Optimizing Graph Query Performance by Indexing and Caching

Jing Wang

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

June 2017

Abstract

Subgraph/supergraph queries, though central to graph analytics, are costly as they entail the NP-Complete problem of subgraph isomorphism. To expedite graph query processing, the community has contributed a wealth of approaches that gradually form two categories, i.e., heuristic subgraph isomorphism (SI) methods and algorithms following "filter-then-verify" paradigm (FTV). However, they both bear performance limitations. And a significant drawback of current studies lies in that they throw away the results obtained when executing previous graph queries.

To this end, the current work shall present a fresh solution named iGQ, principle of which is to acquire and utilize knowledge from the results of previously executed queries. iGQ encompasses two component subindexes to identify if a new query is a subgraph or supergraph of previously executed queries, such that the stored knowledge will be turned on to accelerate the execution of the new query graph through reducing the subgraph isomorphism tests to be performed. The correctness of iGQ is assured by formal proof. Moreover, iGQ affords the elegance of double use for subgraph and supergraph query processing, bridging the two separate research threads in the community.

On the other hand, using cache to accelerate query processing has been prevalent in data management systems. In the realm of graph structured queries, however, little work has been done. Meanwhile, modern big data applications are emerging and demanding the high performance of graph query processing. Therefore, this thesis shall put forth a full-fledged graph caching system coined GraphCache for graph queries. From the ground up, GraphCache is designed as a semantic graph cache that could harness both subgraph and supergraph cache hits, expanding the traditional hits confined by exact match. GraphCache is featured by well-defined subsystems and interfaces, allowing for the flexibility of plugging in any general subgraph/supergraph query solution, be it an FTV algorithm or SI method.

Furthermore, GraphCache incorporates the iGQ as the engine of query processing, where previously issued queries are leveraged to expedite graph query processing. With the continuous arrival of queries and the finite memory space, GraphCache requires mechanisms to

effectively manage the space, which in turn emerges the problem of cache replacement. But none of the existing replacement policies are developed specifically for graph cache. This work hence proposes a number of graph query aware strategies with different trade-offs and emphasizes a novel hybrid replacement policy with competitive performance.

Following the established research in literature, GraphCache handles graph queries against a static dataset, i.e., all graphs in the underlying dataset keep untouched during the continual arrival and execution of queries. However, in real-world applications, the graph dataset naturally evolves/changes over time. This poses a significant challenge for the current graph caching technique and hence gives rise to the requirement of advanced systems that are capable of accelerating subgraph/supergraph queries against dynamic datasets. To address the problem, this work shall contribute an upgraded graph caching system, namely GraphCache+, stressing the newly plugged in subsystems and components of dealing with the consistency of graph cache. GraphCache+ is characterized by its two cache models that represent different designs of ensuring graph cache consistency, as well as the novel logics of alleviating subgraph and supergraph query processing with formal proof of correctness.

Additionally, this work is bundled with comprehensive performance evaluations of GraphCache/GraphCache+ with over 6 million queries against both real-world and synthetic datasets with different characteristics, revealing a number of non-trivial lessons.

In overall, this work contributes to the community from three perspectives: it provides a fresh idea to expedite graph query processing, applicable for both SI methods and FTV algorithms; it presents GraphCache, to the best of our knowledge the first full-fledged graph caching system for general subgraph/supergraph queries; it explores the topic of graph cache consistency, putting forth a systematic solution GraphCache+.

Acknowledgements

It is a long long journey to go through the tunnel. Fortunately, I am not alone, as a torch of hope and courage is always there, lightening my way ahead.

First, I would like to express my great gratitude to my supervisor Prof. Peter Triantafillou. I am so lucky to be a PhD student directed by Peter – an outstanding supervisor who affords us the opportunity of meeting him every day if we need help; a dedicated scientist with immense knowledge and strict attitude to assure we are on the right track of research. Peter has spent a large amount of time and efforts in supervising my work from baby-steps. I can never thank him enough.

Moreover, I owe a big thank to my second supervisor Dr. Nikos Ntarmos. Nikos has given me far more help than an averaged second supervisor would do. I shall always appreciate the interesting discussions of addressing trivial technical issues, the diagrams covering the whole whiteboard, and the package of meaningful conversations.

Furthermore, I am grateful to my friends and colleagues for their accompanies and sincere helps during the days when I am away from home. They are Foteini Katsarou, Georgios Sfakianakis, Atoshum Cahsai, Fotis Savva, Gudrun Seebauer, Anja Schott, SvetLana Simonyan, Jieting Chen and Chloe Gray.

I dedicate this thesis to my family – the tranquil harbor of my mind and the endless source of my happiness.

To my grandfather Yujun Liu. Grandpa had taught me to bravely pursue the integrity and follow the heart when I was a kid. Time may fade a lot of things; but I will memorize Grandpa's instructions for ever.

To my parents Limu Wang and Defu Liu for their unconditional love. Dad and Mum enlighten each milestone of my life in an imperceptible manner.

To my younger brother Wenjie Wang, who is continually bringing fun.

To my husband Dr. Zichen Liu. His unwavering love supports me to proceed courageously. Nothing is better than exploring the whole world with him together.

Dedication to Grandpa, Dad, Mum and Zichen.

Author's Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Jing Wang

Abbreviations

FTV	Graph Query Processing: Filter-Then-Verify Paradigm
SI	Graph Query Processing: Subgraph Isomorphism Method
sub-iso	Subgraph Isomorphism Testing
$\mathbf{Q}_{sub}$	Subgraph Query
$\mathbf{Q}_{super}$	Supergraph Query
iGQ	Indexing Graph Queries Approach
GC	GraphCache System
GC+	GraphCache+ System
POP	Graph Cache Replacement Policy: Popularity Based Strategy
PIN	Graph Cache Replacement Policy: POP + Number of Sub-Iso Tests
PINC	Graph Cache Replacement Policy: PIN + Sub-Iso Tests Costs
HD	Graph Cache Replacement Policy: The Hybrid Dynamic Strategy
EVI	Cache Model: Evicting Graph Cache Contents
CON	Cache Model: Ensuring Graph Cache Consistency

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graph structured data are prevalent in many modern applications, ranging from chemical, bioinformatics, and other scientific datasets to social networking and social-based applications (such as recommendation systems). In biology, for example, there is a great need to model "structured interaction networks". These abound when studying species, proteins, drugs, genes, and molecular and chemical compounds, etc. In these graphs, nodes can model species, genes, etc. and edges reflect relationships between them. Molecular compounds, for instance, which consist of atoms and their bonds are naturally modeled as graphs. Ditto for social networks, where nodes refer to people and edges represent their relationships, or for recommendation engines where graph nodes model entities and attributes.

Boosted by real-world applications, a formidable challenge is to develop systems and algorithms that can store, manage, and provide analyses over large numbers of graphs. Already, there exist several very large graph datasets. For instance, the PubChem [1] chemical compound database contains more than 35 million graphs and ChEBI [2] (the Chemical Entities of Biological Interest) database contains more than half a million graphs. Further applications extend to software development and debugging [3] and for similarity searching in medical databases [4]. As a result, a very large number of graph database (DB) systems, optimized for handling graph data have emerged, such as Neo4j [5] and InfiniteGraph [6]. This is in addition to graph DBs designed by big data companies for their own purposes, such as Twitter's FlockDB [7], and Google's graph processing framework, Pregel [8] (and the list is continuously expanding, this being a very lucrative investment direction and an area promising important technological advances). Hence, the demand for high performance analytics in graph data systems has been steadily increasing.

1.1 Thesis Statement

Central to graph analytics is the ability to locate patterns in dataset graphs. Informally, given a query (pattern) graph g, the system is called to return the set of graphs in the dataset that contain g (**subgraph query** [1]) or are contained in g (**supergraph query** [9] [2]), aptly named the *answer set* of g.

The fundamental problem entailed by subgraph/supergraph query processing is subgraph isomorphism, which could have two versions. [3] The decision problem answers Y/N as to whether the query is contained in each graph in the dataset. The matching problem locates all occurrences of the query graph within a large graph (or a dataset of graphs). For both versions, the brute-force approach is to execute subgraph isomorphism (abbreviated as *sub-iso* or *SI* in the rest of this work) tests of the query against all dataset graphs. Unfortunately, these operations can be very costly and even the popular SI algorithms [10, 11, 12] are known to be computationally expensive. Moreover, SI algorithms deteriorate when the dataset is comprised of a large number of graphs, as each graph will have to be tested.

This has prompted the prevalence of the "filter-then-verify" (*FTV*) paradigm: dataset graphs are indexed so as to allow for the exclusion (filtering) of a number of those that are definitely not in the query's answer set; the remaining graphs, called the *candidate set* of g, need then to undergo testing (verification) for subgraph isomorphism. However, recently extensive evaluations of FTV methods [13, 14] show significant performance limitations, i.e., different algorithms are good for different datasets and query workloads.

All in all, the high-performance graph analytics requirement of emerging big data applications, the NP-Complete nature of subgraph isomorphism problem, and the performance limitations of current research give rise to significant questions: **Can one design and implement a suite of techniques and accompanying system that can be used to ensure high-performance graph query processing? And do so while being able to work complementing existing state of the art approaches and perform well across a variety of datasets and workloads?**

Recall the procedures of graph query processing. SI algorithms shall verify each dataset graph for subgraph isomorphism. Although FTV solutions can produce candidate sets that are much smaller than the original dataset, they still end up executing unnecessary sub-iso tests: in the simplest of cases, if the same query is submitted twice to the system, it will also be sub-iso tested twice against its candidate set.

Furthermore, a key observation is that in many real-world applications, graph queries submitted in the past share subgraph or supergraph relations with future queries. These relationships

[1] In such case, g itself is usually referred as a subgraph query; this work uses "g is a Q_{sub}" for abbreviation.
[2] In turn, g is a supergraph query, abbreviated as "g is a Q_{super}" in this thesis.
[3] In the context of this work, graph query processing is of subgraph/supergraph queries by default.

arise naturally. Queries against a biochemical dataset could range from queries for simple molecules and aminoacids, all the way to queries for proteins of multi-cell organisms. In exploratory smart-city analytics, queries referring to road networks may pertain to neighborhoods, towns, metro areas, etc. In social networking queries, exploratory queries may start off broad (e.g., all people in a geographic location) and become increasingly narrower (e.g., by homing in on specific demographics). In time-series graph analytics, queries are typically associated with time intervals, which contain (or are contained within) other intervals.

