, q)$ returns the frequency count of all query 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:
P1 Badge-User, User-Post:Badge.Name,Post.Score,Post.PostTypeId=2
P2 User-Comment, Comment-Post: User.UpVotes, Comment.Score, (AVG)Post.Score, Post.PostTypeId=1
P3 User-Post, Post-Vote: User.UpVotes, Vote.VoteTypeId
P4 User-Post, Post-Tag: (AVG)User.CreationDate_Year, Tag.TagName
P5 User-Comment, Comment-Post: User.ActiveMonth, Post.CreationDate_Year=2016
P6 User-Comment, Comment-Post: User.Age, (AVG)Comment.Score, Post.PostTypeId=2
Future Workload:
Q1 User-Comment, Comment-Post: User.UpVotes, (AVG)Post.Score, Post.PostTypeId
Q2 User-Comment, Comment-Post: User.Age, Post.PostTypeId
Q3 User-Post, Post-PostHistory: User.UpVotes, PostHistory.PostHistoryTypeId
Q4 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

Obviously, *User-Comment, Comment-Post* is a *hot structure*. We materialize cuboids over structure *User-Comment, Comment-Post* by passing previous query P2, P5 and P6 to CubePlanner. CubePlanner will materialize cuboids that benefit processing of future queries Q1 and Q2 (which have *User-Comment, Comment-Post* structure). Then, we pass the three remaining queries of less hot structures, P1, P3, and P4 to StructurePlanner, which 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 queries Q3 and Q4.

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 a known NP-complete problem [**?**]. The difficulty lies in that the overall benefit of materialized views is not a simple sum of the individual benefits 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 while the space limit is strictly respected.

We 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 marginal benefit against the space limit. After a candidate is picked, we re-evaluate the benefit of remaining candidates. Re-evaluation is mandatory as the marginal benefit of a candidate may be reduced as a result of materialization of a selected candidate.

Algorithm 2: Greedy Selection

System setting: σ: space limit
Input: C: a set of candidates of cuboids or substructures in lattice structure
P: A set of previous queries
Output: R: a list of selected candidates to materialize

```
1  foreach c ∈ 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 < σ do
7  |  selected ← c in C with highest score;
8  |  R.add(selected);
9  |  repeat Lines 1-5;
10 end
```

Lines 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**. Lines 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 plugging in their own implementation of the score calculation function that may consider different database features such as execution plans. 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 materialization. As mentioned in Section 3.1.3, a cuboid is only useful for

queries sharing the exact identical 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 a process is to first hold regional competitions and then select national winners from regional winners. Next we will explain *CubePlanner* and *SingleCubePlanner* in detail.

CubePlanner

As we mentioned above, *CubePlanner* performs cuboid selection in a holistic manner by selecting cuboids one-by-one from results of *SingleCubePlanner*. We first explain the workflow of *CubePlanner*, as shown in Algorithm 3. Intuitively, *CubePlanner* first groups Q by structure using the function $group(Q)$. Lines 2-4 perform cuboid selection in each partition using *SingleCubePlanner*. A queue of ordered candidates is generated within each group of queries. Lines 5-8 repeatedly check 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: n: maximum number of cuboids to be precomputed
Input: Q: a set of previous queries not necessarily with the same structure
Output: C: a queue of selected cuboids to be precomputed

```
1 Group← group(Q);
2 foreach group ∈ Group do
3 |   group.results ← SingleCubePlanner(group);
4 end
5 for i=1 to n do
6 |   group' ← group in Group with highest group.results.top().score;
7 |   C.offer(group'.Dequeue());
8 end
```

SingleCubePlanner

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

Algorithm 4: SingleCubePlanner

System setting: n: as in "top-n"
Input: P: a set of previous queries with a same structure
Output: C: a queue of selected cuboids to precompute

```
1  Lattice ← buildLattice(Q);
2  foreach query Q ∈ P do
3  |  q.time ← time(q);
4  end
5  foreach cuboid ∈ Lattice do
6  |  cuboid.space ← space(cuboid);
7  |  cuboid.benefit ← 0;
8  |  foreach query Q ∈ P and q.properties ⊆ cuboid.properties do
9  |  |  cuboid.benefit+ = max(0, q.time − aggreTime(cuboid));
10 |  end
11 |  cuboid.score ← cuboid.benefit/cuboid.space;
12 end
13 for i=1 to n do
14 |  nextBestCube ← cuboid in Lattice with highest score;
15 |  if nextBestCube.score < 0 then
16 |  |  break;
17 |  end
18 |  C.Enqueue(nextBestCube);
19 |  foreach cuboid Q ∈ Q and q.dimension ⊆ nextBestCube.dimension do
20 |  |  q.time ← min(q.time, aggreTime(nextBestCube));
21 |  end
22 |  Repeat 5-12;
23 end
```

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 [**?**]. Lines 2-4 initialize the best-so-far processing time for each previous query with its estimated naive database processing time. Lines 5-12 perform

score calculation following the greedy selection framework presented in Algorithm 1. For each cuboid, we estimate its space (line 6). Lines 8-10 calculate the marginal benefit by iterating over previous queries that can be answered by 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). Lines 13-23 perform the pick-and-update, where lines 15-17 terminate the selection process when there is no more extra marginal benefit, and lines 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 time of naïve processing of a query in a graph database. Implementation of $time(query)$ is database specific as physical storage and execution plans vary among different databases (i.e., not using materialized views). 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 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 the sum of estimated rows to estimate the total processing time cost.

