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

Abstract

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

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

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

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

Acknowledgements

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

Dedication

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

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Being a flexible and semantic rich model for graph structured data, the property graph model has been widely adopted and we have seen emerging Graph database systems supporting this model, like Neo4j [?], 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., smart business [?], market analysis [?], trend monitoring [?], risk management [?]. However, empirical studies show that existing graph database systems do not efficiently support OLAP workloads, especially structure wise aggregation queries. Moreover, current graph database systems do not support view-based query or materialize some "hot" intermediate results to serve future queries. Therefore, in this thesis, we study the efficient processing of OLAP queries over property graph data using a materialization approach.

1.1 Property Graph Model

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

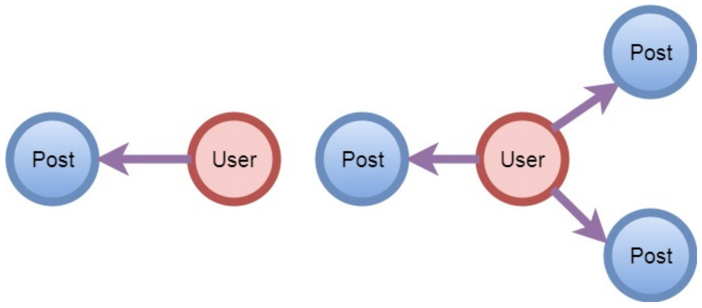


Figure 1.1: A simple property graph modeling "users post posts"(data graph).

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

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

1.2 OLAP over Property Graph

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

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

Get the average post score grouped by users upvotes.

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

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

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

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

1.3 Challenges of Graph OLAP

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

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

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

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

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

1.4 Our Solution and Contributions

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

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

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

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

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

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

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

Chapter 2

Background and Related Work

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

2.1 OLAP over Property Graph Model

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

$$[NodeType].[PropertyType]$$

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

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

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

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

2.1.1 OLAP Examples

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

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

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

Sample query result:

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

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

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

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

Sample query result:

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

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

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

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

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

Sample query result:

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

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

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

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

Sample query result:

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

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

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

2.1.2 Structure, Dimension, and Measure

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

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

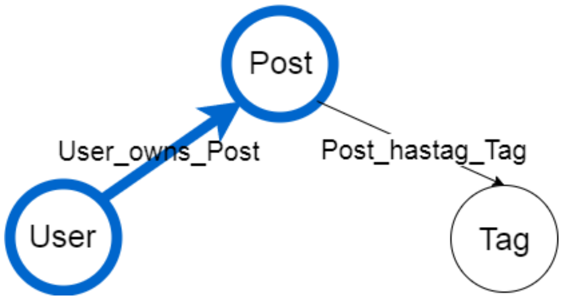


Figure 2.1: *Structure* of Query #1

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

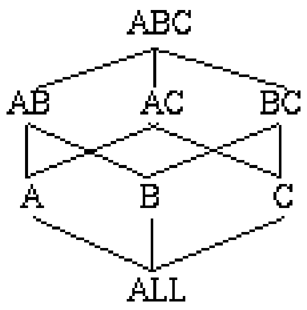


Figure 2.2: Cube of properties {A,B,C}.

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

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

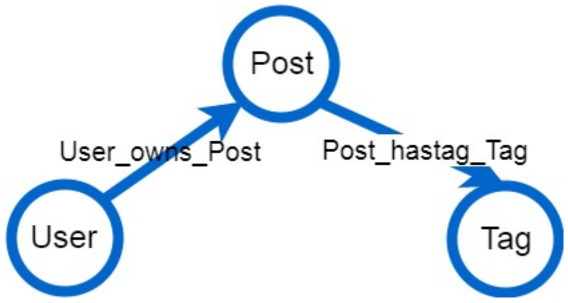


Figure 2.3: *Structure* of Query #3

Dimensions: Tag.Tagname

Measures: AVG(User.Age)

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

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

2.2 Graph Databases and Neo4j

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

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

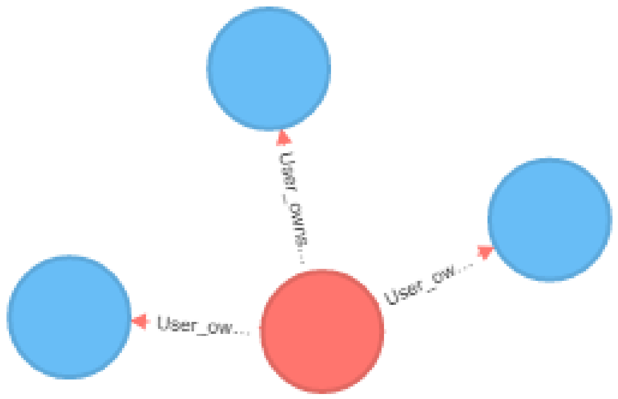


Figure 2.4: A simple property graph.

User		
Uid	Age	UpVote
1	22	5

Post	
Pid	Score
1	0.5
2	0.8
3	0.6

Owns	
Uid	Pid
1	1
1	2
1	3

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

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

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

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

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

2.3 Related Work

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

Cube-based [?] 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 provides a formal definition of Gagg on top of SPARQL Algebra and defines its operational semantics and describe an algorithm to answer graph aggregation queries. Gagg achieves significant improvements in performance compared to plain-SPARQL graph aggregation.

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

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(distributive and holistic etc) by their properties during aggregation. It looks into different semantics of OLAP operations, and classifies the framework into two major subcases: informational OLAP and topological OLAP. It points out a graph cube can be fully or partially materialized by calculating a special kind of measure called aggregated graph.

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

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

	G. Type	Q. Pattern	Layered	Featuer
Cube-based [?]	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

We summarize some of the most related ones as follows:

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

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

Chapter 3

Problem Definition

In this section, we first illustrate the terminology and notations adopted in this thesis. Then we give formal definitions of problems on efficient OLAP query processing.

3.1 Terminologies

We first present definitions and notations of property graph, queries, and materializations. Then we introduce concepts of "cuboid" and "substructure", which are two types of materializations we will use in our solution.

3.1.1 Definition of Property Graph

For clear presentation purpose, we first formal define the property graph model employed in this thesis. We define a property graph as $G(V, Vid, E, Eid, A, L, f)$ where $V = \{v_1, v_2, ..., v_n\}$ is a set of nodes. Vid is a set of unique IDs of V. $E = \{e_1, e_2, ..., e_m\}$ is a set of edges. $E \subseteq V * V$. Eid is a set of unique IDs of E. A is a set of predefined properties. L is a set of predefined labels. f is a mapping function that maps V and E to A, L, Vid and Eid: $f_{VA} : \{v_i \to A_i\}, v_i \in V, A_i \subseteq A$, maps each node to its properties; $f_{VL} : \{v_i \to L_i\}, v_i \in V, L_i \subseteq L$, maps each node to its labels; $f_{EA} : \{e_i \to A_i\}, e_i \in E, L_i \subseteq A$, maps each edge to its properties; $f_{EA} : \{e_i \to L_i\}, e_i \in E, L_i \subseteq L$, maps each edge to its labels; $f_{Vid} : \{v_i \to vid_i\}, v_i \in V, vid_i \subseteq N$, maps each node to its unique ID; $f_{Eid} : \{e_i \to eid_i\}, e_i \in V, eid_i \subseteq N$, maps each edge to its unique ID.

