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, like social networks, financial transaction networks and ect. On-Line Analytical Processing(OLAP) provides an important tool for data analyses by allowing users to perform data aggregation through different combinations of dimentions. For instance, over a Q&A forum dataset, in order to study if there is a correlation between age and post quality,one may ask what is the average user's age group by post score. In the field of music industry, we may process a query like what is total sales of records with respect to music company and year to study market activities.

State-of-art graph databases like neo4j do not have efficient support for OLAP aggregation queries. Neo4j processes each OLAP query in two steps. First expands nodes and edges to the query structure, and then perform aggregation. Even if a query is repeatedly executed for multiple times, in each round Neo4j still processes the query from scratch, without caching any structure-wise "knowledge" from previous workload. When it comes to large property graphs, current graph databases' efficiency is far from satisfaction. It is unacceptable for users to wait for hours for a single query to return.

We implement a prototype system upon Neo4j that greatly improves efficiency of OLAP over large property graphs. The idea is to smartly materialize some views computed based on previous workload. Such materialization can be used to accelerate future query processing.

We implemented our system on top of Neo4j and compared our system with orginal Neo4j system. Based on an empirical study over real-world property graph dataset, result shows that, with an acceptable cost of memory or disk usage, our solution achieves 10-30x improvement in time efficiency for OLAP queries.

Acknowledgements

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

Dedication

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

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Property Graph Model 1

1.2 Graph OLAP 4

1.3 Challenges on Graph OLAP 4

1.4 Proposed Solution and Contributions 6

2 Background and Related Work **8**

2.1 OLAP over Property Graph Model 8

2.2 Graph Databases and Neo4j 14

2.3 Related Work 16

2.4 Graph Cube 17

3 Problem Definition **19**

3.1 Terminologies 19

3.1.1 Query Notations 19

3.1.2 Cuboid vs Substructures 20

3.2 Problem Definition 21

4 Solution **22**

4.1 Solution Framework . . . 22

4.2 Partial Materialization . . . 23

4.2.1 Greedy Selection Framework . . . 25

4.2.2 Cuboid Planner . . . 27

4.2.3 Structure Planner . . . 32

4.2.4 ID and Property Selection . . . 34

4.3 Query Processing . . . 34

4.3.1 Substructure Selection . . . 36

4.3.2 Query Decomposition . . . 38

5 Experiments **42**

5.1 Experiment Setup . . . 42

5.1.1 Datasets . . . 42

5.1.2 Query Workloads . . . 42

5.1.3 System Setting . . . 45

5.1.4 Neo4j Configuration . . . 45

5.2 Aspects of Interest . . . 45

5.3 Efficiency Test . . . 46

5.3.1 Neo4j BaseLine . . . 46

5.3.2 My System . . . 46

5.3.3 Frequency Threshold . . . 46

5.3.4 Memory Limit . . . 46

5.3.5 Selection Algorithms . . . 46

5.3.6 View Selection . . . 46

5.3.7 Decompose_Join . . . 46

5.4 Discussion . . . 46

6 Conclusion **47**

6.1 Future Work 47

6.2 Reflection on Neo4j 48

6.2.1 Aggregation Size Estimation 48

References **50**

APPENDICES **51**

A PDF Plots From Matlab **52**

A.1 Using the GUI 52

A.2 From the Command Line 52

List of Tables

2.1 Comparison 17

List of Figures

1.1 A simple property graph modeling "users post posts"(data graph). 2
1.2 Meta graph containing User, Post and Tag. 3
1.3 A snapshot of data graph containing User, Post and Tag. 3
1.4 Execution plans of Query #3 for first time and fifth time. 5

2.1 *Structure* of Query #1 12
2.2 *Structure* of Query #2 12
2.3 Cube of properties {A,B,C}. 13
2.4 *Structure* of Query #3 13
2.5 A simple property graph. 14

4.1 Solution framework. 22
4.2 Substructure lattice. 33

Chapter 1

Introduction

OLAP (On-Line Analytic Processing) is the fundamental to many decision-making applications, like smart business, advertising, and risk management. 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 [**?**]. However, existing graph database systems do not support efficient OLAP queries. Surprisingly, they 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 created every second. As a result, massive amount of data of connected systems is generated at the same time. When a user creates a new post not only a post is created, a "creates" connection between the user and the post is established as well. When the user tags the post with a tag, a "has tag" connection is built between the tag and the post. When a bank transfer happens, a "transfers" connection between two accounts is created.