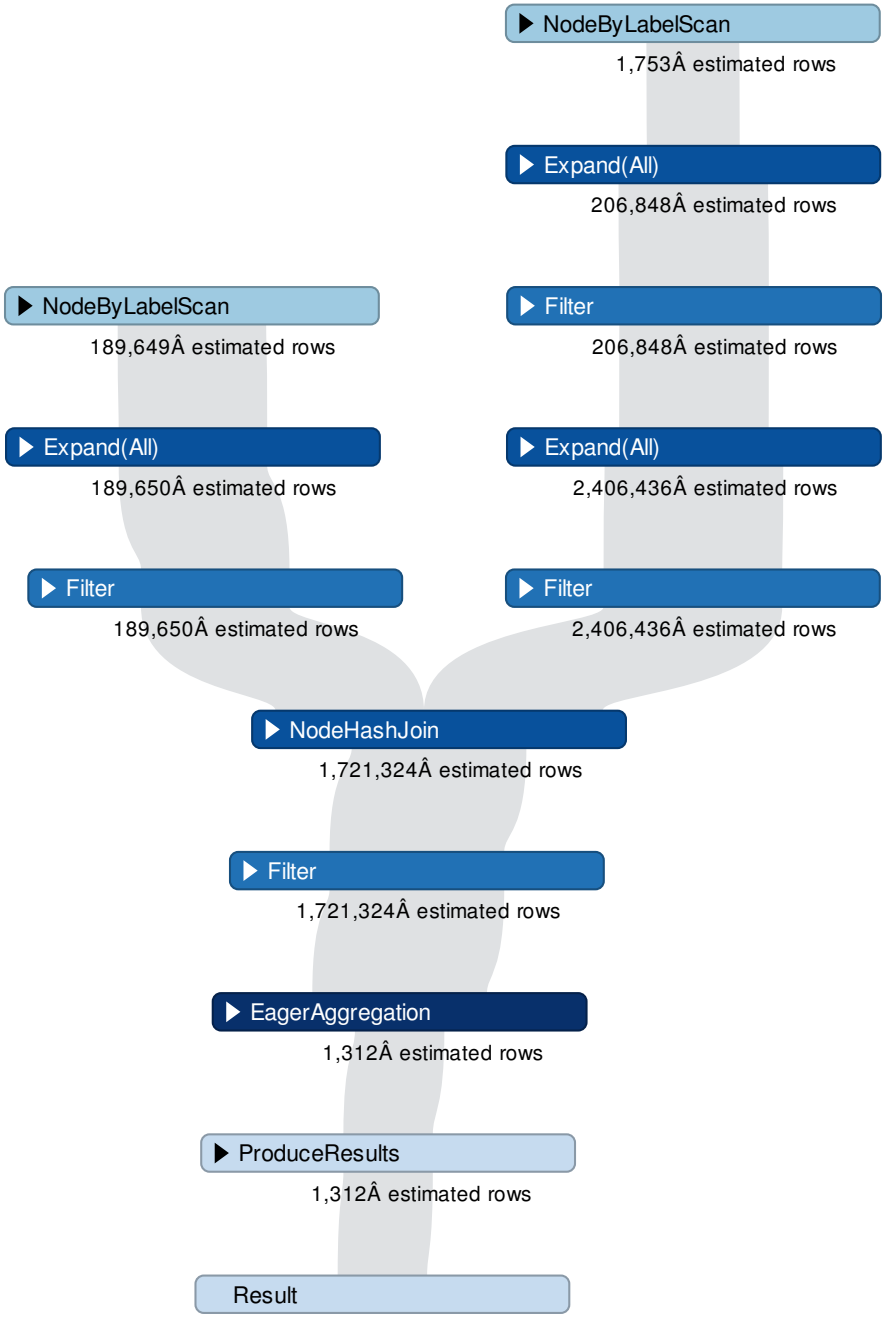

Figure 4.2: Neo4j's execution plan for query *User-Badge, User-Post, Post-Tag: Tag.TagName*.

For graph databases that do not provide an API to access execution plans and estimated intermediate result sizes, users can construct different estimation functions following the same intuition, which usually depends on specific database implementations. There are many studies on cost estimation for database operations (e.g., join operation). Users may consider joining (expanding) order [**?**] and estimation of intermediate result sizes [**?**] 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 **??**. 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.

Then, we initialize the covered substructures for each previous query as an empty set (lines 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. Lines 5-12 perform score calculation. For each substructure, Line 6 estimates its space. Lines 8-10 iterate over all "favored" previous queries (favored by current substructure) and

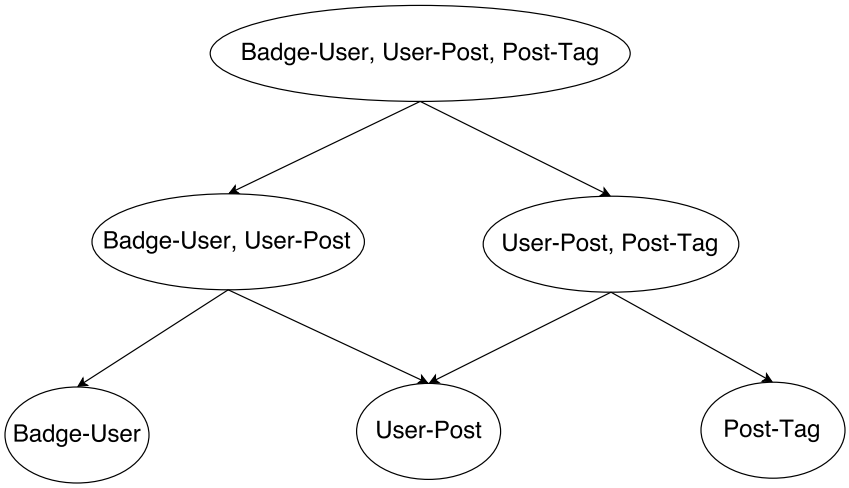

Figure 4.3: A substructure lattice with *Badge-User, User-Post, Post-Tag* as its root node.

add on the marginal benefit (if any). Here marginal benefit refers to the time saved after adding current substructure to selected substructures (Line 9). Lines 13-23 perform the pick-and-update, and lines 19-22 update 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 **??**. Note that these are specific to our implementation in Neo4j and may differ in the case of other 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 when substructure materialization is used, the actual execution plan (intermediate result) may be different than the naïve processing plan. As a result, estimating the marginal benefit of a substructure is tricky. We estimate the marginal benefit of a substructure using

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

which captures the overall improvement of adding *substructure* to *coveredSubstructres* as materializations.

Algorithm 5: StructurePlanner

System setting: σ: space limit for materialized views
Input: Q: a set of previous queries
Output: S: an queue of selected substructures to precompute

```
1  Lattice ← buildSubstuctureLattice(Q);
2  foreach q ∈ Q do
3  |  q.coveredSubstructres ← ∅;
4  end
5  foreach substructure ∈ Lattice do
6  |  substructure.space ← space(substructure);
7  |  substructure.benefit ← 0;
8  |  foreach q ∈ Q and q.structure ⊆ substructure.structure do
9  |  |  cuboid.benefit+ = max(0, benefit(q, substructure, q.coveredSubstructres));
10 |  end
11 |  substructure.score ← substructure.benefit/substructure.space;
12 end
13 while System.memoryUsage < σ do
14 |  nextBestSubstructre ← substructure in Lattice with highest substructure.score;
15 |  if nextBestSubstructre.score < 0 then
16 |  |  break;
17 |  end
18 |  S.offer(nextBestSubstructre);
19 |  foreach q ∈ Q and q.structure ⊆ nextBestSubstructre.structure do
20 |  |  q.coveredSubstructres ← q.coveredSubstructres ∪ {nextBestSubstructre};
21 |  end
22 |  repeat 5-12;
23 end
```

4.2.5 ID and Property Selection

Given a substructure picked by the *StructurePlanner*, we need to decide which IDs and properties need to be stored. Keeping all IDs and attributes makes a substructure materialization more informative but increases the space cost. Thus, the selection of IDs and properties is an important issue. We 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. 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*. This saves space compared with Policy #1, however 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. But "overlap join" with other substructures is not enabled because IDs of "inner nodes" are not stored. We cannot join *User-Post, Post-Tag* with *Badge-User, User-Post* as Post is an "inner node" and IDs of Post is not stored.

