# Efficient Structure-aware OLAP Query Processing over Large Property Graphs

by

Yan Zhang

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

Waterloo, Ontario, Canada, 2017

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

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

## Abstract

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

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

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

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

# Acknowledgements

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

## Dedication

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

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Being a flexible and semantic rich model for graph structured data, the property graph model has been widely adopted and we have seen emerging Graph database systems supporting this model, like Neo4j [?], 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.

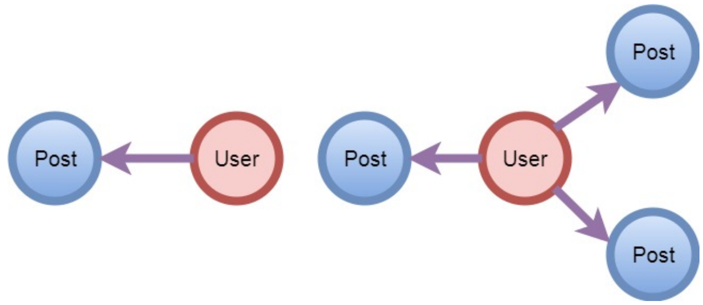



Figure 1.1: A simple property graph modeling "users post posts"(data graph).

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

Among various kinds of queries, OLAP (Online analytical processing) queries play an important role in data analysis.

Add a few sentences to describe OLAP queries of the traditional database and warehousing applications. Showing it is important and we say performing OLAP on property graph is desired.

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

Here you should start with a paragraph saying "Supporing efficient OLAP is 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."

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.

Now you should explain what you can learn from previous workloads, and how these knowledge can help you with future workloads.

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.

Then, you need a paragraph that describe the actual challenge. Because so far you only mentioned that using something from history workload is helpful, then the problem is how to extract these information, how to decide which information should be materialized espeically when there is a space constraint.

We know that materialization on user’s interested structures and properties results in benefit on future workload processing efficiency , and cost on memory space. 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 precompute 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.

here you need to use a paragraph to explain how you select materialized views, e.g., you develop a cost function. Plus, you have a scheduling policy to execute subqueries in the most time efficient way. This part is particularly important, because reader need to briefly understand you overall solution from a high level perspective.

For 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 issue, 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 section, 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).

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:

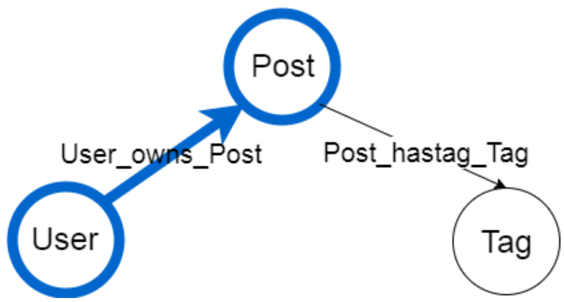



Figure 2.1: *Structure* of Query #1

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.

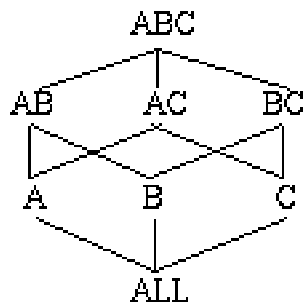



Figure 2.2: Cube of properties {A,B,C}.

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

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

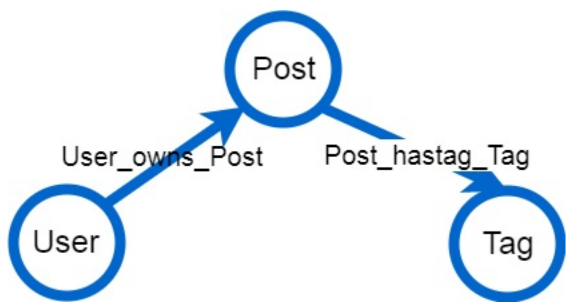



Figure 2.3: *Structure* of Query #3

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:

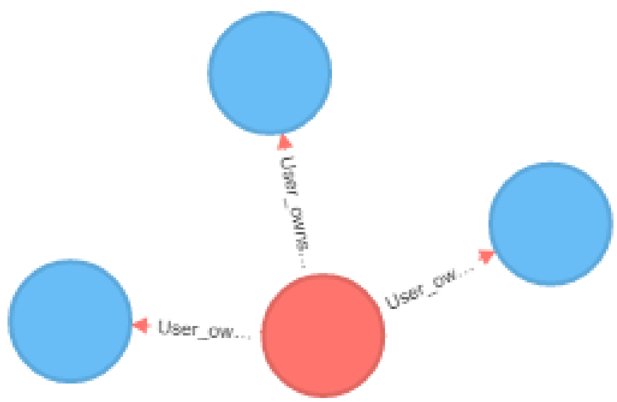



Figure 2.4: A simple property graph.

| 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 work. Then we formally define the problem of efficient OLAP query processing.