Therefore, to deal with the aforementioned questions, the basic idea of this work is **to make use of the knowledge gained from previously executed queries rather than throwing them away and expedite future graph query processing in the end**. Such approach had not been investigated by the community yet and in turn motivates the research of this thesis.

1.2 Key Questions and Contributions

Regarding the execution framework, FTV advances by including an extra filtering stage than that of standard SI algorithms, such that fewer graphs shall be verified for subgraph isomorphism and the query processing time is reduced. However, solutions in literature still bear excessive graph query time. By looking into the top performing FTV methods, this thesis first identifies the performance bottleneck of graph queries – the overwhelming verification time. SI solutions follow this conclusion naturally as their query processing involves only a verification stage (no filtering).

A significant drawback of approaches in literature lies in the fact that they all throw away executed queries without exploiting the knowledge to accelerate future query processing. On the other hand, a large number of real-world applications indicate that graph queries submitted in the past share subgraph/supergraph relationships with future queries. Such status puts forth the first key question for this work.

- **Q1:** How can the knowledge gained from processing of past subgraph/supergraph queries be harnessed to expedite future such queries?

Employing cache to accelerate queries has long been a mainstay in data management systems. However, little work had been done for graph structured queries, high performance of which is demanded by modern big data applications as never before. Hence, the second key question is delivered.

- **Q2:** How can a cache be designed for subgraph/supergraph queries so as to deal effectively and efficiently with central aspects of caching, such as admission control, graph replacement, etc.?

The current research of graph query processing is confined with a static dataset, where all graphs in the underlying dataset remain untouched during the continual arrival and execution of queries. Whereas in real-world applications, the graph dataset naturally evolves/changes over time. This poses a significant challenge for the community and thus motivates the third key question.

- **Q3:** How can consistency of a subgraph/supergraph query cache be ensured in the face of updates to the graph dataset?

Answering these questions identifies the fundamental work of this thesis, which contributes to the community a suite of techniques and systems that could optimize graph query processing across a variety of datasets, workloads and algorithm contexts in the end. The following shall give an overview of such research contributions.

C1: A Fresh Perspective of Expediting Graph Queries

This thesis proposes a novel approach of optimizing graph query processing, namely iGQ, with insights as to how the work performed by the system when executing queries can be appropriately managed to improve the performance of future queries. Unlike related works that index graphs in the dataset, iGQ rests on a query index, which consists of two components to determine the subgraph/supergraph status between new and previous queries. Indeed, all the indexing methods in literature could be utilized, since queries are graphs as well. Nevertheless, we manage to put forth a new solution for the supergraph component of iGQ, so as to avoid heavy overheads and other unnecessary sophisticated issues. Furthermore, iGQ affords a complete paradigm of accelerating graph queries, with the formally proved correctness.

C2: A Full-fledged Graph Caching System

Underpinned by the query method of iGQ, a graph caching system coined GraphCache is presented. To the best of our knowledge, GraphCache is the first full-fledged caching system for general subgraph/supergraph query processing. GraphCache is designed as a semantic graph cache from ground up. By harnessing the subgraph/supergraph cache hits, GraphCache substantially expands the traditional exact-match-only hit. Regarding the central issue of replacement in any caching system, GraphCache is bundled with a number of GC exclusive strategies that are aware of graph queries. A novel hybrid graph cache replacement policy with performance always better or on par with the best alternative is highlighted. Moreover, GraphCache is accompanied with a novel cache admission control mechanism to enhance the performance gains.

C3: Ensuring Graph Cache Consistency

Much like related works, GraphCache deals with graph query processing against a static dataset. But it is inherent that the underlying dataset graphs evolve over time. To this end, this thesis explores the topic of graph cache consistency and provides an upgraded system GraphCache+, which manages to handle graph queries against a dynamic dataset. GraphCache+ system is featured by two cache models, namely EVI and CON, reflecting different designs of ensuring the consistency of graph cache. Furthermore, GraphCache+ extends the paradigm of iGQ that is static dataset oriented, so as to accommodate the new setting in which dataset graphs keep changing during the query workload proceeding. The correctness of the new paradigm is assured by formal proofs.

C4: Double Use Characterizing The Design Space

The design of this work demonstrates the elegance of killing two birds using one stone in a number of situations. First of all, iGQ provides the complete logics for subgraph/supergraph query processing, putting two categories of queries under the roof of GraphCache(GraphCache+) and bridging the separate research threads so far in the community. Moreover, the query paradigm of GraphCache(GraphCache+) is applicable for both FTV and SI methods, complementing state of the art approaches in literature. Furthermore, in the heart of iGQ lies a query index, which is shared by two components in detecting the subgraph/supergraph status between new and previous queries.

C5: Extensive Performance Evaluations of GraphCache(GraphCache+)

Like any system, GraphCache(GraphCache+) must be evaluated by a large number of queries so as to achieve reliable results. However, the NP-Complete [15] nature of subgraph isomorphism testings could lead to queries with very long execution times. Nevertheless, this work manages to utilize over 6 millions queries to conduct the performance evaluation, from which a variety of non-trivial lessons are delivered.

C6: SI + GC $>=$ FTV

As aforementioned, GraphCache(GraphCache+) affords speedup of graph query processing by removing unnecessary sub-iso testings. Essentially, such approach is similar as that of FTV paradigm, which alleviates SI methods by filtering out some dataset graphs. Hence, it naturally follows the investigation of "SI + GC versus FTV", i.e., operating GraphCache on simple SI methods and comparing with state of the art FTV algorithms. Through extensive experiments, "SI + GC" is proved competitive, obtaining comparable or better performance

for a fraction of the space and no pre-processing cost than the counterpart. The conclusion that using GraphCache on top of SI methods could replace the best-performing FTV algorithms is noteworthy.

1.3 Thesis Structure

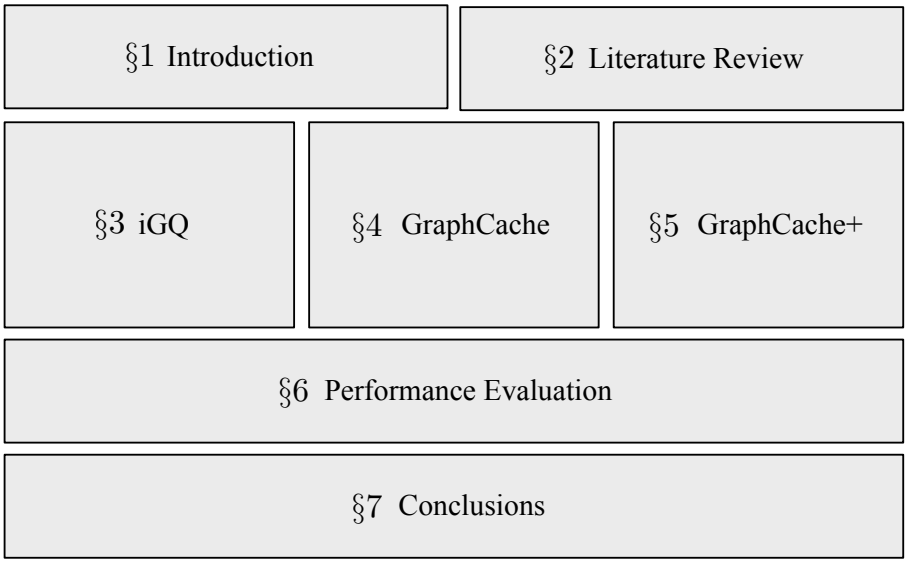


Figure 1.1: Thesis Structure: Organization of Seven Chapters

This thesis is structured into seven chapters in total, as illustrated in Figure 1.1.

- Chapter 1 presents the thesis statement, identifies fundamental questions and overviews the contributions.
- Chapter 2 reviews related work in literature, summarizing the approaches pertaining to graph query processing, subgraph isomorphism problem and caching system.
- Chapter 3 provides a fresh principle of indexing graph queries (iGQ) to speed up query processing. iGQ framework for subgraph/supergraph query processing is highlighted, each is accompanied with formally proved correctness and benefit analysis. In addition, the key algorithms and structures of iGQ are presented.
- Chapter 4 puts forth GraphCache, to the best of our knowledge the first full-fledged caching system for general subgraph/supergraph query processing. GraphCache is pictured by the system architecture that could easily plug-in all current FTV and SI methods, the materialized iGQ query processing engine, the cache admission control mechanism and a number of graph cache exclusive replacement policies.

- Chapter 5 shows GraphCache+, the first study in literature to explore the topic of graph cache consistency, affording a systematic solution to deal with graph queries against dynamic underlying datasets.
- Chapter 6 performs extensive evaluations (with millions of queries) using the well-established FTV and SI methods, on real-world and synthetic datasets with different characteristics and different workload generators, quantifying the benefits and overheads, and revealing a number of insights.
- Chapter 7 summarizes the contributions of this work from different perspectives, of algorithm/technique and system respectively, concluding a comprehensive solution for optimizing graph queries across the board.

1.4 Publications

The majority of the contents in this thesis has been accepted by the community through publications in related conferences/workshops.

- **P1:** J. Wang, N. Ntarmos, P. Triantafillou, "Indexing Query Graphs to Speedup Graph Query Processing", 19th International Conference on Extending Database Technology, (EDBT16), March 15-18, 2016.
  (This publication is included in Chapter 3.)
- **P2:** J. Wang, N. Ntarmos, P. Triantafillou, "GraphCache: A Caching System for Graph Pattern Queries", 20th International Conference on Extending Database Technology, (EDBT17), March 21-24, 2017.
  (This publication is included in Chapter 4.)
- **P3:** J. Wang, N. Ntarmos, P. Triantafillou, "Ensuring Consistency in Graph Cache for Graph-Pattern Queries", Sixth International Workshop on Querying Graph Structured Data (GraphQ 2017), with EDBT2017, March 2017.
  (This publication is included in Chapter 5.)
- **P4:** J. Wang, N. Ntarmos, P. Triantafillou, "Towards a Subgraph/Supergraph Cached Query-Graph Index", In Proc. IEEE International Conference on Big Data, (BigData2015), pp. 2919–2921, 2015 (poster paper).
  (This publication is included in Chapter 3.)