Property graph is a widely used model for such systems of various connections. A property graph is made of nodes, edges, and properties. Like general graph data models, nodes represent entities and edges represent relationships. For instance, Figure 1.1 is a

simple property graph of from online Q&A forum named www.StackExchange.com. It shows the connections of users (represented by red nodes) and posts(represented by blue nodes). Each arrow pointing from a user node to a post node represents a "User_onws_Post" connection. From the graph, we can clearly know that one user has created 1 post and the other user has created 2 posts.

We will use a property graph of www.StackExchange.com throughout this thesis. We will call this graph "StackExchange graph".

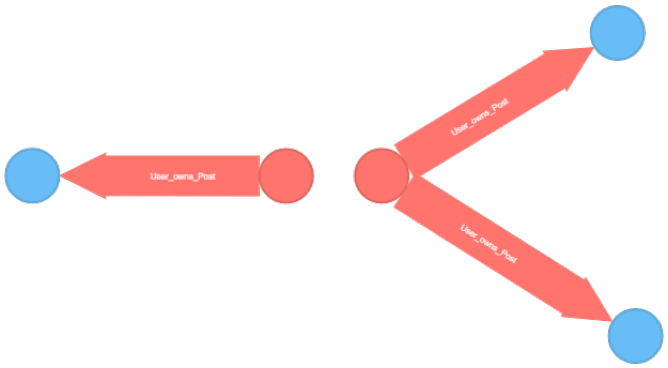


Figure 1.1: A simple property graph modeling "users post posts"(data graph).

Besides nodes and edges, in property graph nodes and edges can have any number and type of properties.

For instance, in the exampling property graph, a User node may have properties like the users Age, Views, UpVotes etc(listed at the end of the picture). Notice that there is no restriction on what properties a User node can have. That is, any node or edge could be freely attributed with any type of property. This makes a property graph very flexible in terms of property attribution.

Property graph is an informative model as it contains not only nodes and edges, but properties of each individual node and edge as well.

Meta graph demonstrates graph information on a schema level while data graph refers to the actual graph specific to node and edge level. Figure 1.2 and Figure 1.3 are meta graph and a snapshot of data graph of a property graph on www.stackexchange.com. It contains:

Nodes: User(in red). Post(in blue). Tag(in green).

Edges: User_owns_Post. Post_hastag_Tag.

Properties: Users View, Posts Score, Tags Tagname etc.

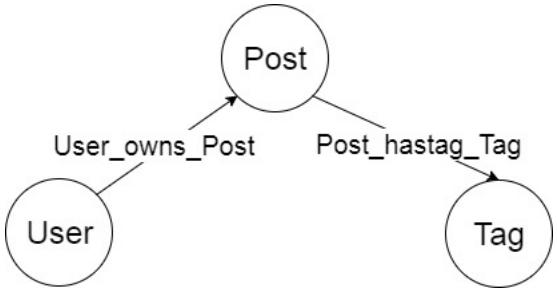


Figure 1.2: Meta graph containing User, Post and Tag.

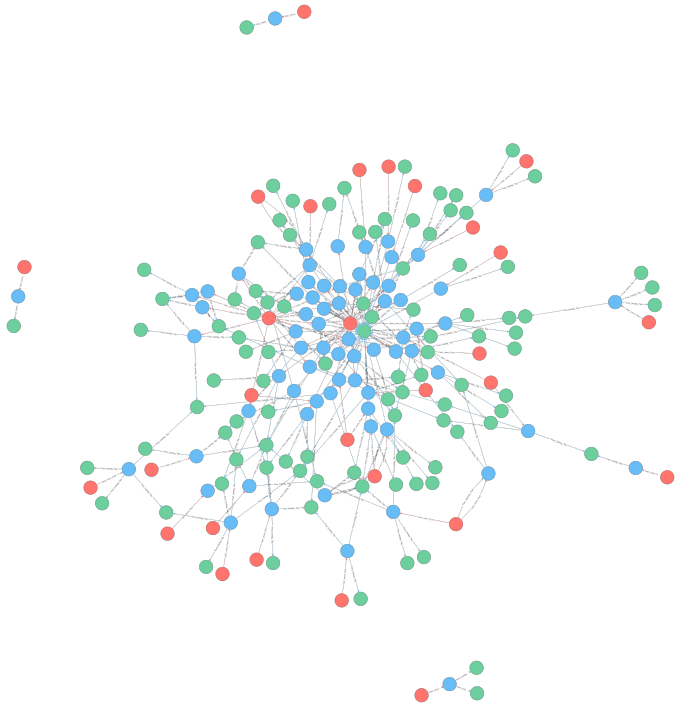


Figure 1.3: A snapshot of data graph containing User, Post and Tag.

1.2 Graph OLAP

From a property graph we can ask many interesting questions and queries. Among various kinds of queries, OLAP(Online analytical processing) queries play an important role in data analysis.