## 3.1 Terminologies

### 3.1.1 Property Graph

We define a property graph as $G(V, E, A, L, f)$ where

- $V$ is a set of nodes.
- $E$ is a set of edges. $E \subseteq V * V$.
- $A$ is a set of properties.
- $L$ is a set of labels.
- $f$ is mapping function that maps $V$ and $E$ to $A$ and $L$.
  $f_{VA} : \{V_i \to A_i\}, V_i \in V, A_i \subseteq A$
  $f_{VL} : \{V_i \to L_i\}, V_i \in V, L_i \subseteq L$
  $f_{EA} : \{E_i \to A_i\}, E_i \in E, L_i \subseteq A$
  $f_{EA} : \{E_i \to L_i\}, E_i \in E, L_i \subseteq L$

### 3.1.2 Notations for Query and Materiailization

As discussed before, Four elements of a graph OLAP query are *Structure*, *Dimension*, *Measure*, and *Slicing Condition*(optional). In this thesis, we use the following form to represent an OLAP query.

***Structure : Dimension, Measure, Slicing Condition***

A *strucuture* is written by listing all its *edges* seperated by comma.

An *edge* is represented by

$LabelofStartingNode - LabelofEdge - LabelofEndingNode$. For instance, "*User-owns-Post*" represents edge with label *owns* starting from node with label *User* to node with label *Post*.

For instance Query #3

"Get average user age grouped by users 2017 posts tags"

is written as

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

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

For a query q, we use q.properties to refer to a set of **all properties** in Dimension, Measure, and Slicing Condition of q. Suppose q is Query #3, then

q.properties={Tag.Tagname, User.Age, Post.Year}.

Similarly,

q.structure= User-owns-Post, Post-has-Tag.

We want to accerlerate OLAP by materializations from previous workload. We use

***$Query***

to refer to materializaiton of a *Query*.

### 3.1.3 Cuboid vs Substructures

Suppose we have the following 3 queries in previous workload:

1. User-owns-Post: User.Age
2. User-owns-Post: User.Age, (AVG)Post.Score
3. 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 over {User.Age, Post.Score}.

Intuitively, $User-owns-Post: User.Age, Post.Score serves as a good materailization for future workload.

Suppose we build a cube as in Graph Cube [4] on User-owns-Post, Post-has-Tag over all queried attributes and precompute some most beneficial cuboids.

For instance we precompute cuboids like

User-owns-Post, Post-has-Tag: User.Age, User.UpVotes, Tag.TagName, User.CreationDate.Year, Post.Score Cuboids only contain attributes.

They do not contain IDs of nodes. 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). Notice that cuboids are not useful for queries with different structures.

We cannot process

Badge-grantedTo-User, User-owns-Post, Post-has-Tag: Tag.TagName, Badge.Name=Teacher

by simply joining cuiboid

User-owns-Post, Post-has-Tag: User.Age, User.UpVotes, Tag.TagName, User.CreationDate.Year, Post.Score

because IDs of User are not provided in the cuboid.

If besides properties, we also keep IDs of User, then the materilization is joinable on User. The drawback is that result would be much more space-costly than cuboids, as ID of User is unique key.

We call materializations that only aggregates on properties without storing IDs cuboids, and we call those with IDs as substructures. The trade-off between cuboids and substructures is space vs usage potential.

Cuboids are generally ligher in terms of space cost, but they can help with queries with exactly same structure.

Substructures generally heavier in terms of space cost, but it can be widely used to join with other substructures.

## 3.2 Problem Definition

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 $\sigma$.

We define our problem as

Given previous queires P, space limit $\sigma$, which cuboids C and substructures S shall we materialize so that future queries F could be faster processed using C and S?

Given P, space limit $\sigma$, select C and S wisely.

Given F, C and S, process F fast.