Chapter 2

Literature Review

As aforementioned, subgraph/supergraph query processing could be costly due to the entailed NP-Complete subgraph isomorphism problem. Researches usually fall into two categories, i.e., SI algorithms and FTV methods. The former uses heuristics so as to improve the efficiency of subgraph isomorphism tests themselves, whereas the latter targets at enhancing the filtering power such that fewer dataset graphs will undergo sub-iso testing. However, both SI and FTV solutions could render excessive query processing time, which in turn poses the fundamental performance limitation of graph queries.

On the other hand, caching of query results is prevalent in accelerating the performance of data management systems, from filesystem block caching to web proxy caching and the cache of query result sets in relational databases. But using cache to expedite subgraph/supergraph query processing has not been investigated yet.

This chapter shall review the relevant studies pertaining to graph query processing approaches and representative caching systems, highlight their differences from that of our work and position this thesis in literature in the end.

2.1 Graph Query Processing and Subgraph Isomorphism Problem

2.1.1 Problem Formulation

Graph queries, either in term of subgraph query processing or supergraph query processing, share the essence of dealing with NP-Complete subgraph isomorphism problem.

In this thesis, we consider undirected labelled graphs, as is typical in the literature (e.g., [16, 17, 18]). For simplicity, we assume that only vertices can have labels. All results and discussions straightforwardly generalize to the case of graphs with edge labels.

Definition 1. *A labeled graph $G = (V, E, l)$ consists of a set of vertices $V(G)$ and edges $E(G) = \{(u, v), u \in V, v \in V\}$, and a function $l : V \rightarrow U$, where U is the label set, defining the domain of labels of vertices.*

A sequence of vertices $(v_0, \ldots, v_n) : \exists(v_i, v_{i+1}) \in E$, constitutes a path of length n. A simple path is a path where no vertices are repeated. A cycle is a path of length $n > 1$, where $v_0 = v_n$. A simple cycle is a cycle which has no repeated vertices (other than v_0 and v_n). A connected graph is a graph in which there exists at least one path between any pair of its vertices.

Definition 2. *A graph $G_i = (V_i, E_i, l_i)$ is subgraph isomorphic to a graph $G_j = (V_j, E_j, l_j)$, by abuse of notation denoted by $G_i \subseteq G_j$, when there exists an injection $\phi : V_i \rightarrow V_j$, such that $\forall(u, v) \in E_i, u, v \in V_i, \Rightarrow (\phi(u), \phi(v)) \in E_j$ and $\forall u \in V_i, l_i(u) = l_j(\phi(u))$.*

Informally, there is a subgraph isomorphism $G_i \subseteq G_j$ if G_j contains a subgraph that is isomorphic to G_i. As is common in the relevant studies, we focus on non-induced subgraph isomorphism.

Definition 3. *Given two graphs $G_i = (V_i, E_i, l_i)$ and $G_j = (V_j, E_j, l_j)$, when $V_i \subseteq V_j$, $E_i \subseteq E_j$ and l_i is the projection of l_j onto the domain $V_i \cup E_i$, G_i is called a subgraph of (contained in) G_j denoted by $G_i \subseteq G_j$, or that G_j is a supergraph of (contains) G_i denoted by $G_j \supseteq G_i$.*

Definition 4. *The subgraph (supergraph) query problem entails a set $D = \{G_1, \ldots, G_n\}$ containing n graphs, and a query graph g, and determines all graphs $G_i \in D$ such that $g \subseteq G_i$ ($g \supseteq G_i$, respectively).*

2.1.2 Brute Force Approach: SI Methods

To process subgraph/supergraph queries, the brute force approach is to perform subgraph isomorphism tests, i.e., SI methods. The subgraph isomorphism problem is a generalization of graph isomorphism problem [9, 19], which has attracted considerable attention [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. As suggested by [9, 19], the classical subgraph isomorphism problem is a decision problem, which answers Y/N as to whether the query is contained in each graph in the dataset. Another version coined matching problem, which locates all occurrences of the query graph within a large graph (or a dataset of graphs), has also attracted the attention of community. The matching problem shares the same time complexity with that of the decision problem and could be viewed as an extension. Think of the example when a query g is not a subgraph of a dataset graph G. Whatever the problem to address, decision or matching, the system will have to enumerate all the possible cases, so as to finally

determine that g is not a subgraph of G (decision problem) and there is zero occurrence of g within G (matching problem). Currently, this work is focusing on the fundamental decision problem of subgraph isomorphism.

The first practical subgraph isomorphism algorithm is proposed by Ullmann [12] in 1976. For a given dataset graph G and a query graph g, a matrix M is created where $M[i, j]$ stores whether the i-th node in g matches the j-th node in G. Ullmanns algorithm employs a backtracking way to enumerate all matrices $M(s)$ such that $g = M(MG)^T$ where g and G refer to the adjacency matrix of query graph g and dataset graph G respectively, and T represents the operation of matrix transpose.

Subsequently, another influential SI algorithm VF2 [10] was proposed in 2004. By means of State Space Representation (SSR) [31], VF2 relies on the adjacency analysis between matched nodes and those not matched yet. Each state of matching process is associated to a partial mapping between the dataset graph G and query graph g. A transition adds one unused node of g to the used set according to the matching rules. One vector stores the matching information for one state. The depth-first search strategy assures that there can be at most N states in memory at a time, where N is the vertex number of query graph g. Hence, unlike Ullmann [12] that requires $O(N^2)$ memory for matrix, VF2 reduces the space cost to $O(N)$, which is significant when dealing with large graph queries.

Following the fundamental principle of VF2 [10], several heuristic algorithms have been proposed over the years. These heuristics usually fall into the three categories as follows.

- Optimizing the matching order of query vertices to minimize redundant Cartesian products: QuickSI [32] computes the global statistics of vertex label frequency and accesses query vertices with infrequent labels as early as possible; Unlike QuickSI's global matching order selection, TurboISO [33] divides candidate vertices into several regions and computes the local matching order within each region, following the rule of prioritizing query vertices with higher degrees and infrequent labels; CFL-Match [34] proposes a framework of postponing Cartesian products by considering the structure of query graph.

- Enlarging the pattern to be matched per partial mapping: SPath [35] proposes a path-at-a-time fashion, which proves to be more efficient than the traditional vertex-at-a-time methods.

- Grouping similar vertices to avoid duplicate computations: TurboISO [33] merges query vertices sharing the same label and same neighborhood; On the other hand, BoostIso [36] provides a framework to combine vertices in dataset graph based on their equivalence and containment relationships.

Furthermore, [11] provides an insightful presentation and comparison of several SI approaches. Through extensive experiments, a significant conclusion is drawn [11] – among a variety of SI algorithms of interest [10, 32, 17, 35], GraphQL [17] is the only solution that could scale with query size (up to 10 edges) when the dataset graph is relatively large (Human dataset [37] with ≈4.6K vertices and ≈86K edges per graph). This lies in the advanced heuristic strategies of GraphQL [17].

- Essentially, GraphQL employs pseudo subgraph isomorphism tests in an iterative manner, during the process of adding each vertex pair (i.e., (u, v), where u is the query vertex and v is the vertex in dataset graph) to form a partial mapping. In the first iteration, two breadth-first search trees T_u and T_v are obtained with depth of one ($d = 1$), where vertex v is discarded if T_u is not contained in T_v. Such process is iterated by increasing the depth d, until it reaches the predefined refinement level. By doing so GraphQL manages to throw redundant candidate vertices in dataset graphs at early stage, which in turn cut down the magnitude of intermediate Cartesian products and expedite the matching process in the end.

- Moreover, GraphQL utilizes the vertex signature of neighborhood labels to reduce the search space, which is followed by TurboISO [33] later.

The community has also looked into subgraph queries against a single, very large graph (consisting of possibly billions of nodes). [38] and [39] employ scale-out architectures and large memory clusters with massive parallelism respectively. [33] and [36] provide a centralized solution to the same problem via efficient heuristics.

Indeed, all SI approaches can be leveraged in the verification stage of FTV paradigm. Arguably, VF2 [10] is the most widely used nowadays by a large number of FTV methods [16, 40, 41, 18, 42]. To further alleviate the time consuming subgraph isomorphism verifications, most competitive FTV approaches manage to utilize information from the index (for those graphs that survive the filtering process) and render performance gains [18, 42].

Specifically, CT-Index [18] developed an upgraded version of VF2 (coined VF2+ in this work) for subgraph isomorphism verification. Compared with vanilla VF2, VF2+ is characterized by a number of additional heuristics.

- First, in the aforementioned index, each dataset graph is accompanied with an extended vertex sequence, construction of which follows two criteria: (i) each vertex is adjacent to a predecessor in the sequence whenever possible; (ii) vertices with rare labels are prioritized than their counterparts.

- Second, before performing the expensive operation of subgraph isomorphism tests, dataset graphs with fewer vertices/edges than query are discarded.

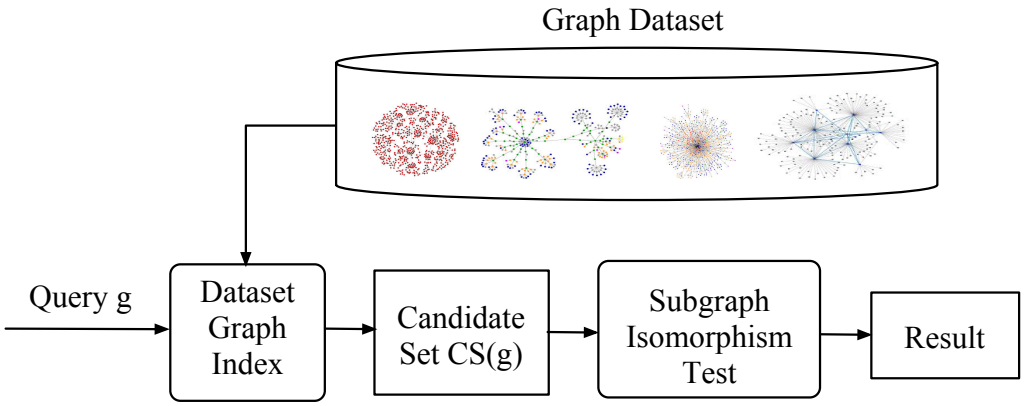


Figure 2.1: FTV Paradigm for Subgraph/Supergraph Query Processing

- Third, during the matching process, each partial mapping assures that the newly added vertex of dataset graph bears higher or equal degree than that of query graph.