For instance on "StackExchange graph" we may perform OLAP

Query: Get average post score grouped by users upvotes.

to see if high upvotes of a user indicate a high-quality post. If the result is shows a significant correlation, we may use the authors upvotes as a factor to estimate the quality of his or her post when a post is freshly posted and score of the post has not been well voted been yet.

With a property graph dataset on music industry we may ask

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

to evaluate a company's previous strategy in order to increase share of young people's market.

We call such OLAP over graphs "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 an interesting research topic in database area.

1.3 Challenges on Graph OLAP

We know that Graph OLAP is important. However there are many challenges on this topic. One of the most challenging part is efficiency issue.

From an academic point of view, most of OLAP studies reside in traditional relational data models, whereas studies on efficient graph OLAP are not enough. What's worse, current graph OLAP researches either focus on accelerating graph OLAP over a special subset of property graphs, or focus on general highlevel topics on graph OLAP other than the important issue of query efficiency improvement.

As a result, current databases do not provide efficent support for graph OLAP, especially when it comes to large datasets. Graph databases are databases that specialize in storage and processing of property graphs. However current graph databases are not

satisfactory in terms of OLAP processing efficiency over large property graphs(often with more than millions of nodes and edges).

We found that current graph databases like Neo4j process each OLAP query in a naive manner: without using any information of previous queries. In an extreme case, even if we executed a query again and again, execution plan for the query always stays the same and thus execution time does not improve.

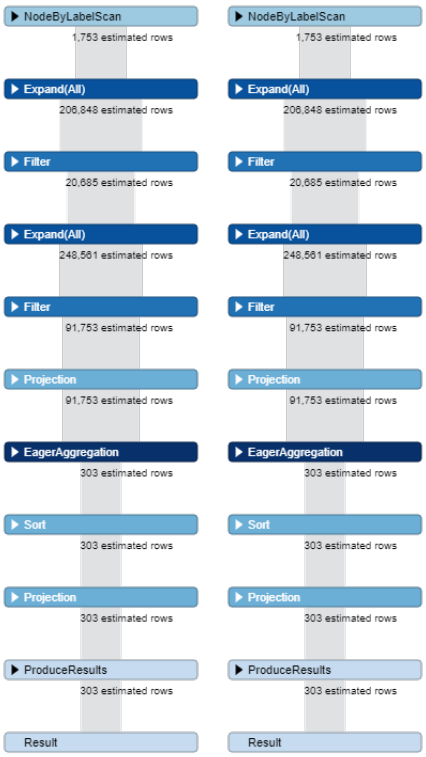


Figure 1.4: Execution plans of Query #3 for first time and fifth time.

This naive feature of always executing queries regardless of previous queries misses valuable information contained in previous workloads.

For instance, 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 for 2 hours or even more than a day for the result of one OLAP query as this

would undermine interactivity which is one of the most best features of OLAP. Therefore we want to accelerate OLAP processing over large property graphs.

1.4 Proposed Solution and Contributions

We want to propose a system that supports efficient OLAP over large property graphs.

In reality, most OLAP queries are conducted in a real-time manner. As a result, future queries are unknown before they arrive. However in reality a clients attention tends to reside in several particular structures and properties (closely related with the topics that the client is interested in). Within a specific time range, it is these "hot" structures that the client tends to repeatedly view in different dimensions. Therefore previous queries can be used as a good reference for understanding which structures and properties the clients are currently interested in.

For instance, suppose a client just executed the exampling four queries, here is what we can learn from these four previous workload:

Structure-wise: (User)-[creates]->(Post) is frequently queried. We can tell that client is interested in how users create posts. Thus it is reasonable to guess that the user is likely to issue OLAP queries involving (User)-[creates]->(Post) in following queries.

Property-wise: User.UpVotes, Post.Score, User.Age, Tag.TagName etc. More specifically, {User.UpVotes, Post.Score} and {User.Age, Tag.TagName} are frequent combinations. Thus it makes sense to guess that these property combinations are likely to appear together in future queries.

Intuitively, if we smartly select frequently queried structures and properties based on previous workload and materialize them, they might be covered in future queries and thus be used to accelerate processing.

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.

Therefore we propose a system that realizes automatic and smart pre computation and materialization based on finished workload, and utilization of materialized result to facilitate faster future query processing.

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.

We summurize major contributions of our work is 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 in this way: In part 2 we will discuss about related work. We will introduce pre-military knowledge about OLAP, graph databases, Neo4j; and provide a summary of how existing work solves OLAP queries. In part 3 we will explain our solution framework and system design in details. Part 4 is on experiments. We will talk about experiment design, followed by presentation of experimental results and discussions of results. Part 5 will talk about opening questions and future work.