# Chapter 4

# Solution

## 4.1 Solution Framework

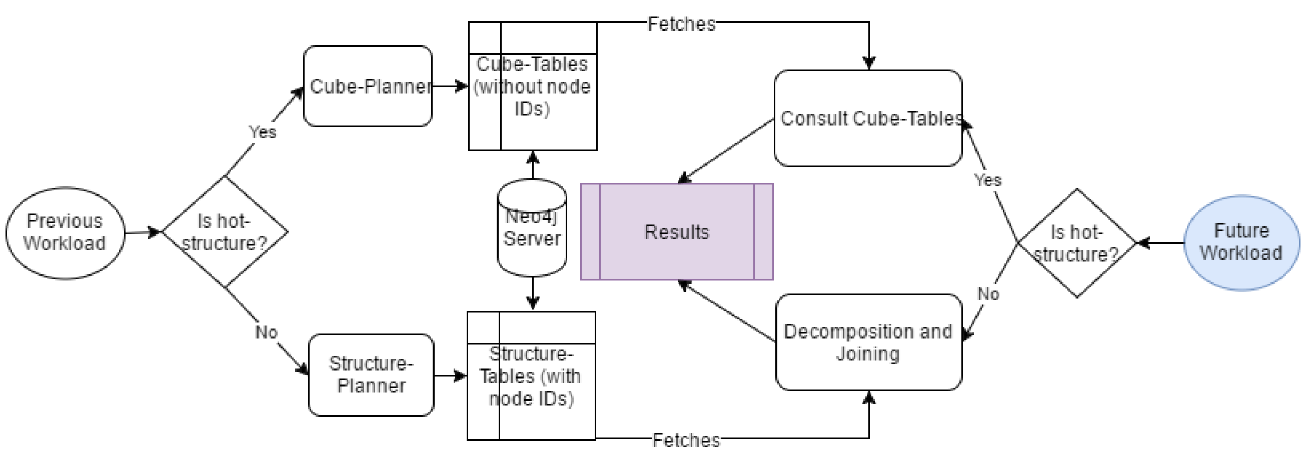



Figure 4.1: Solution framework.

Our solution framework contains two major parts:

Partial Materialization part: Cube-Planner and Structure-Planner select cuboids and substructures to precompute.

Decomposition part: Scan over cuboids or join using substructures to produce results.

We will discuss Partial Materialization part in Section 4.2 and Decomposition part in Section 4.3.

## 4.2 Partial Materialization

In 3.1.2, we have discussed about the trade-off between cuboid and substructures. We know that benefit of a cuboid rely on future queries of exactly same structure. Therefore it is wize that we materizalize cuboids on a structure only when we are fully confident that the structure is of high interest for the client. Otherwize, we may risk wasting space only to materialize cuboids that are rarely "hit" by future queries. On the other hand, substructures are less picky in terms of exact match of future query structures. Therefore we make our partial materialization policy as follows:

For queries of structure frequency over a threshold, consider these queries have "hot structure" and pass them to CubePlanner for cuboid selection.

For other queries with "less hot structure", pass them to StructurePlanner for substructure selection.

**Algorithm 1:** PartialMaterialization

**System setting:** threshold: frequency threshold for hot structures
**Input:** Q: a set of previous queries
**Output:** C: a set of materialized cuboids
S: a set of materialized substructures

```
CInput ← ∅ ;
SInput ← ∅ ;
foreach q ∈ Q do
    if structureFreq(Q, q) > threshold then
        CInput ← CInput ∪ {q};
    else
        SInput ← SInput ∪ {q};
    end
end
C := materialize(CubePlanner(CInput)) ;
S := materialize(StructurePlanner(SInput));
```

For instance:

**Previous Workload:**

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

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

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

User-Post, Post-Tag: (AVG)User-CreationDate$_{Y}ear, Tag - TagName$

User-Comment, Comment-Post: User-ActiveMonth, Post-CreationDate$_{Y}ear = 2016$

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

**Future Workload:**

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

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

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

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 materialze cuboids over *User-Comment, Comment-Post* using

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

User-Comment, Comment-Post: User-ActiveMonth, Post-CreationDate$_{Y}ear = 2016$

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

and hopefully these cuboids benefit processing of future queries

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

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

We pass the rest three queries of "less hot structure"

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

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

User-Post, Post-Tag: (AVG)User-CreationDate$_Year, Tag - TagName$

to StructurePlanner. StructurePlanner will discover most useful substructures. For instance StructurePlanner is likely to find

*User-Post*

as a useful substructure and hopefully the materialized substructure can be used in faster joining the result of

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

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

### 4.2.1 Greedy Selection Framework

Cube-Planner and Structure-Planner adopt the same greedy selection framework. We will discuss this greedy selection framework first so that readers could have a high-level idea of our selection policy.