3.1.2 Notations on OLAP Query

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

Structure : A *structure* consists of *edges*. We write a *structure* by listing all its *edges* separated by comma, where an *edge* is represented by

Starting Node Label - Edge Label - Ending Node Label

For instance, Query #3's *structure* as shown in Figure 2.3 is written as

"User-owns-Post, Post-has-Tag"

Dimension: A *Dimension* is written by listing all properties that act as dimensions in an OLAP query.

Query #3's *dimension* is written as *"Tag.Tagname"*.

Measure: We focus on three most common types of *measure*: *COUNT, SUM and AVG.*

Query #3's *measure* is written as *"AVG(User.Age)"*.

Slicing Conditions: A *Slicing Conditions* is written as

Property = value

Query #3's *slicing conditions* is written as *"Post.Year=2017"*.

OLAP query: With the four elements ready, we write an entire OLAP query as

Structure : Dimension, Measure, Slicing Condition

Query #3 is written as

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

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

Features of a query: For a query *q*, we use *"q.properties"* to refer to a set of **all properties** in *Dimension, Measure, and Slicing Condition* of q. We use *"q.structure"* to refer to structure of q.

Query #3.properties={Tag.Tagname, User.Age, Post.Year}

Query #3.structure= User-owns-Post, Post-has-Tag

3.1.3 Materialization: Cuboid vs Substructures

We use $\boxed{\textbf{\textit{\$Query}}}$ to refer to materialization of a *Query*.

As we mentioned before, in a property graph each node and edge has a unique ID, which can be treated as a special property. Whether a materialization keeps unique ID is an important issue. It is because keeping unique ID often increases space cost of a materialization. We categorize two types of materializations, “cuboid” and “substructure”, based on whether unique IDs of nodes and (or) edges are kept or not. To better understand “cuboid” and “substructure”, let’s look at the following example.

Suppose we have following previous workload and future workload:

Previous query #1 User-owns-Post: User.Age
Previous query #2 User-owns-Post: User.Age, (AVG)Post.Score
Future query #1 User-owns-Post: (AVG)User.Age, Post.Score
Future query #2 User-owns-Post, Post-has-Tag: User.Age, Tag.TagName

We can tell that the user is most interested in *User-owns-Post* structure. {User.Age, Post.Score} is the set of properties that are involved in queries over *User-owns-Post* . We can build a cuboid lattice of all combinations of {User.Age, Post.Score}. Materialization of base cuboid of the lattice is

$\boxed{\textit{\$User-owns-Post: User.Age, Post.Score, COUNT(*)}}$

$User-owns-Post: User.Age, Post.Score, COUNT(*) is useful for future query #1. We can process future query #1 by aggregation over *$User-owns-Post: User.Age, Post.Score, COUNT(*)*. We call such materilization a **“cuboid”**.

However, *$User-owns-Post: User.Age, Post.Score, COUNT(*)* is not useful for future query #2. The reason is that they have different *structures*.

If we add *ID(Post)* into *dimension* and materialize *$User-owns-Post: User.Age, Post.Score, ID(Post) COUNT(*)*, *Post* is “activated” to be able to join with other materializations containing *Post* and produce results for OLAP over more complicated *structures*. For instance, future workload #2 can be processed by

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

2.aggregation on {User.Age, Tag.TagName}.

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

	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.

Note that cuboids can only be used in queries with exactly the same structure. They can be be scanned for more aggregated dimension combinations (drill-down operations) but they are 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 space cost versus the potential saving of join processing.***.

Table 3.1 gives a summary of comparisons between “cuboid” and “substructure”.

3.2 Problem Definition

Our target is to faster process future OLAP workload using materializations computed based on previous workload. We can divide our goal in two steps.

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

Materialization step requires us to solve “Materialization Selection Problem”. Query Processing step requires us to solve “Processing Plan Problem”. We give definitions of these two problems as follows.

“Materialization Selection Problem”:

Using materialization is good for query efficiency, but comes with a storage cost. So we want to study the problem of how to best utilize materialization within a space budget limit σ.

We define “Materialization Selection Problem” as

Given a property graph dataset G, a set of previous queries P on G, space limit σ, find cuboids C and substructures S, $\sum_{c_i \in C} c_i.\mathit{space} + \sum_{s_i \in S} s_i.\mathit{space} \leq \sigma$, so that

$\sum_{p_i \in P} T(G, p_i, C, S)$ is minimized.

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

"Processing Plan Problem":

We define "Processing Plan Problem" as

Given a property graph dataset G, a future query q, materialized cuboids C and substructures S, find a processing plan $\mathit{process}(G, q, C, S)$, so that processing time $\mathit{process}(G, q, C, S).\mathit{time}$ is minimized.

In order to answer query q using materializations C and S as fast as possible, we need to solve two questions. First question: **Which** views in C and S shall we use to answer q? Second question: **How** to answer q as fast as possible using selected views in the first question? We formally define the first question as "Decomposition Problem", which decomposes q into views from C and S, and "remaining views" (which are not covered by C and S and need to be fetched from database server). We formally define the second question as "Composition Problem", which performs basic table operations such as join, projection and selection over views in order to generate result of q.

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

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

The reason why we define "Composition Problem" before "Decomposition Problem" is because we need to consider a composition plan $\mathit{compose}(G, q, C', S', R)$ when making our selection policy of C', S' and R. That is to say, “Composition Problem” and “Decomposition Problem” are logically related.

Chapter 4

Solution

4.1 Solution Framework

Our solution framework (Figure 4.1) contains two major parts. “Materialization Part” takes previous workload as input and perform materialization. We first partition previous queries into “hot” queries and “less hot” queries based on frequency count of their structures. CubePlanner and StructurePlanner take “hot” queries and “less hot” queries as input and select cuboids and substructures (in form of tables) for materialization respectively. Subsection 4.2.1 will explain intuition of categorization of “hot” and “less hot” queries and why we pass them to different “planners”. “Future Query Processing Part” takes future queries as input and generate results. If a future query is of “hot” structure we consult cuboid materializations to see if it can be directly answered by aggregation over a cuboid materialization. In this scenario cuboid materialization will be used. If the future query cannot be directly answered by any cuboid materialization, we turn to substructure materializations. We decompose the query into substructures and produce results by “joining” these substructures. In this scenario, substructure materializations will be used.