Chapter 2

Background and Related Work

2.1 OLAP over Property Graph Model

We have introduced about property graph model in Introduction part. We know that in property graph model, each node and edge could have arbitary number and type of properties. A type of property is represented by

NodeType.PropertyType

For instance User.Age represents "Age" property on "User" node.

In order to identify a node or edge, a unique ID is assigned to each node and edge. In this thesis we represent a type of property by

ID(node) or ID(edge)

to represent unique ID of a node or edge.

OLAP (On-Line Analytical Processing) [**?**] [**?**] [**?**] is an important notion in data analysis. Given the underlying data, a cube can be constructed to provide a multi-dimensional and multi-level view, which allows for effective analysis of the data 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.

Graph OLAP is first proposed by Graph Cube [**?**]. It refers to OLAP over graphs. There is no formal defination of the notion "Graph OLAP". Graph Cube [**?**] views the

outcome of Graph OLAP as aggregated graphs(aggregation of data graph). In our work we view the outcome of Graph OLAP as result tables of OLAP queries.

Graph Cube [**?**] addresses and defines two most important notions in graph OLAP senerio as *dimension* and *measure*. In our work emphasize *structure* (in meta graph) as a third important notion. Graph Cube [**?**] focuses more on OLAP senerios over a fixed *structure*, with varying *dimension* and *measure*. Therefore there is no need to emphasize and discover the notion of *structure*. In our work,we want to deal with OLAP senerios over various *structures*.

The "graph" in our work refers to property graphs. As introduced in Section 1.1, property graph is more flexible and powerful than classic attribute graphs.

In order to better illustrate how "Graph OLAP" is interpreted in our thesis, let's look at the following four example senerios where we perform OLAP over "StackExchange graph".

Query #1

Question: Does high upvotes of a user indicate a high-quality post?

Query: Get average post score grouped by users upvotes.

Result:

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

What we learned: 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.

Query #2:

Question: Following 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: Get average post score grouped by users upvotes and posts post types.

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

What we learned: From the query result we are suggested that high-quality users not only provide good questions but ask valuable questions as well. However, there is a subtle difference on how upvotes correlate with questions and answers. For instance, 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 Tag.

Query #3:

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

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

Selected Result:

TagName	AVG(Age)
router	19.6
python	24.1
internet	26.8
c	30.2
programmer	31.4
software	29.8

What we learned: We can see the average user age of each tag clearly and easily compare them. For instance, python is generally more popular among younger users. "Router" is a relatively "younger" topic than "internet".

Query #4:

Question: Follwing Query #4, lets look at tendency of topics "average popular user age" by years. Is there a tendency of younger age?

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

Selected 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

What we learned: Tendency of younger age on IT topics is seen. Python is getting faster embraced by younger people compared with C. Similarly we can compare two commercial products customer targeting strategy, advitising 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.

We address three key elements of a graph OLAP as *structure*, *dimension*, and *measure*. Taking Query #1 as an example:

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

This query is over the following *structure* in blue on the meta graph:

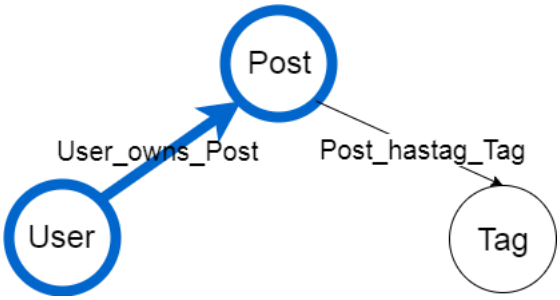


Figure 2.1: *Structure* of Query #1

We say that (User)-[User_owns_post]->(Post) is the structure of Query #1. The query is aggregated by (grouped by) users upvotes. We say that User.Upvotes is the dimension of Query #1. The query is aggregated on avarage of posts score. We say that AVG(Post.Score) is the meansure of Query #1.

Similarly, for Query #2:

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

Structure: (User)-[User_ownspost]-(Post)

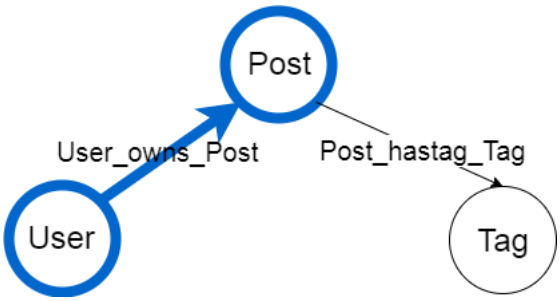


Figure 2.2: *Structure* of Query #2

Dimensions: User.Upvotes, Post.PostTypeId

Measures: AVG(Post.Score)

Notice that Query #2 adds Post.PostTypeId to Query #1s dimensions. That is to say, Query #2 asks for a more detailed partitions over dimentions. We call Query #2 a drill-down from Query #1, and Query #1 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.