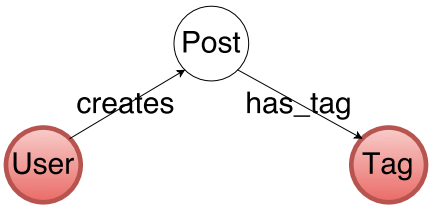

Figure 4.4: "Border nodes" of structure *User-Post, Post-Tag*.

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

The solution we have discussed so far is applicable for static scenarios (with fixed previous workload). We now expand our solution to dynamic scenarios: as queries are executed, 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. After executing a certain number of queries, we perform materialized view selection over only "recent queries" which are in the sliding window. Old views that need to be swapped out are eliminated by simply releasing their memory. For newly selected views we materialize them. For old views that do not need to be swapped out, 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 incoming queries efficiently using materialized substructures and cuboids. When a query q arrives, we first consult materialized cuboids. 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 ← ∞;
2  mincuboid ← NULL foreach cuboid ∈ C do
3      if cuboid.structure = q.structure and q.dimension ⊆ cuboid.dimension then
4          if cuboid.space < minspace then
5              minspace ← cuboid.space mincuboid ← cuboid end
6          end
7          if mincuboid ≠ NULL then
8              r ← aggregate(mincuboid, q);
9          else
10             r ← Decompose_Join(q);
11         end
12     end
13
```

Algorithm 6 describes the generic work flow of query processing. Given an incoming query q, we first look up materialized cuboids and find if any cuboid can be used to answer q (lines 4-9). Note that *cuboid.space* in line 5 was computed in line 9 in Algorithm 4. Then, we check if q can be answered by cuboid materialization. If so, 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) seem to be a possible combination for substructure selection in this case. Actually we may have at least three possible substructure selections: (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 example heuristics are #edges of substructure, score calculated in

StructurePlanner (Line 11 in Algorithm **??**), 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 uncoveredProp ← q.properties coveredStruc← ∅;
2 V ← ∅;
3 foreach s ∈ S ordered by h(s) do
4     if s ⊆ uncoveredStruc and s ⊈ coveredStruc then
5         V ← V ∪ {s};
6         coverdStruc ← coveredStruc ∪ s.structure;
7         uncoveredStruc ← uncoveredStruc − s.structure uncoveredProp ←
            uncoveredProp − s.properties end
8     end
9
```

Lines 1-2 initialize *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 the 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

In this section, we 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 cover 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 the database. We call such remaining structures and properties fetched from the database "complementary components". After all these components (both materializations and "complementary components") are finally ready, we join and aggregate them 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 $Splits(uncoveredStruc, uncoveredProp)$;
5 **foreach** *s: Splits* **do**
6 $\quad \Sigma \leftarrow \Sigma \cup \{retrieve(s)\}$;
7 **end**
8 $r \leftarrow join\_aggregate(\Sigma, q)$;

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 retrieve each connected component from the 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 a classic connected components detection algorithm. It splits *uncoveredStruc* and *uncoveredProp* into connected components (structures). We want to retrieve each connected structure separately from the 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 the database server. Function *join(*Σ*, q)* join tables of Σ together

and aggregate over properties based on q. Joins over multiple tables is a well-studied topic. Joining order and join technique (hash join etc) are two important aspects of 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 the database in a naive manner. We adopt the idea of Semi-Join [**?**] 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 the 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 the 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 Σ ← ∅;
2 V, uncoveredStruc, uncoveredProp ← SelectSubstrucre(q);
3 V*(V);
4 Σ ← Σ ∪ V;
5 Splits(uncoveredStruc, uncoveredProp);
6 foreach s: Splits do
7 |   Σ ← Σ ∪ {retrieve*(s, V*)};
8 end
9 r ← join_aggregate(Σ, q);
```

We explain implementation of function $retrieve^*(s, V^*)$. It fetches results from the database 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 $Decompose\_Join^*$. $Decompose\_Join^*$ 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. The disadvantages are twofold. First, $Decompose\_Join^*$ has to transmit IDs. Second, $Decompose\_Join^*$ performs two rounds of join; first on V before "complementary components" are ready, followed by second round of joins. In terms of optimization on join orders, $Decompose\_Join$ is better because its join order is decided over all the tables, whereas in $Decompose\_Join^*$ tables are seperated by two rounds of joins, which narrows down the scope of all possible join orders.

#3 $Decompose\_Join^+$

We 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 $retrieve(s)$. $Decompose\_Join^+$ is derived from $Decompose\_Join^*$. The trick is to revise 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