Due to these effective heuristics, VF2+ [18] manages to substantially optimize the performance of subgraph isomorphism testing than the plain VF2 [10].

2.1.3 FTV Paradigm

When the dataset is comprised of a large number of graphs, SI algorithms deteriorate since each graph will have to be tested. Thus appeared the "filter-then-verify" (FTV) paradigm.

FTV methods try to reduce the set of graphs to run the subgraph isomorphism test, by filtering out graphs which definitely do not belong to the query answer set. At the heart of these methods lies an index on the dataset graphs. Figure 2.1 illustrates the three major stages of subgraph/supergraph query processing.

- Indexing: Dataset graphs are reduced to their features (i.e., substructures of the graph, such as paths, trees, cycles, or arbitrary subgraphs), which are then indexed in an appropriate data structure (e.g., trie, hash table, etc.).

- Filtering: Given a query graph g, g is also decomposed into its features, following the same process as for dataset graphs. Then the index is searched for g's features; for subgraph queries, the set of graphs that contain all of said features are returned, whereas for supergraph queries the returned set consists of graphs all of whose features are contained in g's features. This set is called the *candidate set*, coined as $CS(g)$.

- Verification: All known FTV algorithms guarantee that there will be no *false negatives*; that is, for subgraph (supergraph) queries, all graphs in the dataset that can possibly contain (resp. are contained in) the query graph g will be included in the candidate

set $CS(g)$. However, *false positives* are possible – not all graphs in the candidate set contain (resp. are contained in) the query graph. Hence, each graph in the candidate set $CS(g)$ must be verified by subgraph isomorphism test, so as to determine whether it falls into the final query result.

FTV approaches in the literature can be classified along two dimensions: whether they employ (frequent) mining techniques or an exhaustive enumeration for the production of features, and based on the type of features of the dataset graphs they index (e.g., paths, trees, subgraphs). Note that exhaustive enumeration can yield huge indices and may take a prohibitively long time to do so. For this reason, all exhaustive enumeration approaches limit the size of features to a typically fairly small number of edges (i.e., 10 or less).

Mining-based approaches, both for supergraph queries ([43, 44, 45, 46, 47]) and subgraph queries (e.g., [40, 48, 47]) utilize techniques to mine for frequent (or *discriminating*, in [46]) (sub)graphs among the dataset graphs that are then indexed. Other mining-based approaches like Tree+Δ [49] and TreePi [41] mine for and index frequent trees.

Lindex [50] and LWindex [51] utilize the frequent mining algorithms of previous approaches, and are thus able to index and query several feature types. Typically such approaches tend to mine for more complex structures, which presents a trade-off between the complexity and time required for the indexing process vis-a-vis the potential for higher pruning power during query processing. However, numerous related performance studies [14, 52, 42, 13, 18] have shown that feature-mining approaches tend to be comparatively worse performers.

On the other hand, SING[52], GraphGrep[53] and GraphGrepSX[16] perform exhaustive enumeration, listing all paths of dataset graphs up to a certain path length. Similarly, CT-Index[18] indexes trees and cycles, whereas Grapes[42] indexes paths along with location information.

A different approach, which does not index features as above, is presented in gCode[54]. For each graph G in the graph dataset, gCode computes a signature per vertex of G (essentially reflecting the vertex's neighborhood) and then computes a signature for G itself. The latter is a tree structure combining the signatures of all its vertices.

Recent performance studies [13, 14] have shown that CT-Index[18] and Grapes[42] are high performing approaches. CT-Index [18] is based on deriving canonical forms for the (tree, cycle) features of a graph G, to the fact that for trees and cycles finding string-based canonical forms can be done in linear time (unlike general graphs). These string representations of a graph's features are then hashed into a bitmap structure per graph G. Checking whether a query graph g can possibly be a subgraph of a graph G, can be done with simple and efficient bitwise operators between the bitmap of g and that of G (as supergraphs must contain all features of a subgraph). Last, in the verification stage, CT-Index upgrades the plain VF2 [10] and results the VF2+ algorithm for subgraph isomorphism testing (see details in §2.1.2).

Grapes [42] is designed to exploit parallelism available in multi-core machines. It exhaustively enumerates all paths (up to a maximum length), which are then inserted into a trie with their location information. This operation is performed in parallel by several threads, each of which works on a portion of the graph, producing its own trie, and subsequently all tries are merged together to form the path index of a graph. Grapes then computes (typically) small connected components of graphs in the candidate set, on which the verification (subgraph isomorphism test) is performed by employing VF2 algorithm [10].

An insightful discussion and comparative performance evaluation of several indexing techniques for subgraph query processing (published prior to 2010) can be found in [13]. Furthermore, [14] presented a systematic performance and scalability study of several older as well as current state-of-the-art index-based approaches for subgraph query processing. We are not aware of similar in-depth studies of solutions to supergraph query processing; however, [51] provides a concise overview of related approaches.

On a related note, recent work also deals with graph querying against historical graphs, identifying subgraphs enduring graph mutations over time [55], which can be viewed as a variation whereby graph snapshots in time can be viewed as different graphs. Note that [55] bears different scenarios than that of our work GraphCache+. The former is regarding a certain (fixed) query graph against a dataset (collection of the historical graphs), i.e., the basic subgraph query processing. Whereas GraphCache+ handles a much more complex case, in which continually arrived queries are executed against a dataset that keeps changing.

There has also been considerable work on approximate graph pattern matching. Relevant techniques (e.g., [18, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67]) perform (sub)graph matching with support for wildcards and/or approximate matches. These solutions are not directly related to our work, as we expedite exact subgraph/supergraph query processing.

Apriori [68] based pruning method (i.e., supergraphs of infrequent patterns are definitely infrequent) has been widely applied in frequent mining [69, 70, 71] to reduce searching space. This is expressed by the second optimal case of iGQ in our work. However, our approach covers more than that, e.g., previously issued queries, whatever the status over dataset graphs, will be well utilized by future queries.

2.2 Leveraging Cache to Accelerate Queries

Though caching of query results has long been a mainstay in data management systems, little work has been done in the realm of graph-structured queries.

For XML datasets, views have been used to accelerate path/tree queries [72, 73, 74]; Besides, [75] firstly proposed the MCR (maximally contained rewriting) approach for tree pattern

queries and [76] revisited it by providing alternatives; both exhibit false negatives for the query answer. Our work GraphCache does not produce any false negative or false positive. Also, GraphCache is capable of dealing with much more complex graph-structured queries, which entail the NP-Complete problem of subgraph isomorphism.

More recently, caching has also been utilized to optimize SPARQL query processing for RDF graphs. [77] introduced the first SPARQL cache, where a relational database was employed to store the metadata. [78] contributed a cache for SPARQL queries based on a novel canonical labelling scheme (to identify cache hits) and on a popular dynamic programming planner [79]. Similar to GraphCache, query optimization in [78] does not require any a priori knowledge on datasets/workloads and is workload adaptive.

However, like XML queries, SPARQL queries are less expressive than general graph queries and thus less challenging [80, 39]; SPARQL query processing consists of solving the subgraph homomorphism problem, which is different from the subgraph isomorphism problem, as the former drops the injective property of the latter. Moreover, GraphCache discovers subgraph, supergraph, and exact-match relationships between a new query and the queries in the cache, something that the canonical labelling scheme in [78] fails to achieve. SPARQL query processing also aims at optimizing join execution plans [81] (based on join selectivity estimator statistics and related cost functions), and the cache in [78] is focusing on this goal, whereas GraphCache aims to avoid/reduce costs associated with executing SI heuristics whose execution time can be highly unpredictable and much higher. As such, the overall rationale of GraphCache and the way cache contents are exploited differs from that in [78] and in related SPARQL result caching solutions.

On a related note, [82] presents a cache for historical queries against a large social graph, in which each query is centered around a node in the social graph, and where the aim is to avoid maintaining/reconstructing complete snapshots of the social graph but to instead use a set of static "views" (snapshots of neighborhoods of nodes) to rewrite incoming queries. [82] does not deal with subgraph/supergraph queries per se; rather, the nature of the queries means that containment can be decided by just measuring the distance of the central query node to the centre of each view. Moreover, [82] does not deal with central issues of a cache system (cache replacement, admission control, overall architecture/design, etc.).

Central to cache management is the replacement policy. Indeed, there are an abundance of cache replacement policies that usually target at fast data access. In this case, each cache hit saves one disk I/O. However, the scenario of GraphCache is different and much more complex – each cache hit shall evoke various numbers of savings in subgraph isomorphism testing, which could in turn render quite different query processing time. Such special circumstance had not yet been investigated by any of the policies in literature. GraphCache hence contributes a number of exclusive replacement strategies that are graph query aware;

details shall be presented in §4.3.3.

Finally, an advanced topic of caching system is to ensure cache consistency. [83] first explicitly specified the consistency constraint in a query-centric approach and presented how it could be expressed succinctly in SQL. Microsoft SQL Server provides a solution for the application server to register queries with database and receive notifications from database upon changes of query result, so as to ensure that query results cached at the application server are up-to-date. Correspondingly, there are studies of cache consistency regarding XML datasets [84] and SPARQL query processing [85]. However, the topic of ensuring graph cache consistency for general subgraph/supergraph queries had not been discussed yet.

2.3 Summary

This chapter has summarized the solutions of graph query processing and reviewed the researches regarding utilizing cache to expedite queries. The challenge of subgraph/supergraph queries stems from the entailed subgraph isomorphism problem, which is NP-Complete [15]. Hence formed two categories of solutions, where SI methods rest on heuristics to accelerate the matching process of subgraph isomorphism and FTV algorithms focus on pruning out dataset graphs that can not possibly enter the final query result before performing the costly subgraph isomorphism verification. However, they both suffer performance limitations.

Therefore, a general solution is on demand such that it could ensure high performance of graph queries across a variety of datasets, workloads and algorithm contexts. Thus emerges the fundamental challenges of technique and system.

- A novel principle of expediting graph queries is required, such that (i) it could afford the elegance of double use for subgraph/supergraph query processing; (ii) it should have the capability of incorporating and exceeding every subgraph/supergraph literature, be it an SI or FTV method; (iii) it could achieve good performance across various datasets, workloads and algorithms.

- A comprehensive system is required to evaluate the applicability and appropriateness of the aforementioned solution, using a large number of queries. Ideally, this system could back on well-established principles, of which the central issues will be addressed in the circumstance of dealing with graph query processing. Moreover, the system should come with good designs and implementations, assuring the flexibility of easy plug-in and the convenience to explore advanced topics.