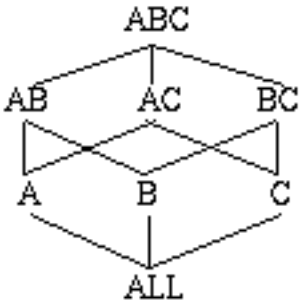


Figure 2.3: Cube of properties {A,B,C}.

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

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

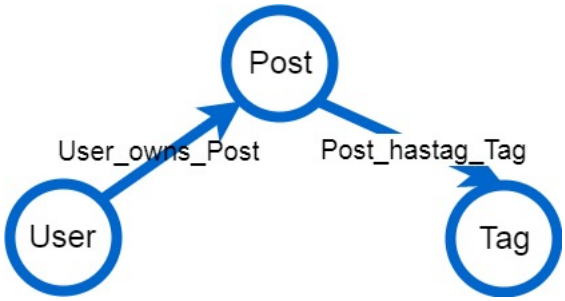


Figure 2.4: *Structure* of Query #3

Dimensions: Tag.Tagname

Measures: AVG(User.Age)

Note that Query #3 has different *strucutre* than Query #1 and Query #2. In Query #3, a requirement that post must be created in year 2017 picks out a particular subset of all. 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 Post.Year=2017 a *"slicing condition"* of Query #3.

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

There are two major types of databases that store and process graph data: traditional relational databases and graph databases.

Relational databases adopt traditional ways of modeling data in form of entity and relationship tables. For instance, for the following property graph, which consists 1 user and his or her 3 posts. A relational database stores the property graph as 3 tables:

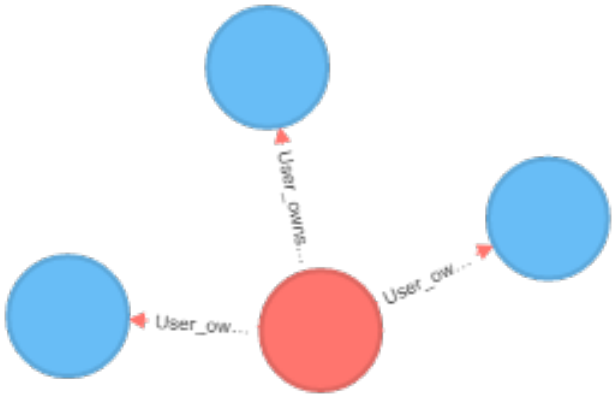


Figure 2.5: A simple property graph.

Table-User

Uid	All user properties
1	Property values

Table-Post

Pid	All post properties
1	Property values
2	Property values
3	Property values

Table-Owns

Uid	Pid
1	1
1	2
1	3

There are two drawbacks of storing property graphs in relational databases. First, each node or edge of in property graph could have arbitrary types of properties. However, schemas of relational tables restrict nodes or edges of a same type to have a uniform set of properties (attributes). Second and more importantly, edges are not stored as a separate table. For instance, 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 independant 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.

Relational databases and graph databases both have their own strengths in term of query processing. However it is generally accepted that graph databases perform better at property graph data processing as it conforms more with the actual graph structure.

Many graph database applications have been implemented and commercialized. One of the popular ones is Neo4j, which holds atomicity, consistency, isolation, durability (ACID).

Instances are modeled and stored as property graphs in Neo4j. Like other graph databases, edges are not stored as physically separate tables, but directly attached to their nodes. One special thing 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 instance 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. Here are some examples of Cypher queries:

Aggregation query: For answers(Post.PostTypeId=2), what is the average score group by different user upvotes?

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

Here User and Post are labels, PostTypeId and Score are properties of Post node, Upvotes is property of User node.

Neo4j is written in Java. While applications in various languages could interact the server by issuing queries and transport using BOLT protocol. BOLT protocol is a binary protocol implemented by Neo4j team. It is more efficient than HTTP protocol.

2.3 Related Work

Numerous papers have been published on graph aggregation.

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 summa- rized 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 Cube [**?**] introduces Graph Cube, a new data warehousing model that supports OLAP queries effectively on large multidimensional networks. It takes account of both attribute aggregation and structure summarization of the networks. In order to deal with curse of dimensions, a greedy algorithm framework is introduced for partial materialization of cuboids.

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

	Input graph	Query pattern	Layer structure	Highlight
Cube-based [**?**]	property graph	simple relation	yes	cubes on edges and no
Gagg [**?**]	property graph	all exact-match patterns	no	structural patterns
Pagrol [**?**]	property graph	edge and node attributes	yes	Map-Reduce distributed co
Graph Cube [**?**]	homogenous	node attributes	yes	partial materialization alg
Graph OLAP [**?**]	property graph	edge and node attributes	yes	distributive and holistic m

Table 2.1: Comparison

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.