We will discuss “Materialization Part” in Section 4.2 and “Future Query Processing Part” in Section 4.3.

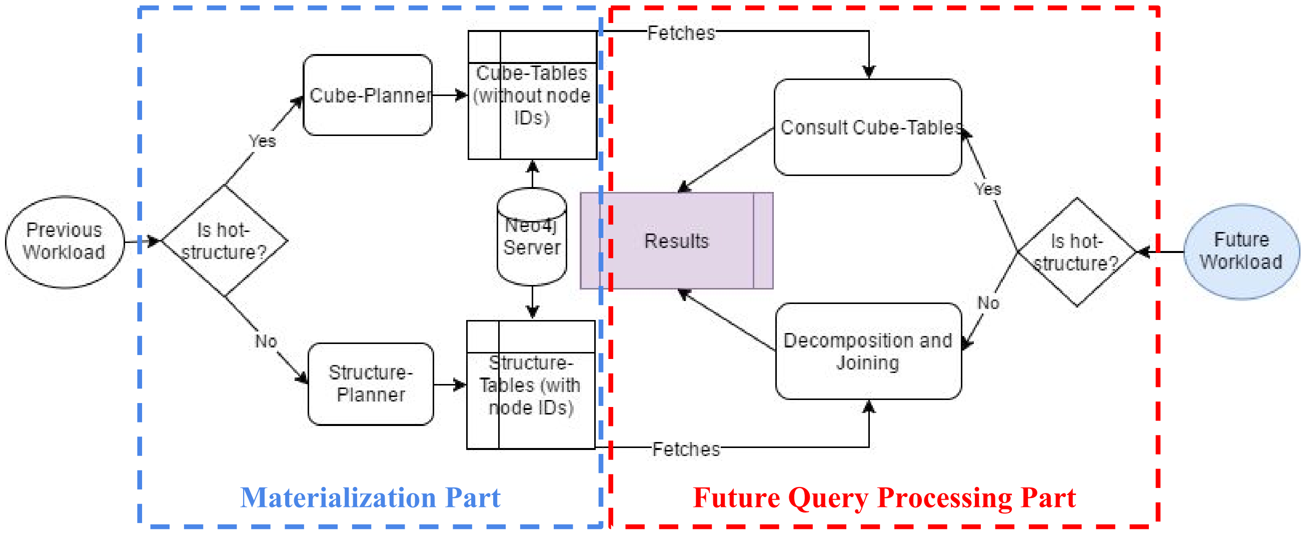


Figure 4.1: Solution framework.

4.2 Materialized View Selection

We will discuss materialized view selection in this section. We will first give an overview of materialized view selection and then focus on cuboid and substructure selections respectively.

4.2.1 Overview of Materialized View Selection

In Section 3.1.3, we have discussed about the trade-off between cuboids and substructures. We know that utilization of a cuboid materialization requires future queries to have exactly the same structure as the cuboid. It is wise that we materialize a cuboid only when we are confident that the structure of a cuboid is likely to be “hit” by future queries, because otherwise we may risk wasting space only to materialize cuboids that are rarely “hit”. Compared with cuboids, substructures do not have such strict “structure match” requirement. A substructure can be used as long as it is covered by a future query.

We make our materialization policy based on such different features of cuboids and substructures. We first perform frequency count of previous queries. For queries of structure frequency over a threshold σ, consider these queries have “hot structure” and pass them to CubePlanner for cuboid selection. For the rest queries with ”less hot structure”, pass them to StructurePlanner for substructure selection.

Algorithm 1: Materialization Overview

```
System setting: $\sigma$: 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  $CInput \leftarrow \emptyset$;
2  $SInput \leftarrow \emptyset$;
3  foreach $q \in Q$ do
4      if $structureFreq(Q, q) > \sigma$ then
5          $CInput \leftarrow CInput \cup \{q\}$;
6      else
7          $SInput \leftarrow SInput \cup \{q\}$;
8      end
9  end
10 $C := materialize(CubePlanner(CInput))$;
11 $S := materialize(StructurePlanner(SInput))$;
```

Function *structureFreq*(Q, q) returns frequency count of q's structure in Q. Functions *CubePlanner* and *StructurePlanner* return selected cuboids and substructures by CubePlanner and StructurePlanner. Function *materialize* performs materialization of cuboids and substructures.

For example, suppose we have the following previous queries and future queries.

Previous Workload:

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

Future Workload:

- #1 User-Comment, Comment-Post: User.UpVotes, (AVG)Post.Score, Post.PostTypeId
- #2 User-Comment, Comment-Post: User.Age, Post.PostTypeId
- #3 User-Post, Post-PostHistory: User.UpVotes, PostHistory.PostHistoryTypeId
- #4 Badge-User, User-Post:(AVG)Post.Score,Post.PostTypeId=2

We count previous queries by structure:

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

We are confident that *User-Comment, Comment-Post* is a "hot structure". We materialize cuboids over structure *User-Comment, Comment-Post* by passing previous query #2, #5 and #6 to CubePlanner. CubePlanner will materialize cuboids that benefit processing of future query #1 and #2 (which have *User-Comment, Comment-Post* structure).

We pass the three remaining queries of "less hot structure" previous query #1, #3 and #4 to StructurePlanner. StructurePlanner will discover and materialize most useful substructures. In this case StructurePlanner is likely to find *User-Post* as a useful substructure it can be used in joining the result of future query #3 and #4.

4.2.2 Greedy Selection Framework

We adopt a greedy selection framework in materialized 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 that "Materialization Selection Problem" aims at finding best materializations under a space limit σ. "Materialization Selection Problem" is known to be an NP-hard problem [**?**]. It is hard because overall benefit of materialized views is not a simple sum of individual benefit of each materialized view. A materialized view's marginal benefit may be deducted when another view is selected. For example, marginal benefit of a substructure over "*User-Post, Post-Tag*" will be affected by selection of substructures over "*User-Post*" and "*Post-Tag*". A naive approach to solve "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 naive may results in an unacceptable time complexity. What's worse, suppose we find an answer C' and S' in a naive way. It is not guaranteed that actual total space cost of C' and S' is strictly lower than σ as we made estimations in our calculation. As a result, 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 until space limit σ is hit.

We will discuss this greedy selection framework first so that readers have a high-level idea of our selection policy. We use greedy algorithms for cuboid and substructure selection. The idea is to always pick next candidate with highest ratio of margin benefit against space. After a candidate is picked, we re-evaluate benefit of remaining candidates. Re-evaluation is essential as margin benefit of a candidate may be deducted owing to materialization of a selected candidate.