Our problem is

given previous queires P, space limit $\sigma$, select cuboids C and substructures S to materialize so that future workload F processing could be mostly benefited. Suppose P and F consist of similar queries.

We used greedy algorithms for cuboid and substructure selection. The idea is to always pick next candidate with highest ratio of margin benefit/space. After a candidate is picked, re-evaluate benefit of the rest candidates. Re-evaluation is essential as margin benefit of a candidate may be influenced after materializaion of another 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 ∈ C do
2      c.space := estimateSpace(c) ;
3      c.benefit := estimateMarginBenefit(c, P, Q) ;
4      c.score := c.benefit/c.space ;
5  end
6  for Q.totalsize < σ do
7      selected := c in C with highest score ;
8      Q.offer(selected);
9      repeat 1-5 ;
10 end
11
```

1-5 estimates space cost, marginal benefit for future workload, and score for each candidate. We call this parse "score calculation".

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, 9 refreshes scores for all candidates by repeating 1-5. We call this parse "pick-and-update".

Cube-Planner and Structure-Planner apply this greedy selection framework with specific implementation of score caculation in 1-5. Future users can plug-in their implementation and design their planners with consideration of their database settings. We will introduce how we implement our Cube-Planner and Structure-Planner for Neo4j in the following sections.

### 4.2.2 Cuboid Planner

#### Single Cube

Given previous queries of a same structure, we implement SingleCubePlanner from greedy selection framework to select cuboids.

**Algorithm 3:** SingleCubePlanner

**System setting:** n: maximum number of cuboids to precompute
**Input:** P: a set of previous queries with a same structure
**Output:** C: an queue of selected cuboids to precompute