We summarize some of the most related ones as follows:

From the summary we can categorize the papers into two kinds:

First, like Graph Cube [**?**], focuses on a simple subset of property graphs(e.g. graphs with only homogenous nodes and edges) and proposes optimizations in order to accelerate OLAP query processing. 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 [**?**], focus on an abstract high-level framework that process generic queries over generic property graphs. However, query processing efficency is not studied.

To conclude, we can see a lack of study on structure-aware optimizations for efficent graph OLAP. As mentioned in Section 1.3, efficency 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.

2.4 Graph Cube

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 C1 and C2 from the same graph cube, as long as dimension(C2) is a subset of dimension(C1), result of C1 can be used to generate result of C2. 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.

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 Query Notations

In this thesis, we use the following way to represent an OLAP query.

Structure : Dimension, Measure, Slicing Condition

For instance Query #3

Get average user age grouped by users 2017 posts tags.

is written as

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

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-Post, Post-Tag.

3.1.2 Cuboid vs Substructures

Suppose we have the following 7 queries in previous workload:

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

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

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

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

User-Post, Post-Tag: User.UpVotes, (AVG)Post.Score,Tag.TagName=algorithm

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

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

We can tell that the user is interested in User-Post, Post-Tag structure. Suppose we build a cube as in Graph Cube [4] on User-Post, Post-Tag over all queried attributes and precompute some most beneficial cuboids.

For instance we precompute cuboids like

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

by simply joining cuiboid

User-Post, Post-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.

We define our problem as

Given previous queires P, space limit σ, 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 σ, select C and S wisely.

Given F, C and S, process F fast.

Chapter 4

Solution

4.1 Solution Framework

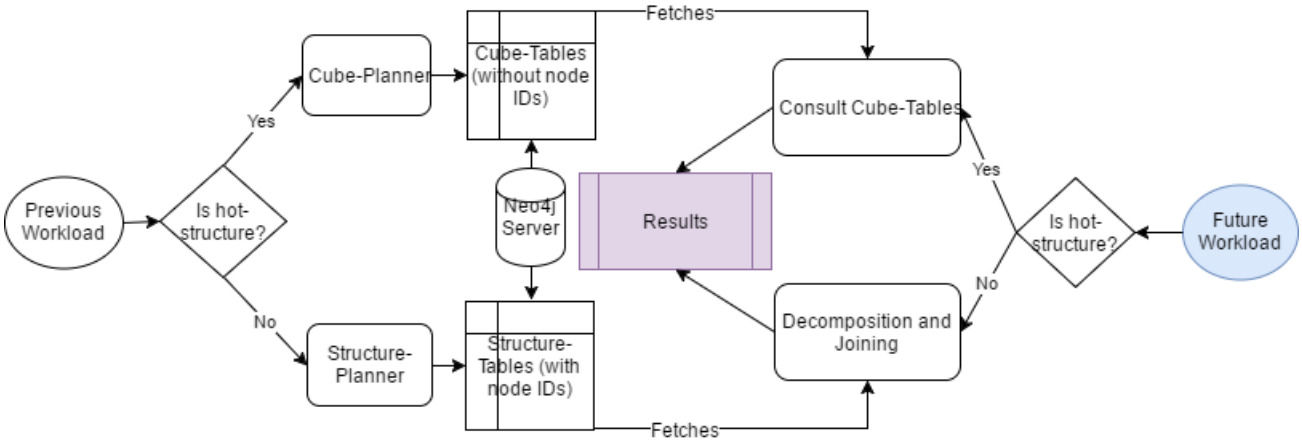


Figure 4.1: Solution framework.

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

```
1  CInput ← ∅ ;
2  SInput ← ∅ ;
3  foreach q ∈ Q do
4  |   if structureFreq(Q, q) > threshold then
5  |   |   CInput ← CInput ∪ {q};
6  |   else
7  |   |   SInput ← SInput ∪ {q};
8  |   end
9  end
10 C := materialize(CubePlanner(CInput)) ;
11 S := materialize(StructurePlanner(SInput));
12
```

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$_Year, Tag - TagName$

User-Comment, Comment-Post: User-ActiveMonth, Post-CreationDate$_Year = 2016$

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$_Year = 2016$

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 σ, 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: σ: 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.