```
1  Σ ← ∅;
2  V, uncoveredStruc, uncoveredProp ← SelectSubstrucre(q);
3  V*(V);
4  Σ ← Σ ∪ V;
5  Splits(uncoveredStruc, uncoveredProp);
6  foreach s: Splits do
7  |  if decide(s,V*) then
8  |  |  Σ ← Σ ∪ {retrieve*(s, V*)};
9  |  else
10 |  |  Σ ← Σ ∪ {retrieve(s)};
11 |  end
12 end
13 r ← join_aggregate(Σ, q);
```

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

1. Randomly sample a small number of candidate IDs.
2. Execute $retrieve^*$ but pass only sampled candidate IDs in (1). We call this a "trial query". We want to use the "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".

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 running some prior tests in order to adjust γ to a proper value. The prior tests are conducted by executing some sample queries by both

$Decompose\_Join$ and $Decompose\_Join^*$. In each test, the value of $(retrieve(s).estSize - retrieve^*(s, V^*).estSize)/sizeOf(candidateIDs)$, together with correct decision between $Decompose\_Join$ and $Decompose\_Join^*$ (the one with less processing time) are recorded. We view it as a binary classification problem and it can be easily solved by linear regression.

We compare $Decompose\_Join^+$ 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 to make a wise decision on "complementary components" retrieval. Our experiments show that a good decision making on "complementary components" retrieval saves much more time than the time cost of "trial queries", results and discussion of which will be presented in next chapter.

Chapter 5

Evaluation

In this chapter, we validate 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 experiments, we test query processing time and space cost by running the queries using both our approach and Neo4j implementation. 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 Section 5.2). Moreover, we assess and explain how each aspect in our system affects processing efficiency. Finally, we discuss our reflection on the Neo4j system.

5.1 Experiment Setup

There are two purposes of our experiments. First 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.

5.1.1 Datasets

We use the StackOverFlow dataset in our experiments that 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 that are involved in our experiments. The data graph contains over 300 million vertices and more than 400 million edges and takes 44.5GB of storage.

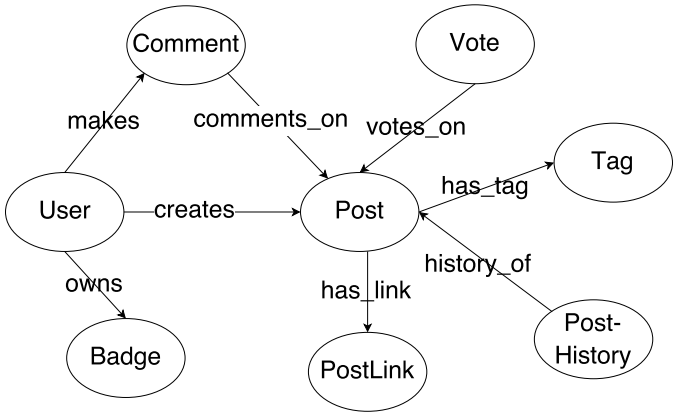

Figure 5.1: The meta graph of StackOverFlow used in experiments.

5.1.2 Query Workloads

We design 24 queries against the StackOverFlow dataset given below. We randomly select 12 queries as the previous workload, leaving the rest 12 ones as the future workload.

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 the experiments on a Linux server with 256GB main memory. Our system is implemented in Java. We set the initial JVM memory to 100GB, and the maximal JVM memory to 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.