Chapter 3

Indexing Query Graphs to Speed Up Graph Query Processing

To expedite graph queries, the community has continuously contributed novel SI and FTV approaches. As to the framework design, FTV improves by adding a filtering stage on top of standard SI algorithms. Filtering makes fewer graphs undergo the test for subgraph isomorphism and reduces the query processing time. But solutions in literature still suffer performance limitations.

For this reason, this chapter starts from analyzing the performance of state of the art FTV methods, identifying the bottleneck of graph query processing – the overwhelming verification time. Such conclusion naturally applies for SI solutions, since their query processing is covered by verification only.

A significant drawback of current solutions lies in the fact that they all throw away executed queries without exploiting the knowledge to accelerate future query processing. Moreover, a large number of real-world applications indicate that graph queries submitted in the past share subgraph/supergraph relationships with future queries.

Why not make use of the knowledge accrued by previous queries to facilitate the overwhelmingly expensive verifications for subgraph isomorphism? Base on these considerations, this chapter shall present a fresh perspective of indexing graph queries (iGQ) to speed up query processing, which is applicable for both FTV algorithms and SI methods of handling subgraph/supergraph queries.

3.1 Avoiding the Obstacles

Graph query processing could render expensive execution time, for essentially solving the NP-Complete subgraph isomorphism problem. The current work targets at optimizing graph

queries through investigating a general solution. Hence, this chapter shall first employ state-of-the-art approaches to identify the performance bottleneck, i.e., what makes graph queries costly, towards which the perspective of this work will be positioned.

3.1.1 Insights

Here reports on the fundamentals of the performance of three state-of-the-art FTV approaches, GraphGrepSX [16] (GGSX), Grapes [42], and CT-Index [18], over two real graph datasets AIDS [86] and PDBS [87]. Characteristics of these algorithms and datasets shall be detailed in §6. Briefly, AIDS is a graph dataset consisting of 40,000 graphs with averaged edge size 47, while PDBS is a graph dataset containing 600 graphs with 3,064 edges on average. Please note that the way the queries were generated is standard among related work [42, 18].

Subgraph Query Performance: Where Does Time Go?

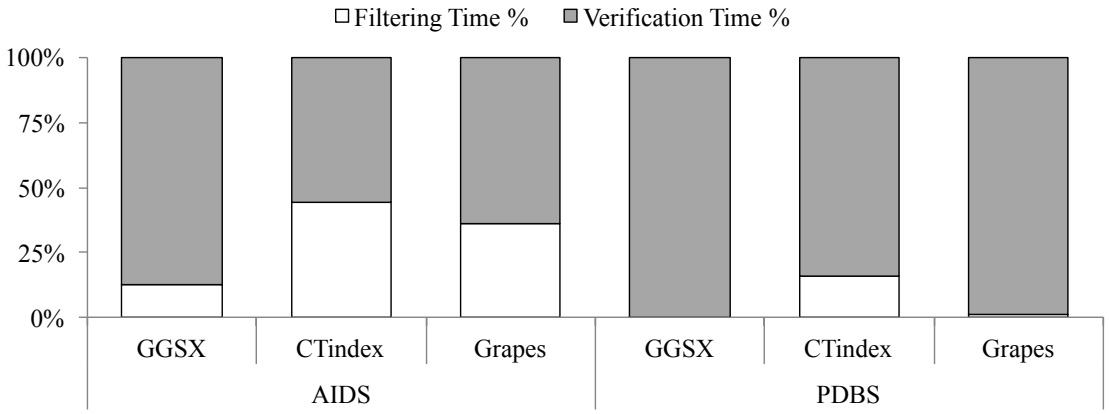


Figure 3.1: Dominance of the Verification Time on the Overall Query Processing Time of Three FTV Algorithms on Two Different Real-world Graph Datasets

There are two key components of the overall query processing time: filtering time (to process the index and produce the candidate set) and verification time (to perform the verification of all candidate graphs). Figure 3.1 shows what percentage of the total query processing time is attributed to each component.

The dominance of the verification step is clear. This holds across the three different approaches that employ different indexing methods and utilize different strategies for cutting down the cost of subgraph isomorphism. Recall that subgraph isomorphism performance is highly sensitive to the size of both the input graph and the stored graph. Hence, one would expect that for smaller stored graphs (as in the AIDS dataset) the verification step would be much faster. Notably, however, even when graphs are very small, the verification step is the biggest performance inhibitor and as graphs become larger (e.g., PDBS) the verification

step becomes increasingly responsible for nearly the total query processing time. Of course, given the NP-Completeness of subgraph isomorphism, one would expect that verification would dominate, especially for large graphs. But the fact that even with very small graphs this holds is noteworthy.

Filtering Power: Is It Good Enough?

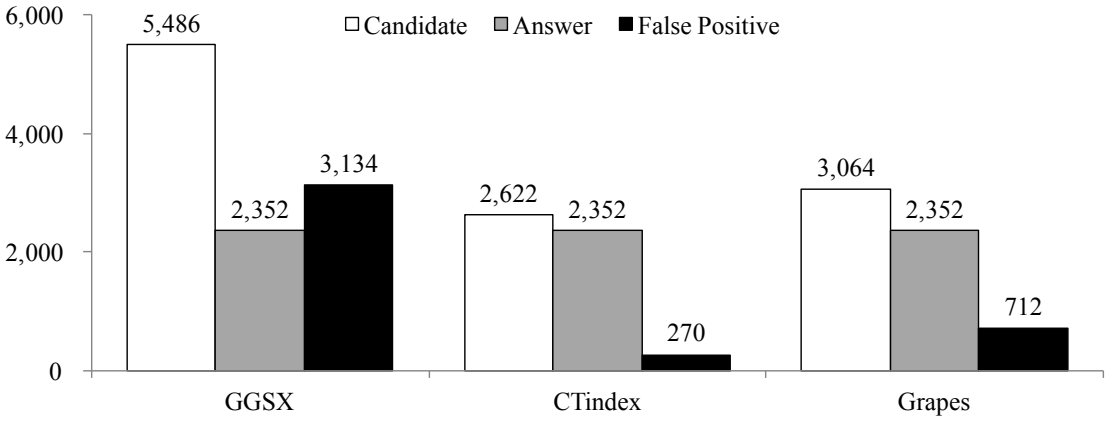


Figure 3.2: Average Number of Candidate Set Size, Answer Set Size, and False Positives in the AIDS Dataset

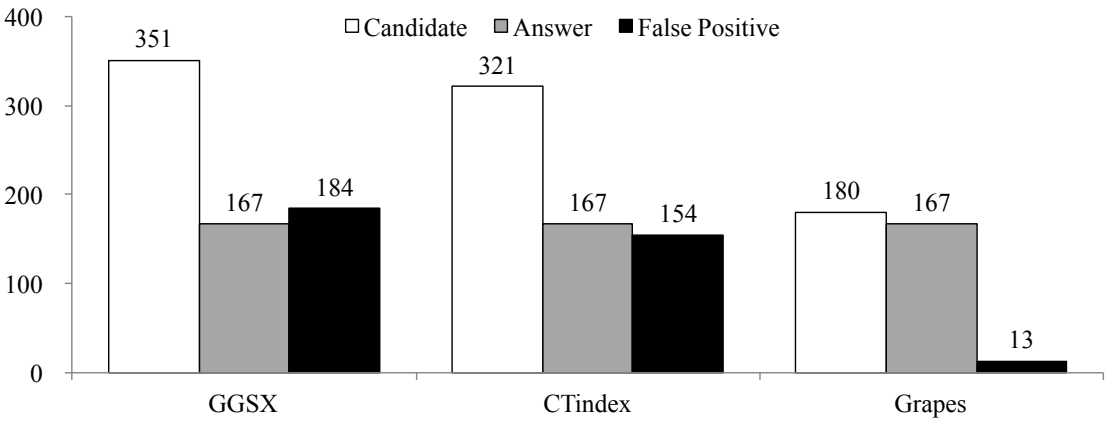


Figure 3.3: Average Number of Candidate Set Size, Answer Set Size, and False Positives in the PDBS Dataset

The second fundamental point pertains to how the verification cost can be reduced. Related works highlight that their approaches prove to be very powerful in terms of filtering out the vast majority of dataset graphs. Figures 3.2 and 3.3 show the results with respect to the average size of candidate sets and of the answer set, as well as the average number of false positives for the AIDS and PDBS datasets.

First, note that different algorithms behave differently in different datasets (e.g., Grapes significantly outperforms CT-Index in PDBS while the reverse holds for AIDS). Second, note

that despite the powerful filtering of an approach, when the dataset contains a large number of graphs (see Figure 3.2) in absolute numbers, there is a very large number of unnecessary subgraph isomorphism tests (i.e., false positives) that is required. The above two combined imply that even the best algorithm will suffer from a large number of unnecessary subgraph isomorphism tests under some datasets.

Turning the attention to Figure 3.3 one see that for datasets with medium to small number of graphs, the high filtering power can indeed result in requiring only a relatively small number of subgraph isomorphism tests. However, considerable percentages of false positives can appear in the candidate sets of even top-performing algorithms; e.g., CT-Index, which exhibited the best filtering in the AIDS dataset, has an almost 50% false positive ratio in the PDBS dataset. Furthermore, not all subgraph isomorphism tests for the graphs in the candidate set are equally costly. As the cost of subgraph isomorphism testing depends on the size of the graph, the larger graphs in the candidate set contribute a much greater proportion of the total cost of the verification step. Note that, naturally, false positive graphs tend to be the largest graphs in the dataset, since these have a higher probability to contain all features of query graphs.

Note that this work placed emphasis on the number of unnecessary subgraph isomorphism tests (i.e., the false positives), as filtering can be further improved by reducing this number. However, it is not the only source of possible improvements; iGQ can improve on the number of subgraph isomorphism tests even beyond this, by exploiting knowledge gathered during query execution (to be detailed later).

The insights that can be drawn are as follows:

- Despite the fact that state-of-the-art techniques (based on indexing features of dataset graphs) can enjoy high filtering capacity, there is still large room for improvement, as even the best approaches may perform large numbers of unnecessary subgraph isomorphism tests.
- Improving further the filtering power of approaches can significantly reduce query processing time, as this will cut down the number of subgraph isomorphism tests, which dominates the overall querying time.
- Even approaches that are purported to enjoy great filtering powers, can behave much more poorly under different datasets.
- Unnecessary subgraph isomorphism tests are not solely caused by false positives; even graphs in the candidate set that are true positives can be unnecessarily tested if the system fails to exploit this knowledge (accrued by previous query executions).