Algorithm 2: Greedy Selection

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

We use a queue as data structure for output Q in above algorithm presentation because in some cases we may want to keep information of orders of selection. When selection orders are not important we may as well simply use a set to store selected views. Line 1-5 estimates space cost, marginal benefit for future workload, and score for each candidate. We call this parse **"score calculation"**. Line 6-10 keeps picking up candidates with highest score one by one until space limit is hit. Notice that each time a candidate is selected, Line 9 refreshes scores for all candidates by repeating 1-5. We call this parse **"pick-and-update"**.

CubePlanner and StructurePlanner apply this greedy selection framework by implementation of "score calculation" and "pick-and-update". Future users can vary CubePlanner and StructurePlanner by plug-ins of their own implementation with consideration of their database features. We will introduce how we implement our CubePlanner and StructurePlanner for Neo4j in the following subsections.

4.2.3 CubePlanner

We will discuss CubePlanner in this section. CubePlanner takes "hot" previous queries as input and output selected cuboid materializations. In Subsection 3.1.3, we mentioned that one feature about cuboid is that cuboid are only useful for queries of exactly same structure. To put it another way, cuboids of different structures do not affect each other at all in terms of benefits for future queries. As a result even though 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 algorithm "SingleCubePlanner" to select top-n cuboids. After all groups are finished, we select final results across top-n cuboids of all groups. A good analogy for such process is to first hold regional competitions and then select national winners from regional winners. We will discuss algorithm "CubePlanner" in Subsection 4.2.3, followed by "SingleCubePlanner" explained in Subsection 9.

CubePlanner

As we mentioned above, CubePlanner performs cuboid selection in a holistic manner by one-by-one selection of cuboids from results of SingleCubePlanners.

Algorithm 3: CubePlanner

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

Function *group*(Q) groups Q by structure. *SingleCubePlanner* will be discussed in Subsection 9.

Line 1 partitions Q by structure. Each partition consists of previous queries of a same structure, which will be passed to a SingleCubePlanner. Line 2-4 performs cuboid selection in each partition using SingleCubePlanner. An ordered queue of candidates is generated by each SingleCubePlanner. Line 5-8 repeatedly checks current top candidate for each partition and picks out the best candidate among them. n is a user defined parameter. In our implementation select n at most cuboids for materialization. Users may choose other ways such as a space limit as a bound for cuboid materiliazation.