Materialization

- Space cost limit (σ in Algorithm 5 in Section 4.2.4) is set to 6GB by default. Results and discussion are presented in Section 5.3.3.
- Algorithms in materialized view selection described in Section 4.2. For cuboid selection, we compare CubePlanner (in Section 4.2.3) with the "Partial Materialization" algorithm (PMA) proposed in Graph Cube [**?**]. For substructure selection we will compare StructurePlanner (in Section 4.2.4) with the well-known maximal frequent pattern mining algorithm (FPM) [**?**]. In other tests, CubePlanner and StructurePlanner are used by default. Results are discussed in Sections 5.3.5 and 5.3.6, respectively.
- Frequency threshold for identification of hot structures (ω in Algorithm 1). We set the default value to be 4; the intuition of default value, together with results and discussion are elaborated in Section 5.3.2.
- Storage level for materialized views. We compare main memory storage vs 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" ($h(s)$ in Algorithm 7 discussed in Section 4.3.1). 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 5.3.7.
- Choice among using *Decompose_Join*, *Decompose_Join** and *Decompose_Join*$^+$ in "Decomposition and Join" in Section 4.3.2. We use *Decompose_Join* as the default method. Detailed discussion is in Section 5.3.8.

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.

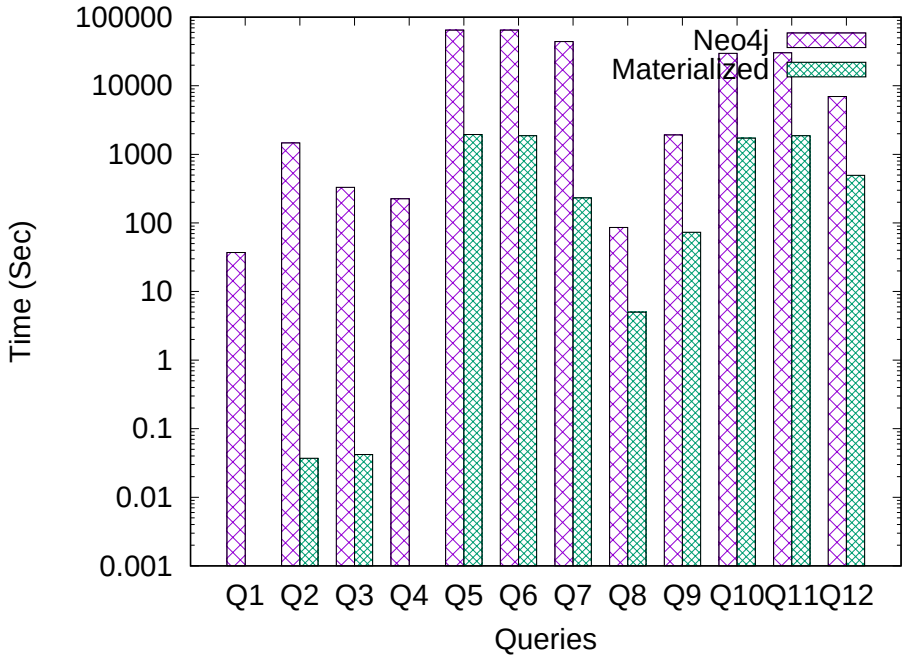

Figure 5.2: Time efficiency on the future workload: our solution vs Neo4j.

Figure 5.2 shows the processing time for 12 future queries by both our system and Neo4j. Note that execution time for Q1 and Q4 in our system is 0.001s, and Neo4j does not finish processing Q5 and Q6 in 18 hours so we terminate execution and record the processing time as 18 hours. It is worth pointing out that with an extra 5.7GB space cost for materialized views, our solution achieves remarkable efficiency improvement. Given the size of our dataset, this extra space is acceptable.

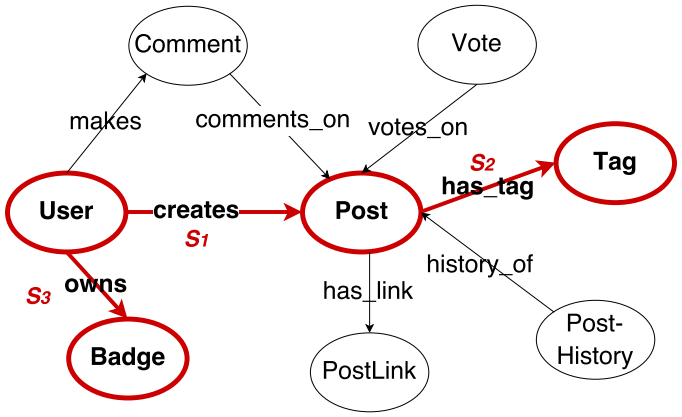

Figure 5.3: Substructure selected by StructurePlanner.

We now detail how our system works over the 12 queries. 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 greater 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 dimensions. 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 operations increases the total time cost. Overall, the overall improvement rate for Q5 - Q9 is around 40 times, while improvement rate for Q10 - Q12 ranges around 15 times. Clearly, our system could greatly improve query processing efficiency with an acceptable incremental 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 threshold for building a cuboid over a "hot structures" ("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