3.1.2 iGQ Perspectives

Providing the analyses on the graph query obstacles, the current work shall offer a new perspective to improve the performance of subgraph/supergraph query processing, with insights pertaining to utilizing executed queries to accelerate future query processing. This approach rests on the following three observations.

- In related works, there exists an implicit assumption that graph queries will be similarly structured to the dataset graphs. In general this is not guaranteed to hold (e.g., in exploratory analytics), and when query graphs have no match in the dataset graphs, query processing cannot benefit at all from indexes that are solely constructed on dataset graphs.
- Even when query graphs have matches against dataset graphs, the system performs expensive computations during query processing and simply throws away all (painstakingly and laboriously) derived knowledge (i.e.,previous query results).
- The success of known approaches depends on and exploits the fact that dataset graphs share features (e.g., when mining for frequent features) and/or that dataset graph features contain or are contained in other graph features (e.g., when using tries to index dataset graph features). However, they completely fail to investigate and exploit such similarities between query graphs.

Furthermore, many applications indicate that new queries could share subgraph/sueprgraph relationships with previous queries; see examples in §1.1. The above combined motivates iGQ of this work.

Instead of "mining" only the stored graphs and creating relevant indexes on them, iGQ also "mine" *query graphs* and accumulate the knowledge produced by the system when running queries, creating a **query index** in addition to the dataset index. The insights identify which is the relevant accumulated knowledge and how to exploit it during query processing in order to further reduce the number of subgraph isomorphism tests. iGQ is capable of accommodating any proposed approach for subgraph/supergraph query processing and help expedite both query categories.

3.2 iGQ Frameworks for Subgraph Queries

iGQ aims to augment the functionality and benefits offered by any one of the subgraph and/or supergraph query processing methods in the literature. Say, the chosen method is called $\mathbb{M}$.

The iGQ framework consists of method $\mathbb{M}$ and the two components of $\mathbb{I}$, namely $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$.

For the sake of simplicity, we shall first describe the operation of iGQ when $\mathbb{M}$ is a method for subgraph query processing (denoted $\mathbb{M}_{sub}$) here and leave the frameworks for supergraph query processing in §3.3.

Initially, method $\mathbb{M}_{sub}$ builds its graph dataset index as per usual. The iGQ index, $\mathbb{I}$, starts off empty; it is then populated as queries arrive and are executed by $\mathbb{M}_{sub}$.

Upon the arrival of a query g, the query processing is parallelized. One thread uses method $\mathbb{M}_{sub}$'s algorithms and indexing structure to breakdown the query graph into its features, and uses its index to produce a candidate set of graphs, $CS(g)$, as usual. Additionally, $\mathbb{I}$ will obtain as many of the intermediate and final results from method $\mathbb{M}_{sub}$'s execution as possible; e.g., it will obtain the features of the query graph, to be compared to those stored in $\mathbb{I}$ (from previously-executed queries). At this point, two separate threads will be created: one will check whether the query graph is a subgraph of previous query graphs and the other will check whether it is a supergraph of previous query graphs. These cases yield different opportunities for optimization and are discussed separately below.

The following proceeds to describe the function of each component of the iGQ framework and how it is all brought together. For the formal proofs of correctness that follow, for simplicity, assumptions are made as follows.

Assumptions The iGQ index components, $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$ work correctly. That is:

$$g' \in \mathbb{I}_{sub}(g) \Rightarrow g \subseteq g' \tag{3.1}$$

and

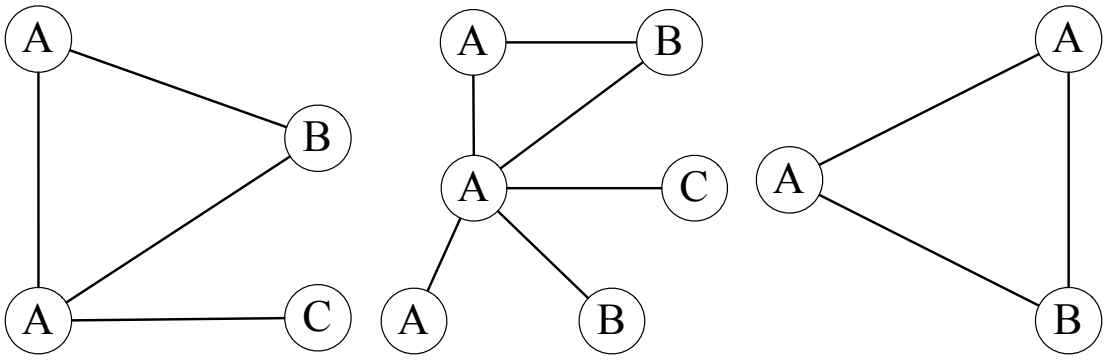
$$g'' \in \mathbb{I}_{super}(g) \Rightarrow g \supseteq g'' \tag{3.2}$$

Figure 3.4 presents a running example showing the status of $g \subseteq g'$ and $g \supseteq g''$, namely g *is a subgraph of* g' *(*g' *contains* g*)* and g *is a supergraph of* g'' *(*g *contains* g''*)*. §3.4.1 and §3.4.2 shall prove that these assumptions hold.

3.2.1 The Subgraph Case: $\mathbb{I}_{sub}$

This case occurs when a new query g is a subgraph of a previous query g'. When g' was executed by the system, the $\mathbb{I}_{sub}$ component of iGQ indexed g''s features. Additionally, iGQ stored the results computed by $\mathbb{M}_{sub}$ for g'.

Figure 3.5 depicts an example for the subgraph case of iGQ. A new query g is "sent" to method $\mathbb{M}_{sub}$'s graph index, producing a candidate set, $CS(g)$, which in this case contains



(a) New Query g (b) Previous Query g' (c) Previous Query g''

Figure 3.4: An Example: the New Query g with Two Previous Queries g' and g''; g is a subgraph of g' ($g \subseteq g'$) and a supergraph of g'' ($g \supseteq g''$).

the four graphs $\{G_1, G_2, G_3, G_4\}$. Similarly, g is "sent" to the iGQ subgraph component, $\mathbb{I}_{sub}$, from where it is determined that there exists a previous query g', such that $g \subseteq g'$. iGQ then retrieves the answer set, $Answer(g')$ (previously produced by method $\mathbb{M}_{sub}$ and indexed by $\mathbb{I}_{sub}$); in this case, $Answer(g') = \{G_1,\ G_2\}$.

The reasoning then proceeds as follows. Consider graph $G_1 \in CS(g)$. Since from $\mathbb{I}_{sub}$ it has been concluded that $g \subseteq g'$ and from the answer set of g' we know that $g' \subseteq G_1$, it necessarily follows that $g \subseteq G_1$. Similarly, it concludes that $g \subseteq G_2$. Hence, there is no point in testing g for subgraph isomorphism against G_1 or G_2, as the answer is already known. Therefore, one can safely subtract graphs G_1, G_2 from $\mathbb{M}_{sub}$'s candidate set, and test only the remaining graphs (reducing the number of subgraph isomorphism tests in this example by 50%). After the verification stage, G_1, G_2 are added to the final answer set.

Turing our attention to G_3. Providing $Answer(g') = \{G_1, G_2\}$, it naturally follows $g' \nsubseteq G_3$. Though $\mathbb{I}_{sub}$ has concluded that $g \subseteq g'$, whether G_3 contains g is mysterious. Still, the queries g and g' in Figure 3.4 are used for illustration. As shown by Figure 3.6, there could exist both G_{3y} and G_{3n} satisfying the requirement of $g' \nsubseteq G_3$. However, G_{3y} and G_{3n} result different outcomes as to the status of G_3 and g – apparently, G_{3y} contains g whereas G_{3n} does not. Therefore, G_3 has to be left for subgraph isomorphism test, in order to conclude the final status. For the same reason, G_4 will be tested for subgraph isomorphism as well.

In the general case, g may be a subgraph of multiple previous query graphs g'_i in $\mathbb{I}_{sub}$. Following the above reasoning, we can safely remove from $CS(g)$ all graphs appearing in the answer sets of all query graphs g'_i, as they are bound to be supergraphs of g; that is, the set of graphs submitted by iGQ for subgraph isomorphism testing is given by:

$$CS_{sub}(g) = CS(g) \setminus \bigcup_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i) \tag{3.3}$$

where each graph g'_i is such that $g'_i \in \mathbb{I}_{sub}(g)$, with answer set $Answer(g'_i)$ (when g'_i was