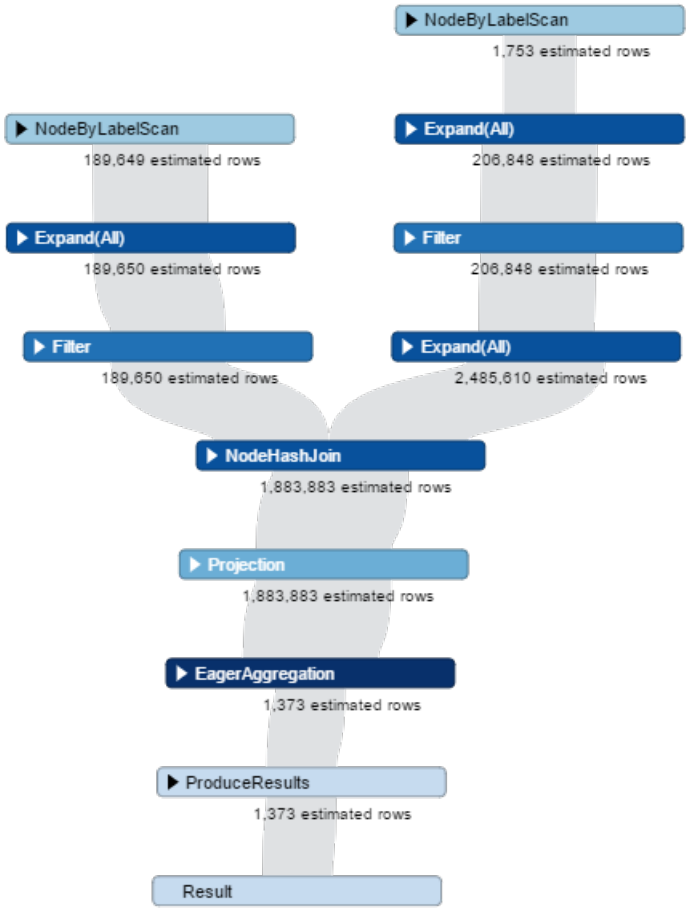
13-23 performs "pick-and-update". 15-17 terminates selection when there is no marginal benefit at all. 19-22 updates best-so-far processing time for previous queries as a result of current round of selection.

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

```
1 Group:= GroupByStructure(Q) ;
2 foreach group ∈ Group do
3 |   group.candidates := SingleCubePlanner(group);
4 end
5 for i=1 to n do
6 |   selectedGroup := group in Group with highest group.candidates.top().score ;
7 |   C.offer(selectedGroup.poll());
8 end
9
```

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.

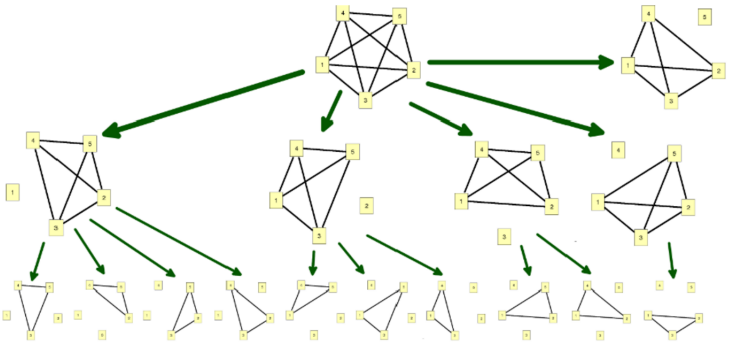


Figure 4.2: Substructure lattice.

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

```
1  minspace := ∞;
2  mincuboid := NULL ;
3  foreach cuboid ∈ C do
4      if cuboid.structure = q.structure and q.dimension ⊆ cuboid.dimension then
5          if cuboid.space¡minspace then
6              minspace := cuboid.space ;
7              mincuboid := cuboid ;
8          end
9      end
10     if mincuboid ≠ NULL then
11         r := aggregate(mincuboid, q);
12     else
13         r:=Decompose_Join(q);
14     end
15     end
16
```