User-Post, Post-Vote	1
Post-Comment, Post-PostLink	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 do 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.

Suppose we increase ω to 5; then there will be no cuboid that is 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 the 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), which is left as future work since it is not the main focus of this thesis.

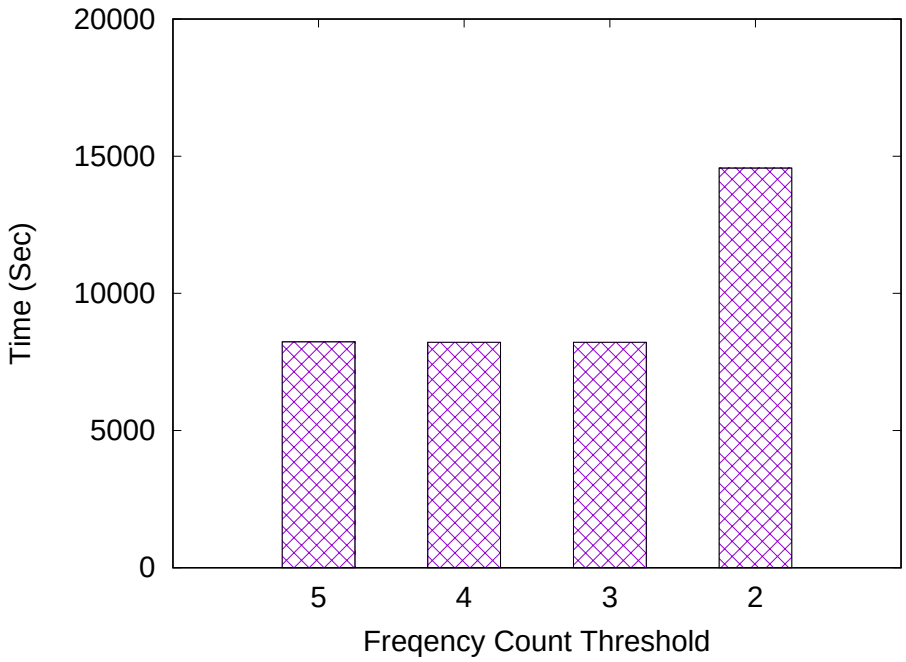

Figure 5.4: Total processing time under different settings of ω.

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 time decreases. However, the marginal benefit from materializing more views also decreases. This indicates that our Greedy Selection Framework (in Section 4.2.2) successfully picks the proper candidates according to marginal benefits they would bring.

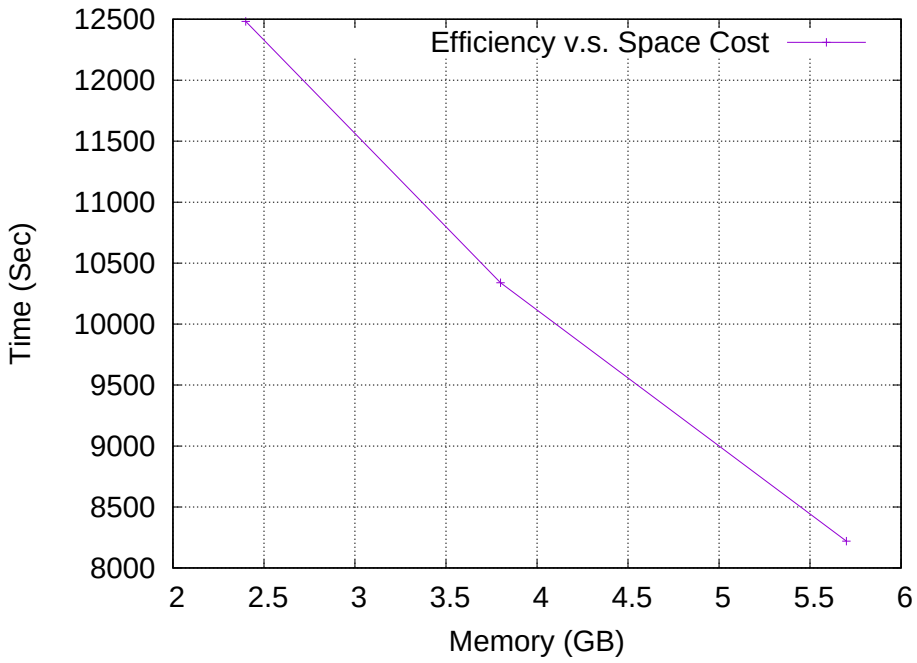

Figure 5.5: Efficiency vs Space Cost

5.3.4 Storage Level for Materialized Views

As listed in Section 5.2, by default, materialized views are stored as objects in main memory. This 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 5.3.4 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 is worth noting that the drop in efficiency is acceptable. Such a drop in efficiency is due to the I/0 overhead in loading materialized views from disk files.

One interesting question is what is the point of storing materialized views in files, since eventually they are to be read into the main memory. Our answer is that in cases when main memory is far too small for holding all selected views, materialization on disk provides an alternative.

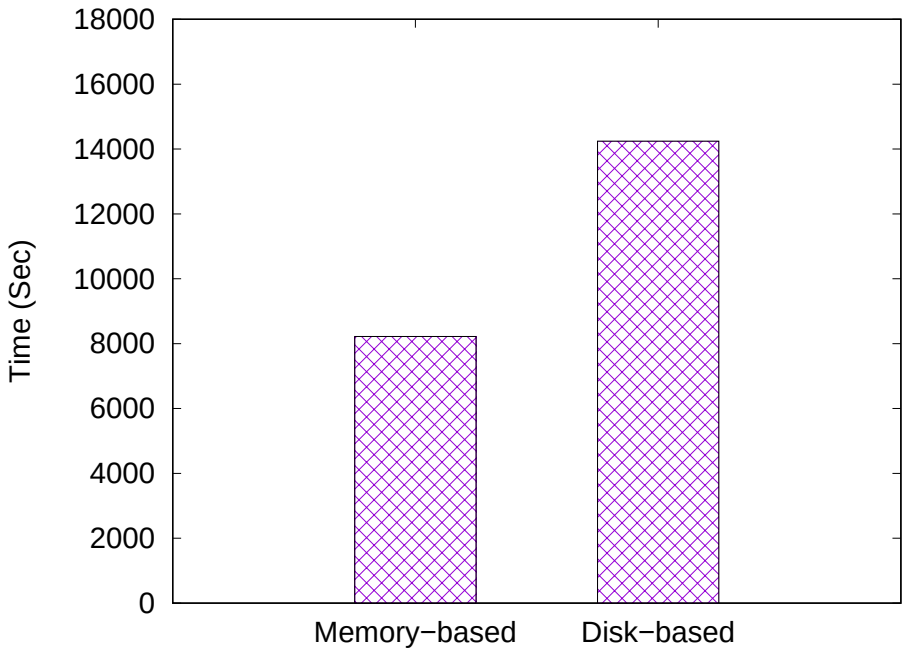

Figure 5.6: Main memory storage vs hard disk storage

5.3.5 CubePlanner vs PMA

We compare CubePlanner in Section 4.2.3 in our solution with PMA in Graph Cube [**?**]. Figures 5.7 and 5.8 show that CubePlanner outperforms PMA in both query processing efficiency and 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}.

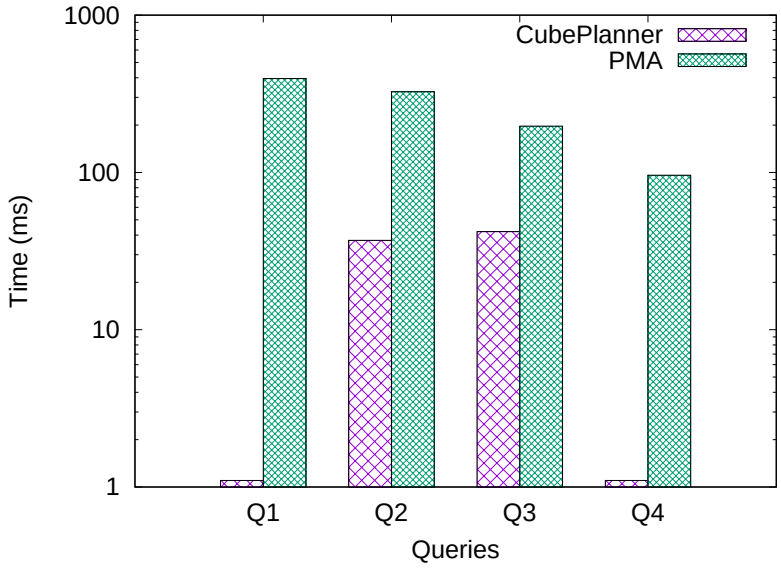

Figure 5.7: Time: CubePlanner vs PMA

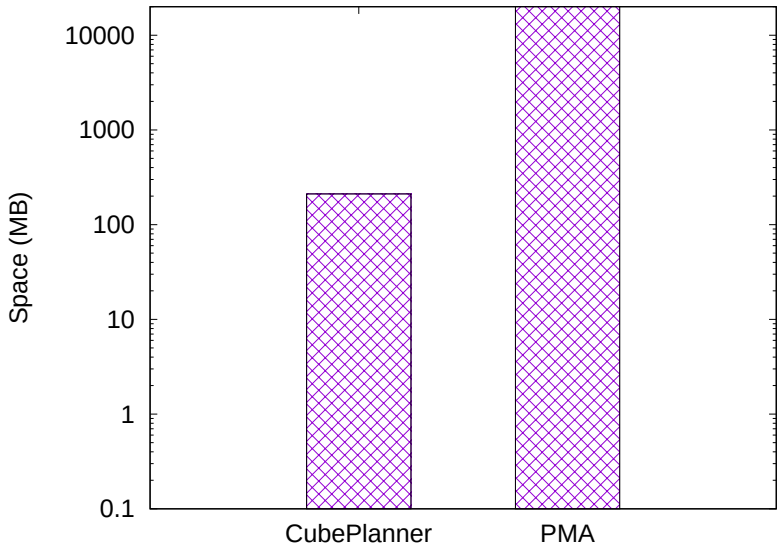

Figure 5.8: Total cuboid space cost: CubePlanner vs PMA

Both two approaches are considered as implementations of the Greedy Selection Framework given in Section 4.2.2. The differences between the two implementations. CubePlan-

ner 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 has 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 vs 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. Figures 5.9 and 5.10 show that our StructurePlanner outperforms FPM in both efficiency and space cost.

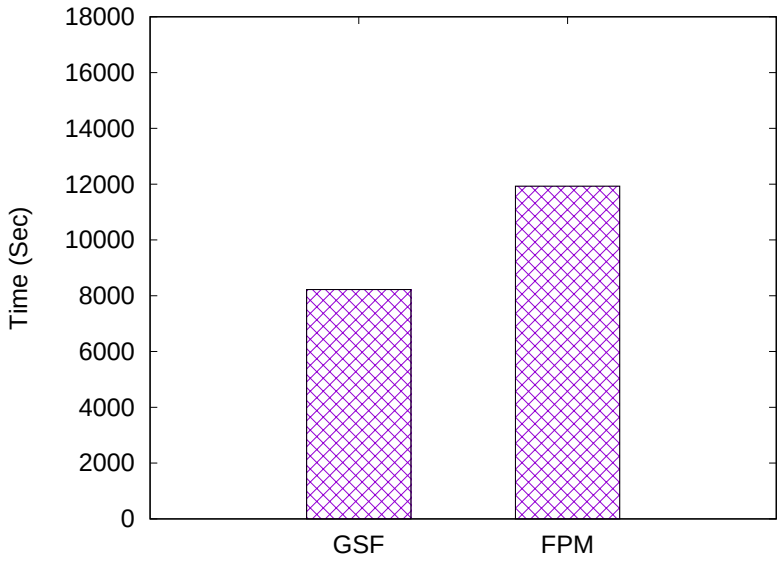

Figure 5.9: Total processing time for future workload: StructurePlanner vs FPM

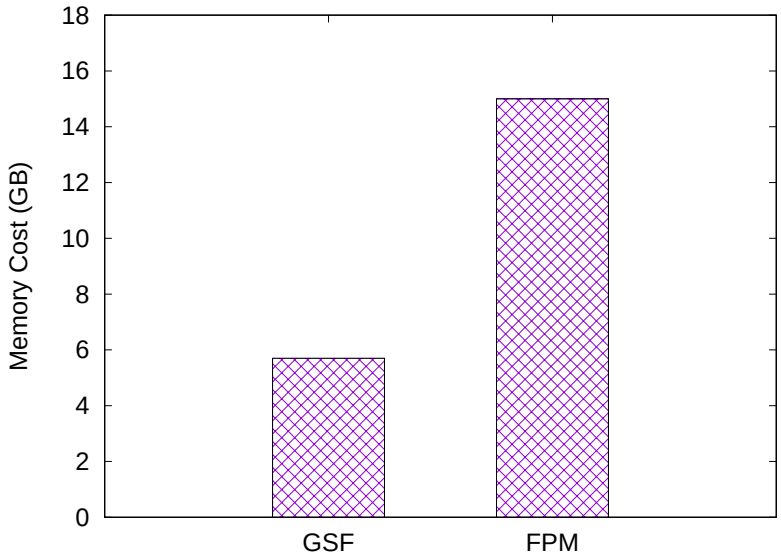

Figure 5.10: Space cost: StructurePlanner vs FPM