SingleCubePlanner

Given previous queries of a same structure, we implement algorithm "SingleCubePlanner" from greedy selection framework to select 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: an queue of selected cuboids to precompute
1  $Lattice \leftarrow buildLattice(Q)$;
2  foreach $query\ Q \in P$ do
3      $q.time \leftarrow time(q)$;
4  end
5  foreach $cuboid \in Lattice$ do
6      $cuboid.space \leftarrow space(cuboid)$;
7      $cuboid.benefit \leftarrow 0$;
8      foreach $query\ Q \in P$ and $q.properties \subseteq cuboid.properties$ do
9          $cuboid.benefit += max(0, q.time - aggreTime(cuboid))$;
10     end
11     $cuboid.score \leftarrow cuboid.benefit/cuboid.space$;
12 end
13 for $i=1$ to $n$ do
14     $nextBestCube \leftarrow$ cuboid in Lattice with highest score;
15     if $nextBestCube.score < 0$ then
16         break;
17     end
18     C.Enqueue(nextBestCube);
19     foreach $cuboid\ Q \in Q$ and $q.dimension \subseteq nextBestCube.dimension$ do
20         $q.time \leftarrow min(q.time, aggreTime(nextBestCube))$;
21     end
22     repeat 5-12;
23 end
24
```

Line 1 builds a lattice over all combinations of dimensions of all attributes which appeared in previous queries P using classic lattice construction algorithms [**?**]. Line 2-4 initializes best-so-far processing time for each previous query by its estimated naive database processing time. Line 5-12 performs "score calculation" in "greedy selection framework". For each cuboid, Line 6 estimates its space. Line 8-10 calculates marginal benefit. Line 8 iterates over previous queries that can be answered by scanning current cuboid. If estimated scanning time is less than a previous query's current best-so-far processing time, we add the difference of two times to the cuboid's total marginal benefit (Line 9). Line 13-23 performs "pick-and-update" in "greedy selection framework". Line 15-17 terminates selection when there is no extra marginal benefit any more. Line 19-22 updates best-so-far processing time for previous queries as a result of current round of selection.

Implementation of functions are listed as follows. Notice that users can implement these functions in their own ways based on their database systems. Function $time(query)$ estimates naive time cost for processing a query by a graph database. Implementation of $time(query)$ is database specific as physical storage and execution plans vary among different databases. Since Neo4j provide APIs to see execution plan and estimated intermediate result size, we directly use total size of intermediate results as an estimation of 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 numbers of "estimated rows" for intermediate results are provided. We use $\sum$ "$estimated\_rows''$ to estimate total processing time cost.

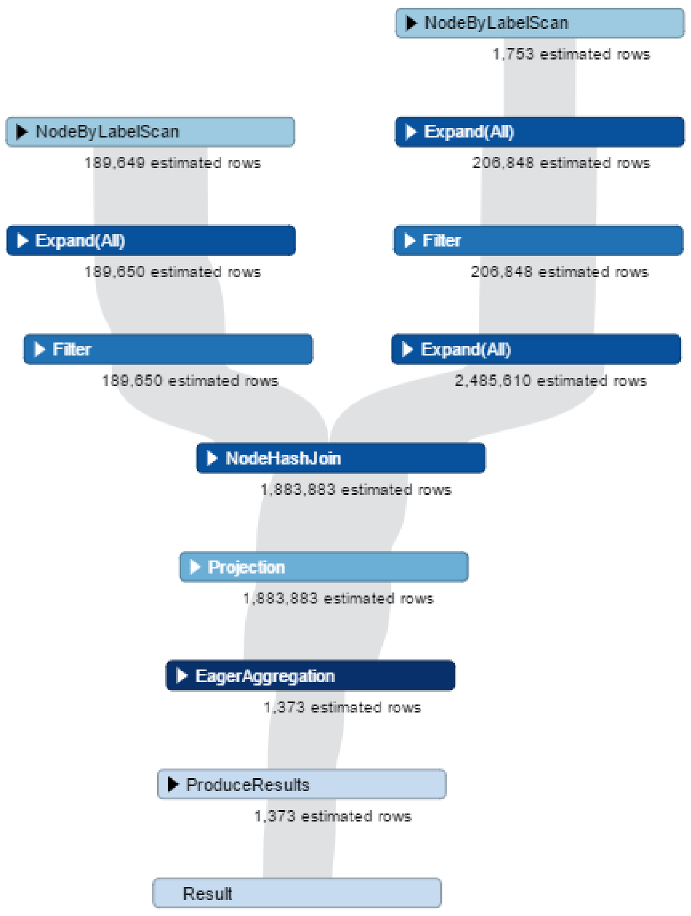


Figure 4.2: Neo4j's execution plan for query *User-Badge, User-Post, Post-Tag: Tag.TagName.*

For graph databases where such APIs to see execution plans and estimated intermediate result sizes are not provided, users need to provide estimation based on their understanding about the database. There are many studies on cost estimations for database operations (joins etc). Users may consider joining (expanding) order [?] and estimation of intermediate result sizes [?] as two important aspects.

Function $aggreTime(cuboid)$ estimates time cost for a scanning cuboid materialization. For cuboid c, we use space cost of c for estimation.

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

$$SpaceCost(c) := spacePerRow * numberOfRows(c)$$

Here *sizeOf(property type)* refers to standard size of data types. For exmaple integer type in “C++” is 2 byte. *numberOfRows(c)* refers to number of rows of *c*. A rough estimation is the product of cardinalities of all queried properties.

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

4.2.4 Structure Planner

Like CubePlanner, Structure Planner also adopts greedy selection framework.

Algorithm 5: StructurePlanner

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

Line 1 builds a lattice over all substructures of previous queries P, using classic lattice construction algorithms (similar to lattice construction algorithms in CubePlanner). Figure 4.3 shows a substructure lattice originating from root node *Badge-User, User-Post, Post-Tag*. Starting from a union of structures of previous queries as the root node, a lattice can be constructed recursively by populating descendants from parent nodes through edge removals.

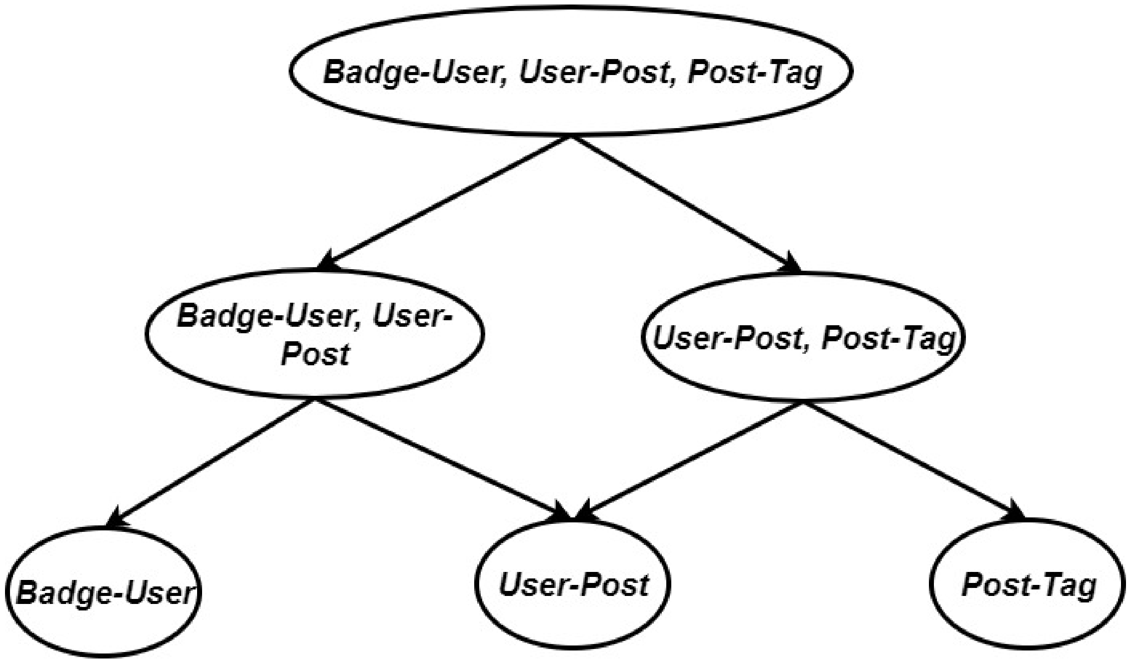


Figure 4.3: A substructure lattice with *Badge-User, User-Post, Post-Tag* as its root node.

Line 2-4 initializes covered substructures for each previous query as empty. For a previous query, *coveredSubstructure* keeps what substructures have been selected so far which are useful for processing this query. It will be updated each time a new substructure is selected. Line 5-12 performs "score calculation" in "greedy selection framework". For each substructure, Line 6 estimates its space. Line 8-10 iterates over all "favored" previous queries (favored by current substructure) and adds on marginal benefit (if any). Here marginal benefit refers to the time saved after adding current substructure to selected substructures (Line 9). Line 13-23 performs "pick-and-update" in "greedy selection framework". Line 15-17 terminates selection when there is no marginal benefit any more. Line 19-22 updates covered substructures for previous queries as a result of current round of selection.