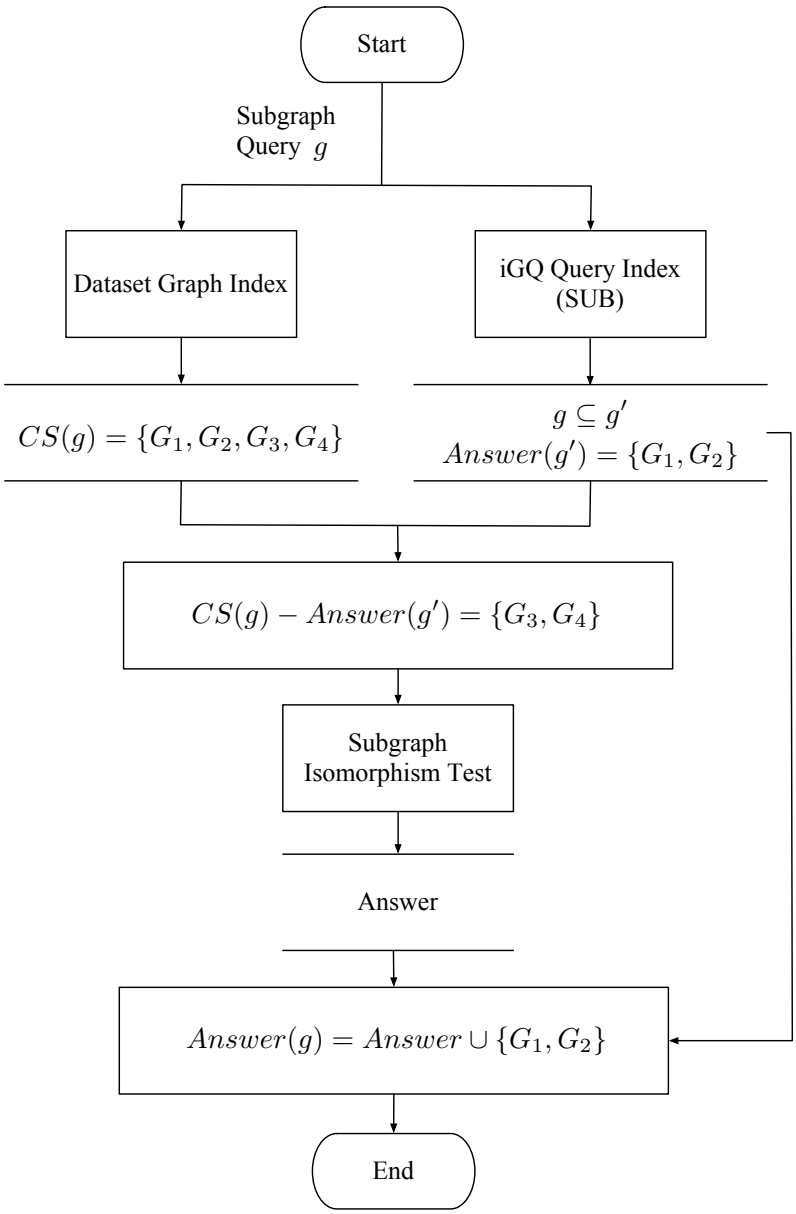


Figure 3.5: iGQ Subgraph Case for Subgraph Query Processing (when g is a Q_{sub})

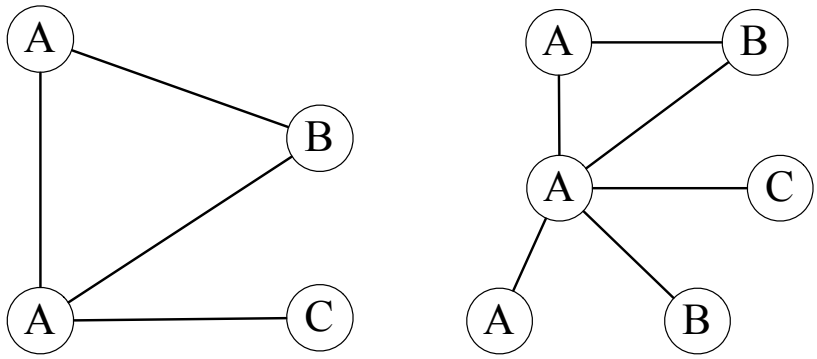
processed by $\mathbb{M}_{sub}$). The set union operation on $Answer(g'_i)$ is based on the fact that for any $g'_i \in \mathbb{I}_{sub}(g)$ with $G \in Answer(g'_i)$, it applies with $g \subseteq G$.

Finally, if $Answer_{sub}(g)$ is the subset of graphs in $CS_{sub}(g)$ verified to be containing g through subgraph isomorphism testing, the final answer set for query g will be:

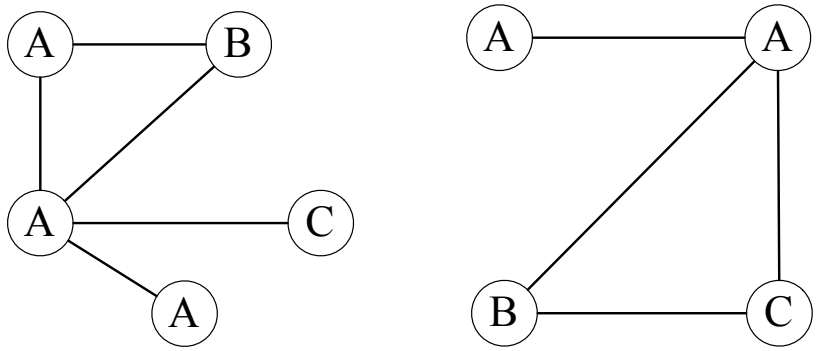
$$Answer(g) = Answer_{sub}(g) \cup \bigcup_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i) \tag{3.4}$$

Lemma 1. *The iGQ answer in the subgraph case does not contain false positives.*

Proof. Assume that a false positive was produced by iGQ; particularly, consider the first



(a) New Query g (b) Previous Query g'



(c) G_{3y} (d) G_{3n}

Figure 3.6: When g is a Q_{sub}, the Uncertain Status of Dataset Graph G_3: $g \subseteq G_{3y}$ and $g \nsubseteq G_{3n}$

ever false positive produced by $\mathbb{I}_{sub}$, i.e., for some query g, $\exists G_{FP}$ such that $g \nsubseteq G_{FP}$ and $G_{FP} \in Answer(g)$. Note that G_{FP} cannot be in $Answer_{sub}(g)$, as the latter contains only those graphs from $CS_{sub}(g)$ that have been verified to be supergraphs of g after passing the subgraph isomorphism test, and hence $g \nsubseteq G_{FP} \Rightarrow G_{FP} \notin Answer_{sub}(g)$. Therefore, by formula (3.4), $G_{FP} \in Answer(g) \Rightarrow \exists g'$ such that $g' \in \mathbb{I}_{sub}(g)$ and $G_{FP} \in Answer(g')$. But by equation (3.1) $g' \in \mathbb{I}_{sub}(g) \Rightarrow g \subseteq g'$, and $G_{FP} \in Answer(g') \Rightarrow g' \subseteq G_{FP}$. Thus $g \subseteq G_{FP}$ (a contradiction). □

Lemma 2. *iGQ in the subgraph case does not introduce false negatives.*

Proof. Assume that a false negative was produced by iGQ; particularly, consider the first ever false negative produced by $\mathbb{I}_{sub}$, i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. As method $\mathbb{M}_{sub}$ is assumed to be correct, it cannot produce any false negatives when processing query g, hence $g \subseteq G_{FN} \Rightarrow G_{FN} \in CS(g)$. Then, the only possibility for error is that G_{FN} was removed using formula (3.3); i.e., $G_{FN} \notin CS_{sub}(g)$. That implies that $\exists g'$ such that $g' \in \mathbb{I}_{sub}(g)$ and $G_{FN} \in Answer(g')$. But then, by formula (3.4), G_{FN} will be added to $Answer_{sub}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). □

Theorem 1. *The iGQ answer in the subgraph case of query processing is correct.*

Proof. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 1 and 2. □

Interpreting the Subgraph Case: Where Does Benefit Come From?

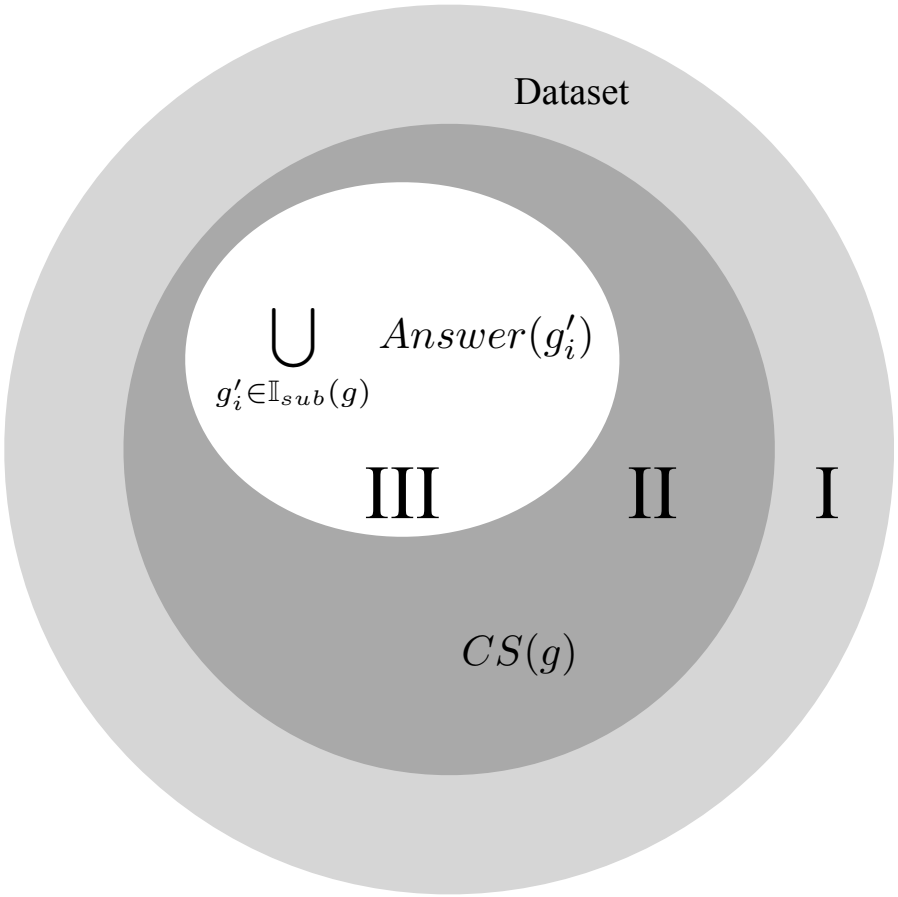


Figure 3.7: Benefit Analysis: iGQ Subgraph Case when g is a Q_{sub} (Areas Satisfy III $\subseteq$ II $\subseteq$ I; Each Notation is inside its Area.)

By pruning the candidate set, iGQ manages to expedite subgraph queries. More specifically, the subgraph case identifies dataset graphs that definitely contain the new query, exempting their testings for subgraph isomorphism.

Figure 3.7 shows the source of benefit for subgraph case when g is a Q_{sub}. The complete set, i.e., area I, covers all the graphs in dataset. Inside I, there exists a subset II denoting the candidate set for new query g, i.e., $CS(g)$. If there were no iGQ, each dataset graph in $CS(g)$ will have to undergo the expensive subgraph isomorphism tests to determine the answer set of g. iGQ, however, could utilize the knowledge derived from previous queries and detect those test-free graphs among $CS(g)$. Such graphs are covered by the area III in Figure 3.7. Recall formula (3.3). III represents exactly the minus part $\bigcup_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i)$, which

shall be added to the final answer of query g by equation (3.4). In other words, for subgraph query processing, benefit of iGQ in the subgraph case rests on the covering of area III – larger III renders higher benefit.