```
1  Lattice ← buildLattice(Q);
2  foreach query q ∈ P do
3      q.time ← estimateProcessingTime(q) ;
4  end
5  foreach cuboid ∈ Lattice do
6      cuboid.space ← estimateSpace(cuboid);
7      cuboid.benefit ← 0;
8      foreach query q ∈ P and q.properties ⊆ cuboid.properties do
9          cuboid.benefit+ = max(0, q.time − estimateScanningTime(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.offer(nextBestCube);
19     foreach cuboid q ∈ Q and q.dimension ⊆ nextBestCube.dimension do
20         q.time ← min(q.time, estimateScanningTime(nextBestCube));
21     end
22     repeat 5-12 ;
23 end
24
```

1 builds a lattice over all conbinations of dimensions of all attributes that appeared in previous queries P, using classic lattice construction algorithms.

2-4 initializes best-so-far processing time for each previous query by its naive database processing time.

5-12 performs "score calculation". For each cuboid, 6 estimates its space. 8-10 iterates over all "roll-up" previous queries adds on marginal benefit if scanning time over the cuboid is smaller than a previous "roll-up" query's best-so-far processing time.

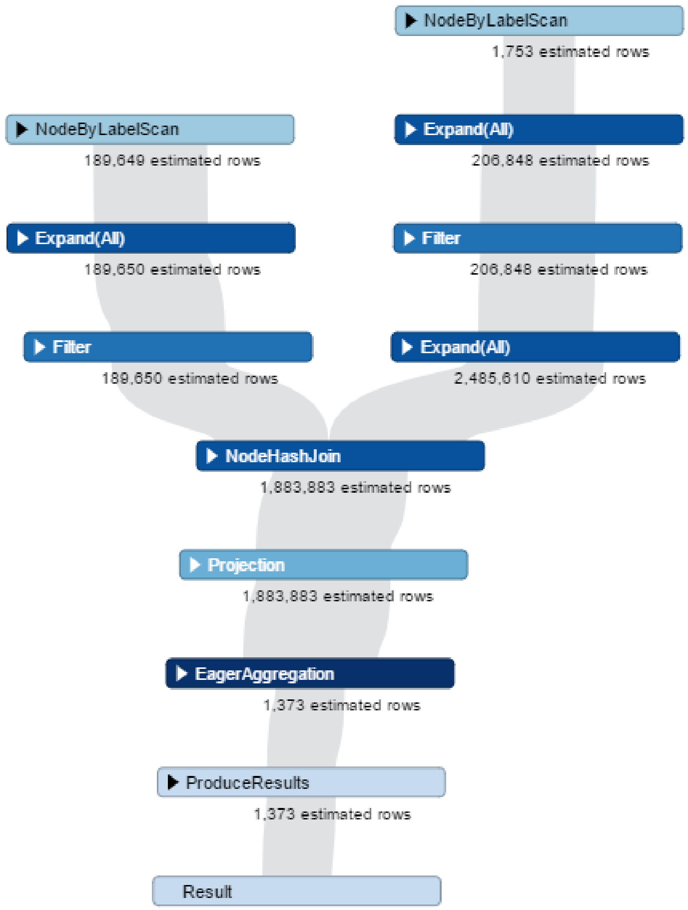
13-23 performs "pick-and-update". 15-17 terminates selection when there is no marginal benefit at all. 19-22 updates best-so-far processing time for previous queries as a result of current round of selection.

Implementation of functions are listed as follows. Notice that users can implement in their own ways based on their database systems.

Function estimateProcessingTime(query) provides naive estimated time cost of processing a query with a graph database. Implementation of estimateProcessingTime(query) is database specific as physical storage execution plans vary among different databases. Some graph databases like Neo4j provide APIs to see the execution plan and estimated intermediate size. In our implementation we used total size of intermediate results as estimation of time cost.

Execution plan of Cypher query:

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




If APIs to see the execution plan and estimated intermediate result sizes are not provided for graph databases, we need to estimate in our own way. There are many studies about joining cost estimation. The key issue is to determine joining order and estimate intermediate result sizes using selectivity. Joining estimation [6] is a good summary of different joining plans and ways of cost estimation.

Function estimateScanningTime(cuboid) estimates time cost for scanning cuboid result. For cuboid C. We use space cost of cuboid result table 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 instance int type in C is 2 byte.

numberOfRows(C) refers to number of rows of C. A rough estimation is product of candinalities of each property. We added a shrinking effect because some conbinations of property values do not exsist in the final result.

$$numberOfRows(C) := \prod_{p \in C.properties} |p| * shrinking\_factor^{|C.properties|-1}$$

### Holistic Cube

SingleCubePlanner selects cuboids from one lattice of one structure. However the input to CubePlanner may consist of previous queries of various hot structures. CubePlanner performs cuboid selection in a holistic manner by calling SingleCubePlanner for each hot structure and rank cuboids across different lattices(cubes).

**Algorithm 4:** CubePlanner

**System setting:** n: 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:= GroupByStructure(Q) ;
**foreach** *group* $\in$ *Group* **do**
    *group.candidates := SingleCubePlanner(group);*
**end**
**for** *i=1* **to** *n* **do**
    *selectedGroup := group in Group with highest group.candidates.top().score ;*
    *C.offer(selectedGroup.poll());*
**end**

1 partitions Q by structure. Each partition consists of previous queries of a same structure, which could be passed to SingleCubePlanner.

2-4 performs cuboid selection in each partition(cube). An ordered queue of candidates is generated for each partition(cube).

5-8 iteratively checks top candidate for each partition(cube) and picks out the best candidate among them.

### 4.2.3 Structure Planner

**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 ← buildSubstuctureLattice(Q);
2  foreach q ∈ Q do
3  |   q.coveredSubstructre := ∅;
4  end
5  foreach substructure ∈ Lattice do
6  |   substructure.space ← estimateSpace(substructure);
7  |   substructure.benefit ← 0;
8  |   foreach q ∈ Q and q.structure ⊆ substructure.structure do
9  |   |   cuboid.benefit+ = max(0, benefit(q, substructure, q.coveredSubstructre));
10 |   end
11 |   substructure.score ← substructure.benefit/substructure.space ;
12 end
13 for i=1 to n 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.coveredSubstructre ← q.coveredSubstructre ∪ {nextBestSubstructre};
21 |   end
22 |   repeat 5-12 ;
23 end
24
```

1 builds a lattice over all substructures are covered in previous queries P, using classic lattice construction algorithms.