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 the previous queries. However, FPM selects *Badge-User, User-Post, Post-Tag*, which 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 5.11 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.

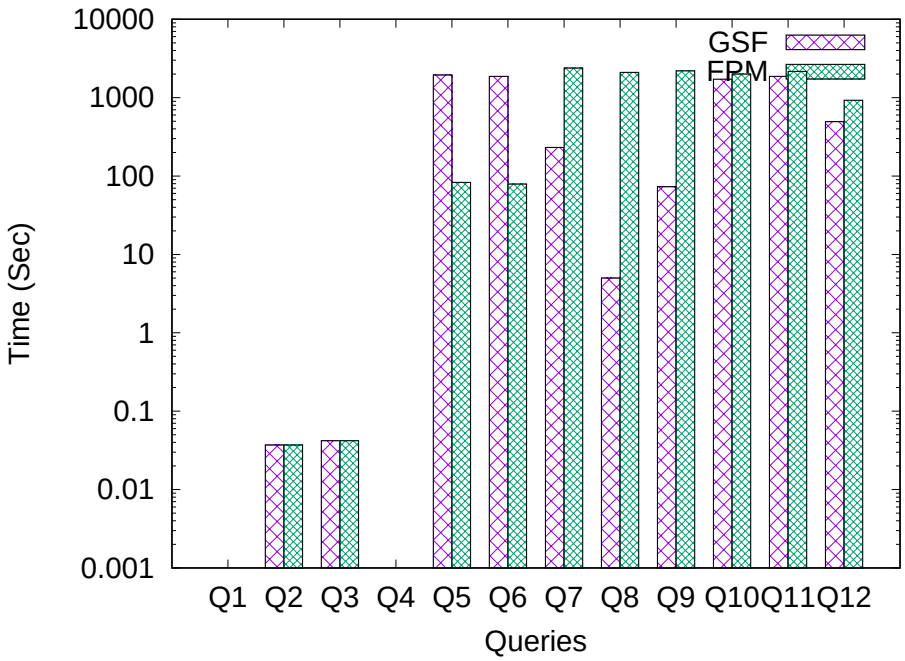

Figure 5.11: Processing time for each query: StructurePlanner vs FPM

5.3.7 Substructure Selection

In our experiment, $h(s)$ in Algorithm 7 does not make any difference during "Substructure Selection". This is because the three selected substructures do not share any edges. 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 *Decompose_Join*, *Decompose_Join** and *Decompose_Join*$^+$ in "Decomposition and Join" (presented in Section 4.3.2). Three different implementations in "Decomposition and Join" would lead to different processing performance for Q10 - Q12, as they are partially covered by S and fetching "complementary components" from Neo4j is necessary. Figure 5.12 provides the processing time for Q10 - Q12 using the three approaches. *Decompose_Join** performs better than *Decompose_Join* in Q10. This is because *Decompose_Join** passes to Neo4j candidate IDs of users who have badges,

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 of *Decompose_Join*, thanks to the "filtering effect". This is reflected in Figure 5.13, 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 5.14 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 Section 4.3.2). Note that the time cost for a "trial query" is bounded by a constant sample size, and is not proportional to actual data size in the dataset. A too small sample size results in bias in estimation, whereas a unnecessarily large sample size causes too much time cost. We tried different sample sizes and find 100 to be an appropriate value, which serves estimation properly and saves the time cost. Figure 5.15 shows that the time cost "trial query" is negligible compared to the overall processing time.

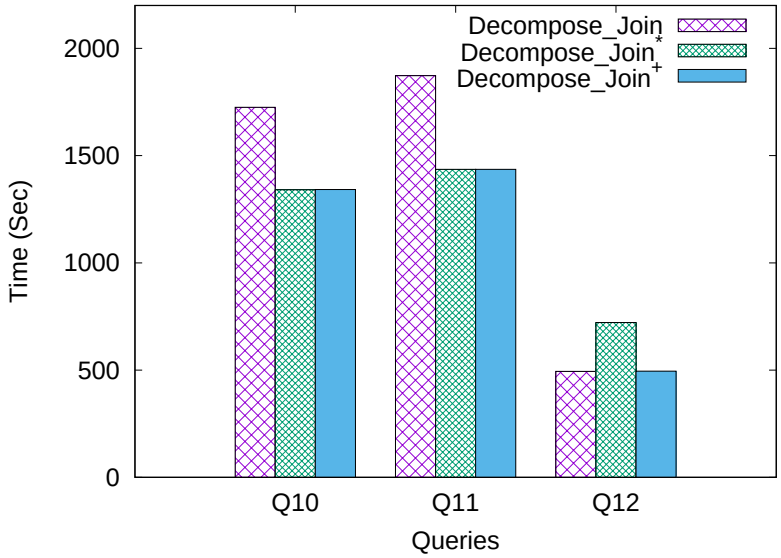

Figure 5.12: Processing time for Q10 - Q12 by three approaches.

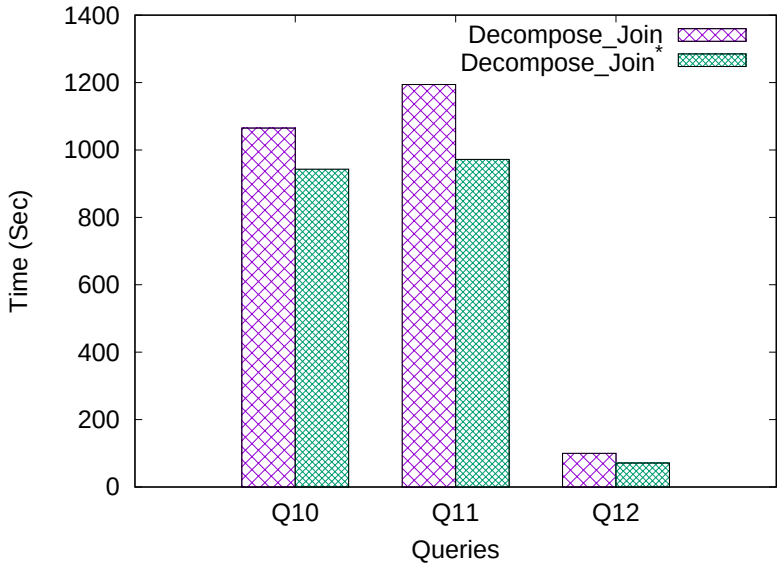

Figure 5.13: Joining time in processing Q10 - Q12 by three approaches.

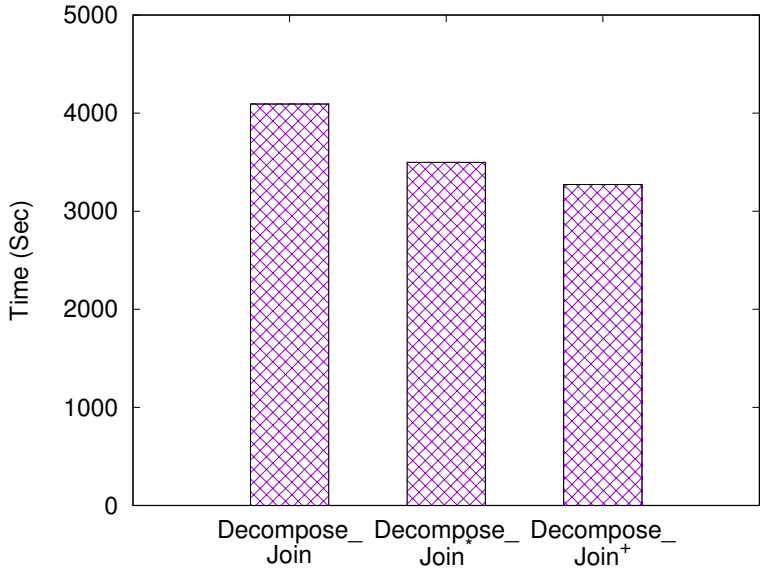

Figure 5.14: Total processing time for Q10 - Q12 by three approaches.

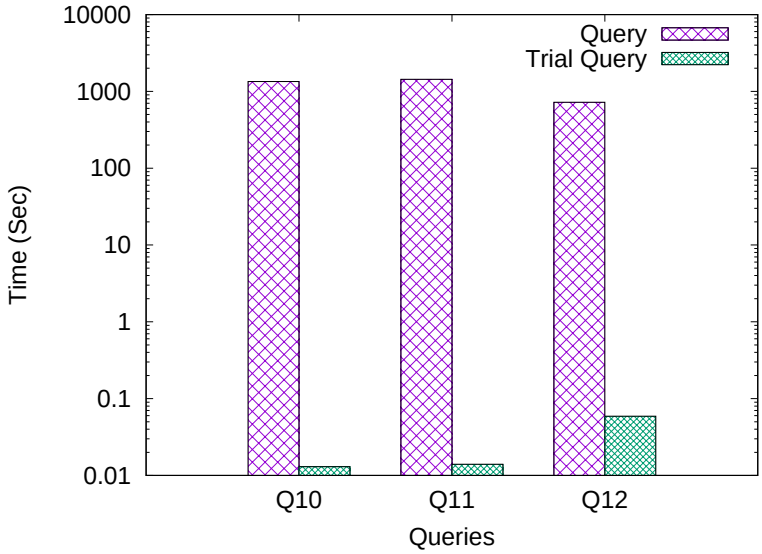

Figure 5.15: Total processing time vs "trial query" processing time.

To conclude, "filtering effect" is an important factor in the 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 Reflections on Neo4j

During the experiments, we found that Neo4j uses a naïve approach to 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 5.16 presents Neo4j's execution plans for queries *User-Post: User.Age* and *User-Post: ID(User), ID(Post)*, respectively. Obviously the latter query should have much larger estimated result size than the former one. However Neo4j returns exactly the same estimated result size by simply taking the square root of table length (216791 estimated rows in "Projection" step) in the previous step.