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 sub-

structure, 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
 1 uncoveredStruc := q.structure ;
 2 uncoveredProp:= q.properties ;
 3 coveredStruc := ∅ ;
 4 V := ∅;
 5 foreach s ∈ S ordered by heuristic do
 6 |   if s ⊆ uncoveredStruc and s ⊈ coveredStruc then
 7 |   |   V := V ∪ {s};
 8 |   |   coverdStruc := coveredStruc ∪ s.structure ;
 9 |   |   uncoveredStruc := uncoveredStruc - s.structure ;
10 |   |   uncoveredProp := uncoveredProp -s.properties;
11 |   end
12 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 Σ ← ∅;
2 V, uncoveredStruc, uncoveredProp ← SelectSubstrucre(q);
3 Σ ← Σ ∪ V;
4 Splits:=split(uncoveredStruc, uncoveredProp);
5 foreach s: Splits do
6 |   Σ ← Σ ∪ {materialize(s)};
7 end
8 r := join(Σ)
```

1 initializes Σ, 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

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

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 Materializion, 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. ← 5

- Memory limit. ←20GB

- Selection algorithm. ← My algorithm

Decomposition:

- Different heuristics when ranking sub-structures during decomposition. ← edges of sub-structure

- Different ways of Decomposation_Join ← 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???

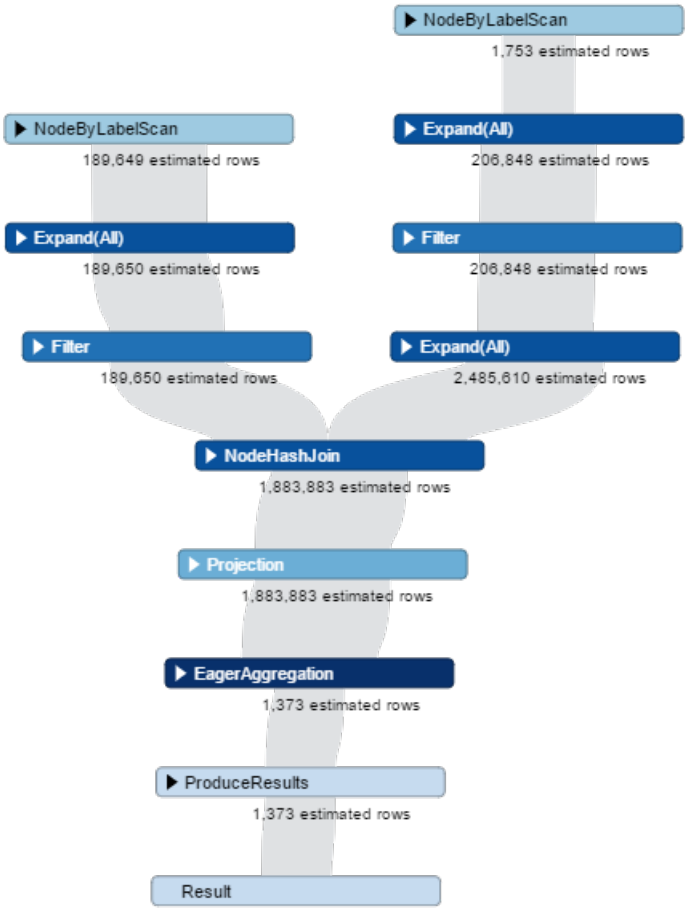
- Multi-Thread implementation

  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.

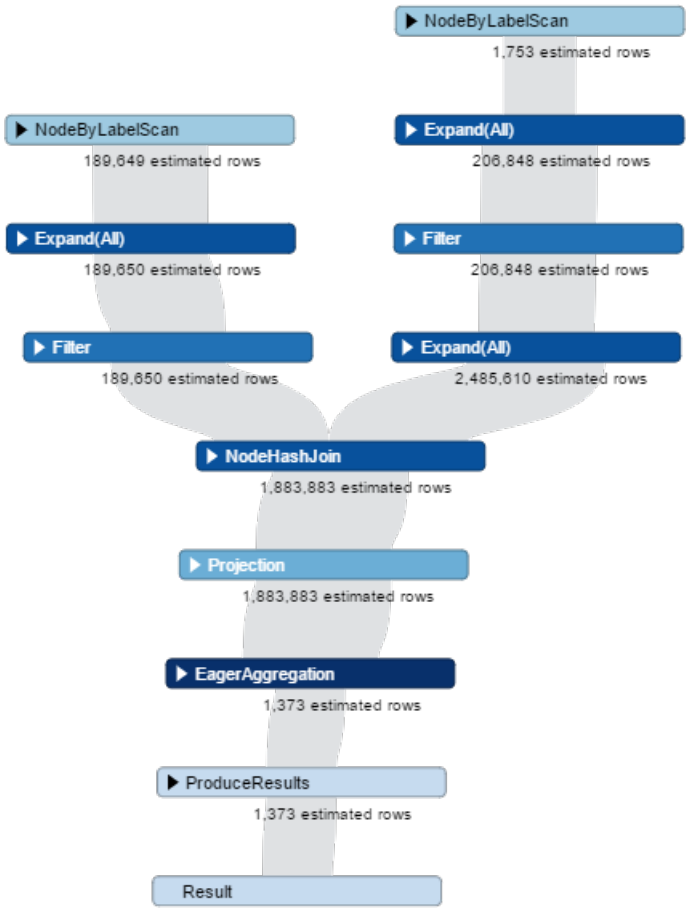
6.2 Reflection on Neo4j

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

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
```