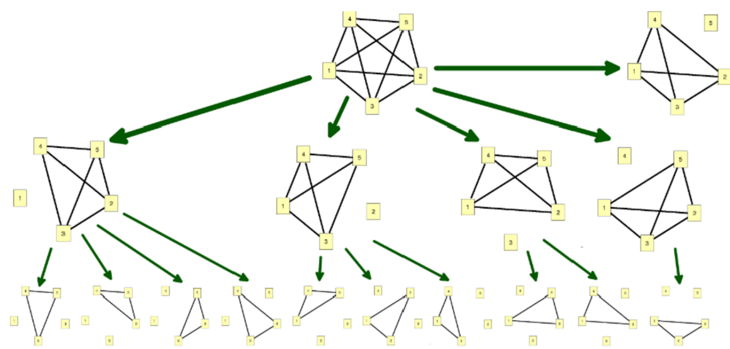


Figure 4.2: Substructure lattice.

2-4 initializes covered substructure for each previous query as empty.

5-12 performs "score calculation". For each substructure, 6 estimates its space. 8-10 iterates over all "favored" previous queries and adds on marginal benefit if any. Here marginal benefit refers to the time saved by adding current substructure to the selected covered substructures.

13-23 performs "pick-and-update". 15-17 terminates selection when there is no marginal benefit at all. 19-22 updates covered substructures for previous queries as a result of current round of selection.

Implementation of functions are listed as follows. Notice that users can implement in their own ways based on their database systems.

Function benefit(q, substructure, q.coveredSubstructre) evaluates marginal benefit of substructure to query q provided that q.coveredSubstructre has been materialized. Such estimation is tricky as using materialized substructure may change original joining plan. Thus estimation on intermediate result sizes is nessasary. We used

$$estimateProcessingTime(q.coveredSubstructre \cup substructure) - estimateProcessingTime(q.covere$$

for estimation. As we think that this roughly indicates improvement of adding *substructure* in terms of reduction of hard-disk access and joining operations.

### 4.2.4 ID and Property Selection

Given a substructure picked by Structure Planner, we need to decide on which IDs and attributes to be stored. Taking all IDs and attributes into account will assure that the structure could be used in any future query which covers it but will on the other hand increase space cost. We are faced with a tradeoff between space and potential usage spectrum.

**IDs:**

- Include IDs of all nodes and edges. This would allow overlap joining of substructures but increases space cost.
- Include IDs of only outer nodes. This saves space cost but does not allow overlap joining of substructures.

Our suggestion is to consider the expansion effect of inner nodes' IDs towards space cost. In our implementation, inner nodes' IDs are kept. However if adding inner nodes' IDs overwhelmingly increase resulting table length, then we may choose to ignore inner nodes' IDs as the overhead on space cost is too much.

**Attributes:**

- Include all attributes.
- Include only attributes that appears in previous workloads.

Our suggestion is to consider the portion of attributes that appeared over all attributes in the data schema. For instance, in our experiment only a small portion of attributes were aggregated, therefore it is more of a waste of space to keep all attributes in the data schema.

## 4.3 Query Processing

Problem:

Given materialization of cuboids C and substructures S how to process future queries F as fast as possible using C and S?

Generally speaking, aggreation on cuboid is faster than joining substructures. When future query q arrives, we first consult materializion of cuboids. If any cuboids get ”hit” by q, then we select the cuboid of minimum space to aggregate the result of q. If no cuboid gets ”hit” by q, then 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 future query
**Output:** r: result of q

```
minspace := ∞;
mincuboid := NULL ;
foreach cuboid ∈ C do
    if cuboid.structure = q.structure and q.dimension ⊆ cuboid.dimension then
        if cuboid.space¡minspace then
            minspace := cuboid.space ;
            mincuboid := cuboid ;
        end
    end
    if mincuboid ≠ NULL then
        r := aggregate(mincuboid, q);
    else
        r:=Decompose_Join(q);
    end
    end
```

4-9 looks up materialized cuboids and find if there is any "hit". If there are multiple "hits" use the cuboid with the smallest scanning cost.

### 4.3.1 Substructure Selection

Given a future query q and materialized substructures S, which substructures shall we select to compose q? Obviously selected substructure s must hold s.structure $\subseteq q.structure$.

However, candidate substructures in S may overlap:

For instance suppose

q.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

Then we may have at least three ways of substructure selection.

(1) and (2)

(3) and (4)

(1), (4) and (5)

We propose a greedy algorithm for substructure selection. The idea is to always pick up next substructure with highest heuristic score. Example heuristics are edges of substructure, Score when selected by Structure-Planner, tuples in the table etc.