Implementation of functions are listed as follows. Again users can implement these functions in their own ways based on their database systems. Function *space(substructure)* returns estimated space cost of a substructure materialization. We use Neo4j's execution plan API to get estimated result size of a substructure. Function *benefit(q, substructure, q.coveredSubstructres)* evaluates marginal benefit of substructure to query Q when substructures in *q.coveredSubstructres* have been materialized. We know that execution plan and estimated intermediate result size are provided by Neo4j's API. But such information is on database's naive processing plan. When substructure materialization is used, execution plan (intermediate result) becomes different from naive processing plan. As a result estimation on marginal benefit of a substructure is tricky. We use $time(q.coveredSubstructres \cup substructure)$ - $time(q.coveredSubstructres)$ for estimation of marginal benefit of a substructure. We think that this roughly indicates overall improvement of adding *substructure* to *coveredSubstructres* as materializations.

4.2.5 ID and Property Selection

Given a substructure picked by Structure Planner, we need to decide on which IDs and properties should be stored. Keeping all IDs and attributes makes a substructure materialization more informative but increases space cost. We are faced with a trade-off between space cost and usage potential. Selection on IDs and properties is an important issue. We will use substructure *User-Post, Post-Tag* as an example and discuss different ID and property selection policies.

For IDs, we consider the following two policies.

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

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

Policy #2 only enables joins on border nodes. For example we may join *User-Post, Post-Tag* with *User-Badge* on their common border node User.

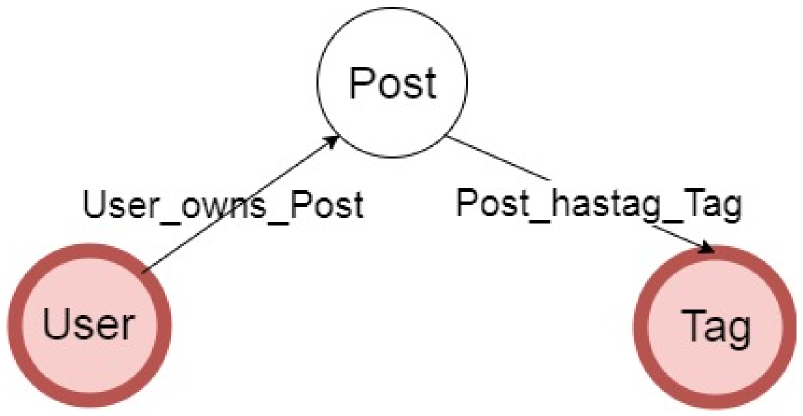


Figure 4.4: "Border nodes" of structure *User-Post, Post-Tag.*

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

For properties, we consider the following two policies.

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

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

4.3 Query Processing

Future Query Processing Part aims at processing future queries efficiently using substructure and cuboid materializations. When future query Q arrives, we first consult cuboids materializations. If Q can be answered by aggregation over any cuboid materializations, we select the cuboid with 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 to compose the result of q.

Algorithm 6: FutureQueryProcessing

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

Line 4-9 looks up materialized cuboids and find if any cuboid can be used to answer query q. If there are multiple useful cuboids we use the cuboid with the smallest scanning cost (*cuboid.space*). Note that *cuboid.space* was computed in Line 9 in Algorithm SingleCubePlanner in Section 9. Line 10 checks if q can be answered by cuboid materializaiton. If yes 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 classic aggregation operation. We will discuss how function $Decompose\_Join(q)$ is implemented in the following subsections.

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 dicision when candidate substructures in S overlap. For example suppose q has structure *Badge-User, User-Post, Post-Tag.*

And S consists of substructures

(1)Badge-User

(2)Badge-User, User-Post

(3)User-Post, Post-Tag

(4)Post-Tag

(5)User-Post.

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

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

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

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

4.3.2 Decomposition and Join

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

#1 *Decompose_Join*

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

Algorithm 8: Decompose_Join

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

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

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

#2 $Decompose\_Join_{informative}$

$Decompose\_Join$ retrieve "complementary components" from database in a naive manner. We adopt the idea of Semi-Join [**?**] and propose anther way of implementation: $Decompose\_Join_{informative}$. Semi-join takes advantage of "selection" effect of natural join. In $Decompose\_Join_{informative}$, we first perform joins over substructures of V. When we retrieve "complementary components" from database server, we inform database server with sets of candidate node and edge IDs as a result of joins over V. We name this approach $Decompose\_Join_{informative}$ as instead of naively query "complementary components" from database, we try to inform database server with sets of candidate IDs. Database backend only needs to search within candidate IDs.

Algorithm 9: $Decompose\_Join_{informative}$

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

Decompose_Join performs joining after "complementary components" are prepared. Unlike *Decompose_Join*, we first join V in Line 3 before retrieval of "complementary components" from databases in Line 7. Note that substructures in V may reside in multiple connected components. Thus $join(V)$ may come to a result of multiple intermediate tables.

In Line 7, $retrieve\_informative(s, V^*)$ fetches results from databases by passing candidate IDs information (from join result V^*). Different database server may have different syntax to achieve this. In SQL we may pass candidate IDs using *WHERE* statement. Neo4j supports query with a list of IDs as arguments in *WHERE* statement.

$Decompose\_Join_{informative}$ **vs.** *Decompose_Join*

Pros: "Informative materialization" helps accelerate retrieval process from backend databases in two aspects. First, since screened out candidate IDs are provided, database backend only needs to iterate through a portion of nodes and edges. This will reduce database processing time. Second, candidate IDs has a filtering effect thus size of retrieval results is likely to be deducted. Thus time of result transmit will be reduced.