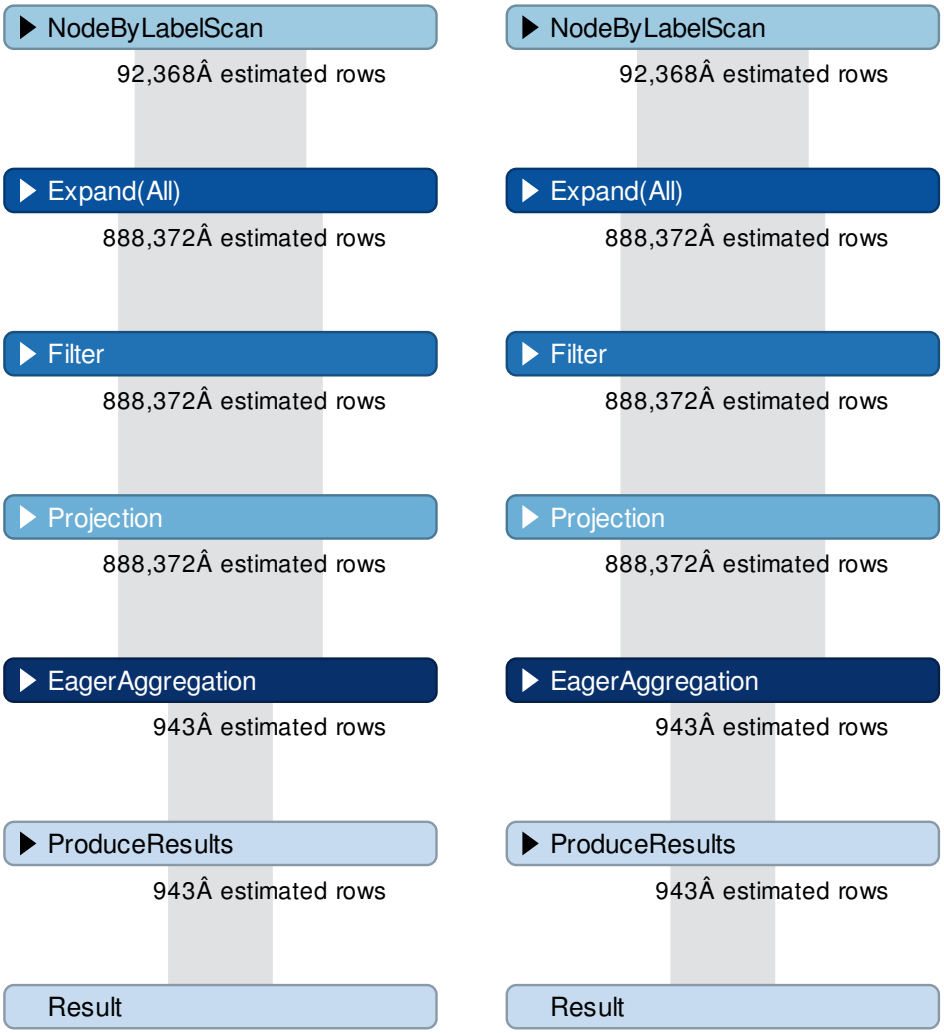

Figure 5.16: Execution plans for *User-Post: User.Age* and *User-Post: ID(User), ID(Post)*.

This is why we use the Cartesian product of dimensions to estimate cuboid sizes in "Single CubePlanner" (covered in Section 4.2.3).

Chapter 6

Conclusion

Our work addresses the urgent need for efficient OLAP processing over large property graphs. We present an end-to-end system to tackle the problem that we integrate into Neo4j. The main idea of our solution is to accelerate future query processing using materialized views that were selected based on their benefits in previous workload. We emphasize structure as a key feature of graph OLAP queries, which is a complement of previous work [**?**]. We introduce and distinguish the concepts of cuboid and substructure, which are two important types of materialized views and should be handled differently. Our solution aims at property-aware and structure-aware materialization by taking both cuboids and substructures into account. To achieve this, a greedy selection framework is proposed for materialized view selection. Besides, we provide three approaches for substructure join and data retrieval from Neo4j, and analyze their advantages and disadvantages. Our solution works for any SPARQL-like query over the schema graph of any property graph, and it is compatible for other graph databases. We conduct experiments by running a set of queries using our system and native Neo4j. Results show that with acceptable space cost in materailization (12.8% of the total size of the dataset), our system acheives 30x speedup over native Neo4j. The experiments demonstrate our system to be a feasible solution that achieves efficient OLAP queries processing on large graph datasets. In addition, we make reflections on Neo4j and find out a defect in its result size estimation for aggregation queries.

There are a number of directions for future work. First, as reflected in experiments in Section 5.3, performance of our system is affected by system settings, hence investigation on appropriate system settings would be a valuable follow-up to our work. Second, the system we implemented so far supports SPARQL like queries over schema graph, instead of data graph. It is possible to execute SPARQL-like queries over data graph within our

proposed solution framework. The key issue is to take isomorphism into consideration during query decomposition and composition. Finally, greedy selection framework is not a perfect solution for "Materialization Selection" problem. More sophisticated solutions can be developed for this hard but valuable problem.

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?