**Algorithm 7:** SelectSubstrucre

**System:** S: a collection of materialized substructures
heuristic: heuristic for ordering 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

uncoveredStruc := q.structure ;
uncoveredProp:= q.properties ;
$coveredStruc := \emptyset$ ;
$V := \emptyset$;
**foreach** $s \in S$ *ordered by heuristic* **do**
    **if** $s \subseteq$ *uncoveredStruc and* $s \not\subseteq$ *coveredStruc* **then**
        $V := V \cup \{s\}$;
        $coverdStruc := coveredStruc \cup s.structure$ ;
        uncoveredStruc := uncoveredStruc - s.structure ;
        uncoveredProp := uncoveredProp -s.properties;
    **end**
**end**

5 iterates substructures by user-defined heuristics.

6 assures that a candidate substructure that is totally covered by selected substructures will be dequalified as major effect of the candidate is already occupied.

### 4.3.2 Query Decomposition

Given future query q, we use SelectSubstrucre to select materializaion V. For the q's remaining structure and properties that V does not cover, we have to retrieve them from database server. Finally we join and aggregate all materials together to produce results.

**Decompose_Join**

**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 | $\Sigma \leftarrow \Sigma \cup \{materialize(s)\}$;
7 **end**
8 r := join($\Sigma$)

1 initializes $\Sigma$, which stores all materials that are needed.

2 selects substructures using SelectSubstructure algorithm. *uncoveredStruc*, *uncoveredProp* are structures and properties not covered. They need to be retrieved from database servers.

4, splits uncoveredStruc and uncoveredProp into connected components. We will retrieve each connected component from database server. Nocice that uncoveredStruc may not be exactly one connected component.

8 joins and aggregates all materials together to produce results.

Function split(uncoveredStruc, uncoveredProp) is implemented by classic connected components decomposation algorithms.

Function join(Σ) can be implemented with different table-joining strategies. In our implementation we keep joining two tables which have common colmn(s) and with minimum sum of table sizes. That is, we always select small tables to join.

$Decompose\_Join_{informative}$

Alternatively, we may adopt the idea of Semi-Join. We first join V. The process of joining has a "filtering" effect. When we retrieve uncovered components from database server, we inform database server with the screened out IDs information. We name this approach "informative materialization". "Informative materialization" may accerlate retrieval from backend databases in two aspects:

First, since screened out IDs are provided, database backend only need to search within screened out IDs. This will reduce database processing time.

Second, size of retrieval results can be deducted. Thus time of transporting results will be reduced.

**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

$\Sigma \leftarrow \emptyset$;
$V, uncoveredStruc, uncoveredProp \leftarrow SelectSubstrucre(q)$;
V:=join(V);
$\Sigma \leftarrow \Sigma \cup V$;
Splits:=split(uncoveredStruc, uncoveredProp);
**foreach** *s: Splits* **do**
  $\Sigma \leftarrow \Sigma \cup \{materialize_{informative}(s, V)\}$;
**end**
r := join(Σ)

Decompose_Join perform joining after everything is ready. Unlike Decompose_Join, we first join V in 3 before retrieval from databases(7). Note that substructures in V may reside in multiple connected components. Thus join(V) may result to multiple Intermediate tables.

In 7, $materialize_{informative}$ fetches results from databases by passing candidate ID information "screened-out" from 3. For instance, Neo4j supports such operations of passing a list of IDs as arguments.

---

$Decompose\_Join_{decisive}$

However "Informative materialization" creates an overhead of transport of screened out IDs. We propose a decisive way to evaluate the trade-off between overhead and benefits of "Informative materialization" and choose between "Informative materialization" and "normal materializaiton".

**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);
4 $\Sigma \leftarrow \Sigma \cup V$;
5 Splits:=split(uncoveredStruc, uncoveredProp);
6 **foreach** *s: Splits* **do**
7     **if** *decide_informative(s, V)* **then**
8         $\Sigma \leftarrow \Sigma \cup \{materialize_{informative}(s, V)\}$;
9     **else**
10         $\Sigma \leftarrow \Sigma \cup \{materialize(s)\}$;
11     **end**
12 **end**
13 r := join($\Sigma$)

In 7, Function decide_informative(s,V) determines between $materialize_{informative}$ or not. In our implementation we caculates ratio of estimated reducted result size by $materialize_{informative}$, divided by size of input overhead. We make decision by comparing the ratio with a threshold.