Cons: First, $Decompose\_Join_{informative}$ has an transmit overhead of IDs. Second, *Decompose_Join* performs one round of joins after all components are ready. $Decompose\_Join_{informative}$ performs first round of joins on V before without "complementary components" are ready and then second round of joins. In terms of joining orders, *Decompose_Join* is better as its one-round joining order is based on all components with all possible orders of joining.

#3 $Decompose\_Join_{decisive}$

We have mentioned two advantages of $retrieve\_informative(s, V^*)$. However a disadvantage of $retrieve\_informative(s, V^*)$ is an overhead of transport of candidate IDs. We propose a decisive way to evaluate the trade-off between overhead and benefits of

$retrieve\_informative(s, V^*)$ and choose between $retrieve\_informative(s, V^*)$ and $retrieve(s)$.

Algorithm 10: *Decompose_Join*$_{decisive}$

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

In Line 7, Function $decide\_informative(s, V^*)$ makes decision between $retrieve\_informative(s, V^*)$ and $retrieve(s)$. In our implementation we estimate result sizes two retrieval methods: $retrieve\_informative(s, V^*).estimatedSize$ and $retrieve(s).estimatedSize$. $retrieve(s).estimatedSize$ can be returned by space(substructure) in Algorithm "StructurePlanner" in subsection 4.2.4. We calculate $retrieve\_informative(s, V^*).estimatedSize$ in the following way:

1. Randomly sample a small number of candidate IDs.

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

3. Using result length of "trial query", calculate $retrieve\_informative(s, V^*).estimatedSize$ proportionally.

After $retrieve(s).estimatedSize$ and $retrieve\_informative(s, V^*).estimatedSize$ are calculated. We use

$retrieve(s).estimatedSize - retrieve\_informative(s, V^*).estimatedSize / sizeOf(candidateIDs)$ and compare the ratio with a threshold to evaluate trade-off and make decision.

$Decompose\_Join_{decisive}$ **vs.** $Decompose\_Join_{informative}$: We see that $Decompose\_Join_{decisive}$ performs two rounds of joins like $Decompose\_Join_{informative}$. The major difference is that $Decompose\_Join_{decisive}$ plays "trial query". The principle behind "trial query" is to pay an acceptable price of time cost so that we make wise decision on "complementary components" retrieval. A good decision making on "complementary components" retrieval often saves much more time than time cost of "trial queries", especially when dataset is large.

Chapter 5

Experiments

5.1 Experiment Setup

Our main focus is to evaluate different strategies for preprocessing and query evaluation. For instance, the threshold of Is hot-structure part in the diagram, selection policy for materialized substructures in Cube-Planner and Structure-Planner , different heuristics when ranking sub-structures during decomposition in Decomposition and Joining etc.

5.1.1 Datasets

Big StackOverFlow dataset (44.8GB, with 10 different labels on nodes and 12 different types of edges).

Small StackExchange dataset (2.57GB, same schema with Big StackOverFlow dataset).

5.1.2 Query Workloads

48 human-readable meaningful queries are written as a query pool. 24 queries are randomly selected as previous workload while the rest 24 are future workloads. Queries are listed here:

Previous WorkLoad:

User-Comment, Comment-Post: User-UpVotes, Comment-Score, (AVG)Post-Score, Post-PostTypeId=1

User-Comment, Comment-Post: User-Age, (AVG)Comment-Score, Post-PostTypeId=2

User-Comment, Comment-Post: User-ActiveMonth, Post-CreationDate_Year=2016

User-Comment, Comment-Post: (AVG)User-ActiveMonth, Post-CreationDate_Year

Badge-User, User-Post, Post-Tag: Tag-TagName, Badge-Date_Year=2016, Post-CreationDate_Year

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

Badge-User, User-Post, Post-Tag: Tag-TagName, Badge-Name=Student

User-Post, Post-Vote: User-UpVotes, Vote-VoteTypeId

User-Post, Post-Vote: User-Ages, (AVG)Post-Score, Vote-VoteTypeId=1

User-Post, Post-Vote: User-Views, Post-CreationDate_Year=2016, Vote-VoteTypeId

Post-PostLink, Post-Tag: Tag-TagName,Post-CreationDate_Year,

Post-PostTypeId, PostLink-LinkTypeId=3

Post-PostLink, Post-Tag: Tag-TagName, Post-CreationDate_Year

Post-PostLink, Post-Tag: Tag-TagName=database, Post-PostTypeId

Badge-User, User-Post:Badge-Name,Post-Score,Post-PostTypeId=2

Badge-User, User-Post:Badge-Name,(AVG)Post-ActiveMonth,Post-PostTypeId=1

Badge-User, User-Post:Badge-Class, Post-CreationDate_Year

User-Post, Post-Tag: (AVG)User-CreationDate_Year, Tag-TagName

User-Post, Post-Tag: User-CreationDate_Year, (AVG)Post-Score,Tag-TagName

User-Post, Post-Tag: User-Views, (AVG)Post-Score,Tag-TagName

Badge-User: Badge-Name,Badge-Class, Badge-Date_Year

Post-Tag: Post-CreationDate_Year, Tag-TagName

Post-Tag: Post-CreationDate_Year, Tag-TagName

User-Post, Post-PostHistory: User-UpVotes, PostHistory-PostHistoryTypeId

User-Post, Post-PostHistory: User-Age, PostHistory-PostHistoryTypeId=5

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

Badge-User, User-Post:(AVG)Post-Score,Post-PostTypeId=2

User-Post, Post-Tag:User-CreationDate_Year=2016, Tag-TagName

Badge-User, User-Post, Post-Tag: Tag-TagName, Badge-Date_Year=2016

User-Post, Post-Vote: User-Ages, (AVG)Post-Score, Vote-VoteTypeId=2

Post-PostLink, Post-Tag: Tag-TagName, PostLink-LinkTypeId=3

User-Post, Post-PostHistory: User-DownVotes, PostHistory-PostHistoryTypeId

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

Future WorkLoad:

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

User-Post, Post-Vote: User-Views, Vote-VoteTypeId=1

Post-PostLink, Post-Tag: Tag-TagName, Post-PostTypeId=2, PostLink-LinkTypeId

Post-PostLink, Post-Tag: Tag-TagName, (AVG)Post-Score, PostLink-LinkTypeId=1

Post-PostLink, Post-Tag: Tag-TagName, PostLink-LinkTypeId=1

Badge-User, User-Post:Badge-Name, (AVG)Post-Score, Post-PostTypeId

Badge-User, User-Post:(AVG)Badge-Class, Post-CreationDate_Year=2016