# 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???

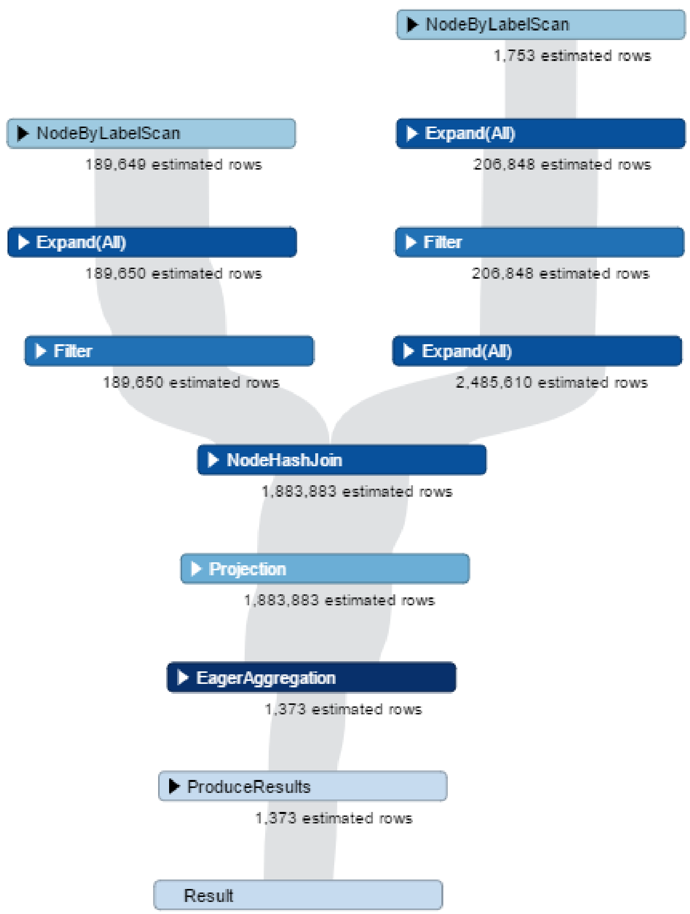
- Multi-Thread implementation

  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.

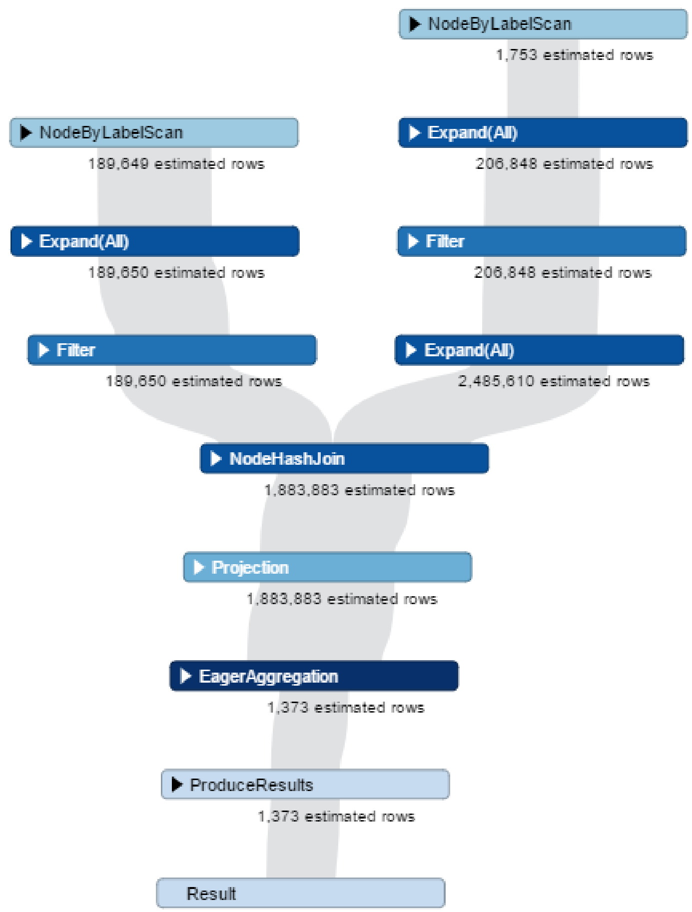
## 6.2 Reflection on Neo4j

### 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$(\mathrm{att}_1, att_2 att_n) = Productof(|att_i|) * (shrinkingfactor)^(n-1)$

# 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
```