Badge-User, User-Post:Badge-Class,(AVG)Post-Score, Post-PostTypeId

User-Post, Post-Tag: User-Age, (AVG)Post-Score,Tag-TagName

User-Post, Post-Tag: User-Views,Post-Score,Tag-TagName

User-Post, Post-PostHistory: User-Age, PostHistory-PostHistoryTypeId

Badge-User, User-Comment: Badge-Class,Comment-Score

Badge-User: Badge-Class, (AVG)User-ActiveMonth, (AVG)User-Age

Post-Tag: (SUM)Post-ActiveMonth, (AVG)Post-Score, Tag-TagName

User-Comment, Comment-Post: User-UpVotes, Comment-Score, (AVG)Post-Score, Post-PostTypeId=2

User-Comment, Comment-Post: User-UpVotes, (AVG)Post-Score, Post-PostTypeId

User-Comment, Comment-Post: User-Age, Post-PostTypeId

User-Comment, Comment-Post: (AVG)User-ActiveMonth, Post-CreationDate_Year=2015

5.1.3 System Setting

We ran the experiments on a Linux cluster machine with 256 GB of memory size.

Our system is implemented in Java.

Initial Java vitual machine memory: 100 GB

Maximum Java vitual machine memory: 200 GB

5.1.4 Neo4j Configuration

Neo4j v4.1.2.

Initial memeroy size: 60GB.

Initial memeroy size: 200GB.

We imported Neo4j's official BOLT driver to interact with Neo4j server. The transport protocol is BOLT protocol(a binary protocal supported by Neo4j).

5.2 Aspects of Interest

Patial Materialization

- Frequency threshold for hot structures.

- Memory limit.

- Selection policy for materialized substructures.

- Comparison with Jiawei Hans algorithm on selecting cuboids.

- Comparison with frequent pattern mining algorithm(FPM) on selecting which substructures to pre-compute.

Future Query Processing

- Different heuristics when ranking sub-structures during decomposition (edges of substructure, Score when selected by Structure-Planner, tuples in the table).

- Different ways of Decomposation_Join(Normal Materializion, Informative Materialization, Decisive Materializaion, Hard Disk Materializaiton).

Dataset Size

- Dataset of different sizes.

5.3 Efficiency Test

5.3.1 Neo4j BaseLine

5.3.2 My System

Precomputation:

- Frequency threshold for hot structures. $\leftarrow$ 5

- Memory limit. $\leftarrow$20GB

- Selection algorithm. $\leftarrow$ My algorithm

Decomposition:

- Different heuristics when ranking sub-structures during decomposition. $\leftarrow$ edges of sub-structure

- Different ways of Decomposation_Join $\leftarrow$ Normal Materializion

5.3.3 Frequency Threshold

5.3.4 Memory Limit

5.3.5 Selection Algorithms

5.3.6 View Selection

5.3.7 Decompose_Join

5.4 Discussion

Chapter 6

Conclusion

6.1 Future Work

We summarize future work as follows:

- Online adaptive

  The system we have implemented is offline. It can be turned into online adaptive one by keeping a sliding window of previous workloads.

- Schema graph to data graph

  Our system currently supports SPARQL like queries over schema graph instead of data graph. It could be further improved to support queries over data graph without changing the high-level solution framework. The key part that needs to be modified is to label each unique node and take isomorphism into consideration during query decomposition.

- Better Cube-Planner and Structure-Planner

  We used greedy approach for ranking cuboids and substructures. Although it worked well in our experiment. But greedy approach is not holistic enough. For instance???

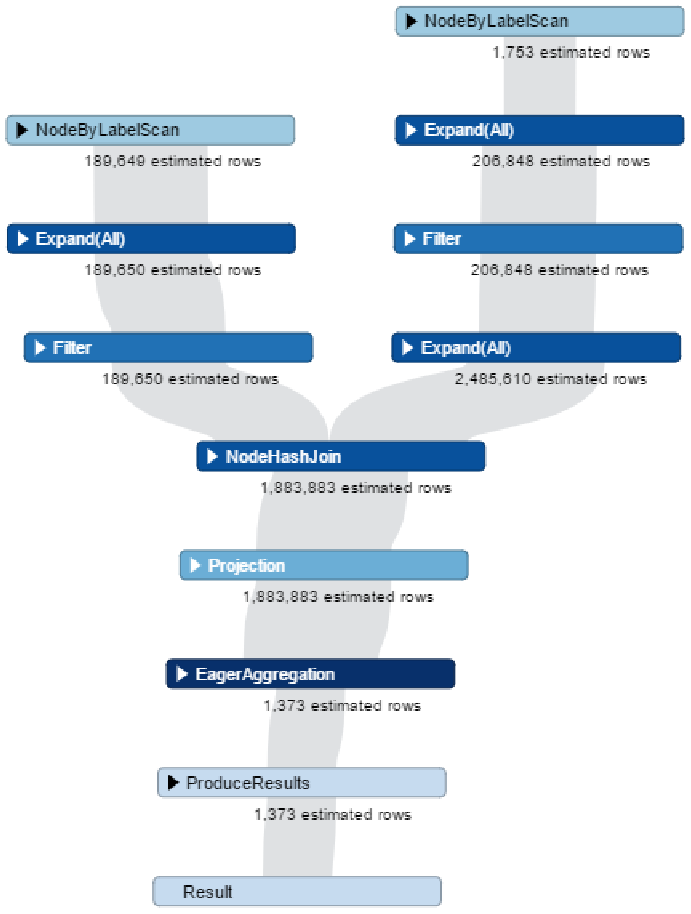
- Multi-Thread implementation

  The system can be made multi-thread so that joining work of queries could be done when the system is waiting for graph databases query execution.

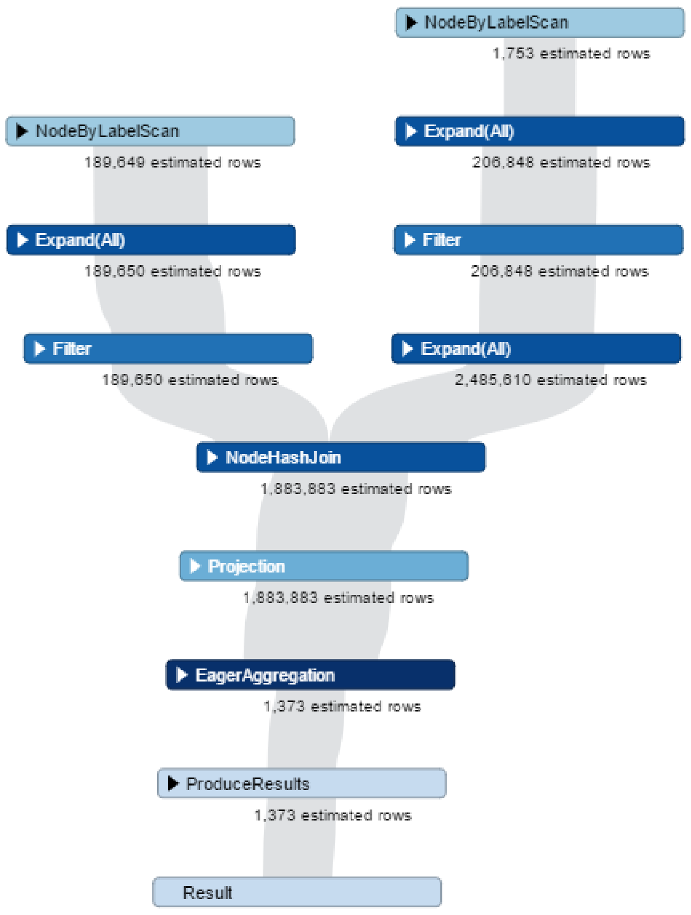
6.2 Reflection on Neo4j

6.2.1 Aggregation Size Estimation

We found that Neo4j has a very coarse way of estimating result size of aggregation queries. It simply takes square root of table size before aggregation, without regards to aggregation attributes. Of course this will lead to a huge bias. For instance, lets look at the following 2 queries with the same structure: (1) match (u:User)-[]-(b:Badge) match (u:User)-[]-(p:Post) match (p:Post)-[]-(t:Tag) return t.TagName, count(*)



(2) match (u:User)-[]-(b:Badge) match (u:User)-[]-(p:Post) match (p:Post)-[]-(t:Tag) return t.TagName, id(u), id(b), id(p), count(*)



Since (2) contains ids of all queried nodes(User, Badge and Post), supposedly (2) should have a much larger result size than (1). However in Neo4j will estimate that (1) and (2) have the same result size. Therefore in our implementation we use the following function to predict cuboid size: Cuiboid(att$_1, att_2att_n) = Productof(|att_i|) * (shrinkingfactor)^{(}n - 1)$

References

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

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

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

APPENDICES

Appendix A

Matlab Code for Making a PDF Plot

A.1 Using the GUI

Properties of Matab plots can be adjusted from the plot window via a graphical interface. Under the Desktop menu in the Figure window, select the Property Editor. You may also want to check the Plot Browser and Figure Palette for more tools. To adjust properties of the axes, look under the Edit menu and select Axes Properties.

To set the figure size and to save as PDF or other file formats, click the Export Setup button in the figure Property Editor.

A.2 From the Command Line

All figure properties can also be manipulated from the command line. Here's an example:

```
x=[0:0.1:pi];
hold on % Plot multiple traces on one figure
plot(x,sin(x))
plot(x,cos(x),'--r')
plot(x,tan(x),'.-g')
title('Some Trig Functions Over 0 to \pi') % Note LaTeX markup!
```

```
legend('{\it sin}(x)','{\it cos}(x)','{\it tan}(x)')
hold off
set(gca,'Ylim',[-3 3]) % Adjust Y limits of "current axes"
set(gcf,'Units','inches') % Set figure size units of "current figure"
set(gcf,'Position',[0,0,6,4]) % Set figure width (6 in.) and height (4 in.)
cd n:\thesis\plots % Select where to save
print -dpdf plot.pdf % Save as PDF
```