3.2.2 The Supergraph Case: $\mathbb{I}_{super}$

This case occurs when a new query g is a supergraph of a previous query g''. Figure 3.9 depicts an example for the supergraph case of iGQ. Again, the subgraph query processing method $\mathbb{M}_{sub}$ produces a candidate set, $CS(g)$ that, say, contains four graphs $\{G_1, G_2, G_3, G_4\}$.

Running g through $\mathbb{I}_{super}$, it is determined that there exists a previous query graph g'' such that $g'' \subseteq g$. Also $\mathbb{I}_{super}$ supplies the stored answer set for g'', $Answer(g'') = \{G_1, G_{20}\}$.

The reasoning then proceeds as follows. Consider graph $G_2 \in CS(g)$. We know from $\mathbb{I}_{super}$ that $G_2 \notin Answer(g'')$. Now, if $g \subseteq G_2$ were to indeed be true, since $g'' \subseteq g$, then it must also hold that $g'' \subseteq G_2$; that is, $Answer(g'')$ would have to contain G_2 as well, which is a contradiction. Therefore, it is safe to conclude that $g \nsubseteq G_2$ and thus G_2 can be safely removed from $CS(g)$. Similarly, one can also safely remove graphs G_3, G_4 from $CS(g)$, reducing in this case the number of required subgraph isomorphism tests by 75%.

Next, consider $G_1 \in CS(g)$. Since $Answer(g'') = \{G_1, G_{20}\}$, $g'' \subseteq G_1$ is at hand. And $\mathbb{I}_{super}$ has discovered that $g \supseteq g''$. However, the status of G_1 and g is still unknown. Again, using queries g and g'' in Figure 3.4, Figure 3.8 shows the possible instances of G_1 such that $g \subseteq G_{1y}$ and $g \nsubseteq G_{1n}$. As a result, G_1 will have to undergo subgraph isomorphism testing.

In the general case, g may be a supergraph of multiple previous query graphs g_i'' in $\mathbb{I}_{super}$. By the above reasoning, only those graphs appearing in the answer sets of all queries g_i'' may actually be supergraphs of g; thus the set of graphs submitted by iGQ for subgraph isomorphism testing is:

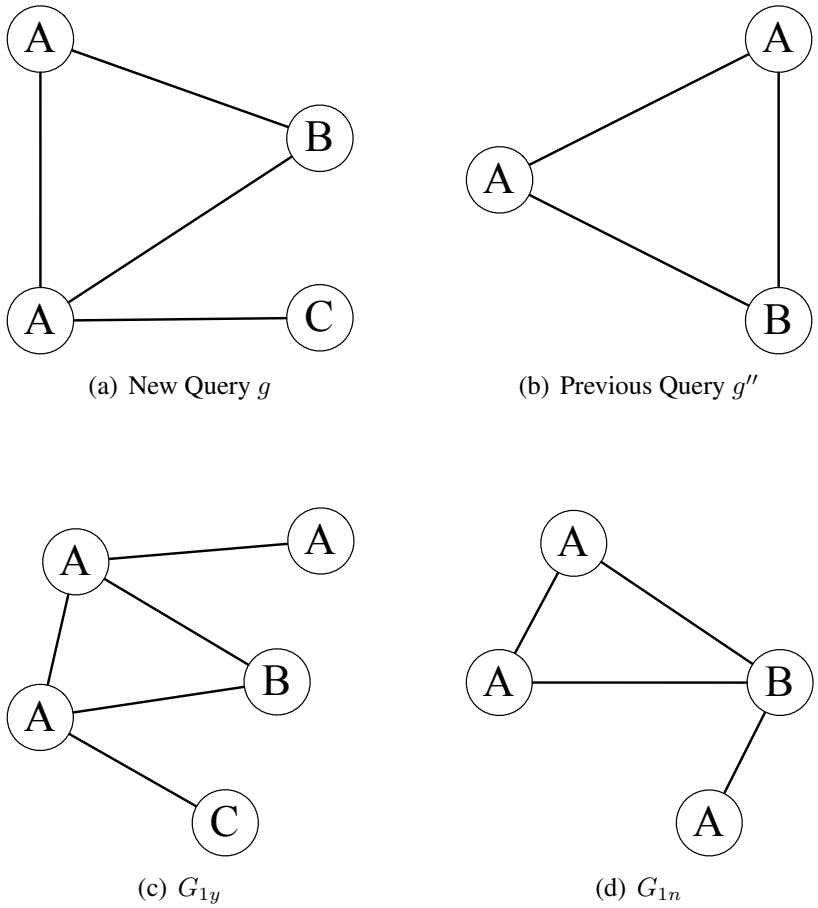
$$CS_{super}(g) = CS(g) \cap \bigcap_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'') \tag{3.5}$$

where $g_i'' \in \mathbb{I}_{super}(g)$, with answer set $Answer(g_i'')$ (when g_i'' was processed by $\mathbb{M}_{sub}$). It is intersection on multiple sets $Answer(g_i'')$, because only those stored graphs containing all $g_i'' \in \mathbb{I}_{super}(g)$ could possibly contain g.

The final answer produced for query g by iGQ, $Answer(g)$, will be the subset of graphs in $CS_{super}(g)$ that have been verified by the subgraph isomorphism test.

Lemma 3. *The iGQ answer in the supergraph case does not contain false positives.*

Proof. This trivially follows by construction as all graphs in $Answer(g)$ have passed through subgraph isomorphism testing at the final stage of processing. □



(a) New Query g (b) Previous Query g''

(c) G_{1y} (d) G_{1n}

Figure 3.8: When g is a Q_{sub}, the Uncertain Status of Dataset Graph G_1: $g \subseteq G_{1y}$ and $g \nsubseteq G_{1n}$

Lemma 4. *The iGQ answer in the supergraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by $\mathbb{I}_{super}$; i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. Method $\mathbb{M}_{sub}$ does not produce in its candidate set any false negatives, hence $G_{FN} \in CS(g)$. Then, the only possibility for error is for iGQ to have removed graph G_{FN} from $CS_{super}(g)$ with formula (3.5). This implies that $\exists g''$ such that $g'' \in \mathbb{I}_{super}(g)$ and $G_{FN} \notin Answer(g'')$. But since $g'' \in \mathbb{I}_{super}(g)$, by equation (3.2), $g'' \subseteq g$, and then $g \subseteq G_{FN} \Rightarrow g'' \subseteq G_{FN} \Rightarrow G_{FN} \in Answer(g'')$ (a contradiction). □

Theorem 2. *The iGQ answer in the supergraph case of query processing is correct.*

Proof. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 3 and 4. □

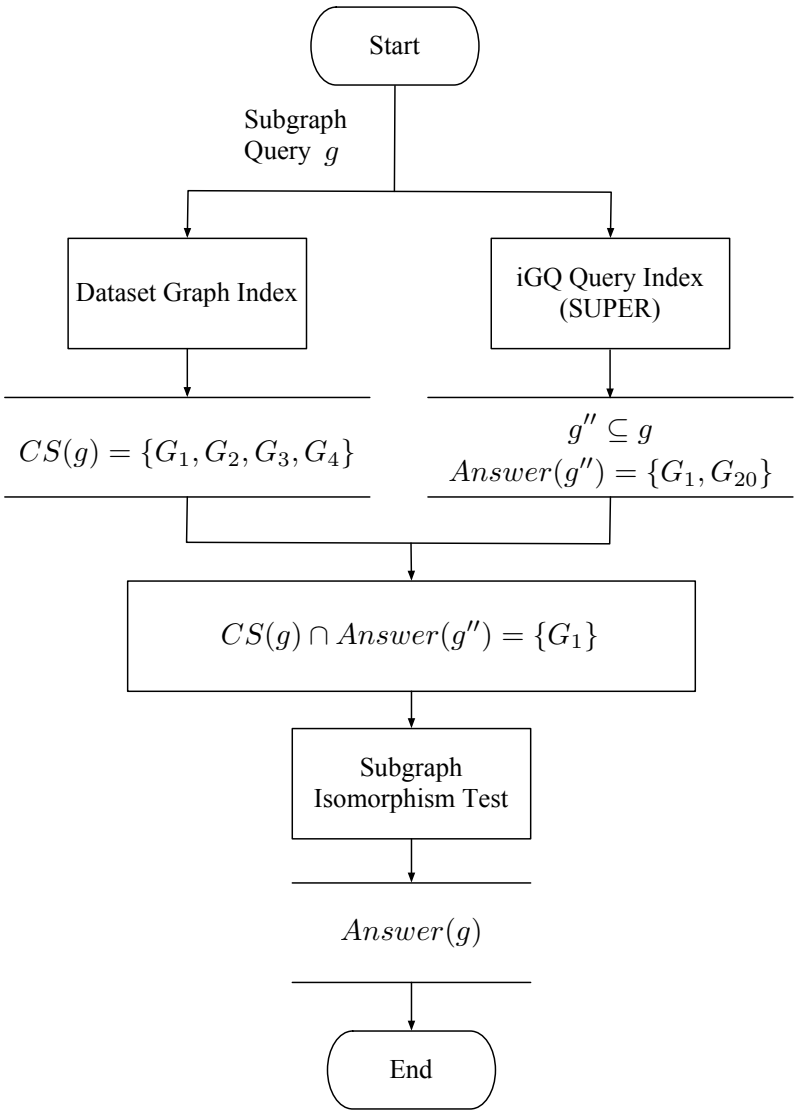


Figure 3.9: iGQ Supergraph Case for Subgraph Query Processing (when g is a Q_{sub})

Interpreting the Supergraph Case: Where Does Benefit Come From?

Like in the subgraph case, the supergraph case accelerates query processing by reducing the candidate set. Whereas in the supergraph case, by making use of the previous queries, graphs in the candidate set are further differentiated – some are determined that they can never contain the query and hence no need for subgraph isomorphism testing, while the remaining will be tested as per usual.

Figure 3.10 interprets the iGQ benefit for supergraph case when g is a Q_{sub}. The area I covers the whole dataset. Note that the candidate set $CS(g)$ is covered by a round area II and dataset graphs in $\bigcap_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'')$ are included by another round area III. Apparently, both II and III are inside area I. The intersection of II and III is coined as IV, i.e., $CS(g) \cap \bigcap_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'')$ in formula (3.5), which consists of the new candidate set. Performing subgraph isomorphism tests on graphs in IV hence evicts the answer set of query

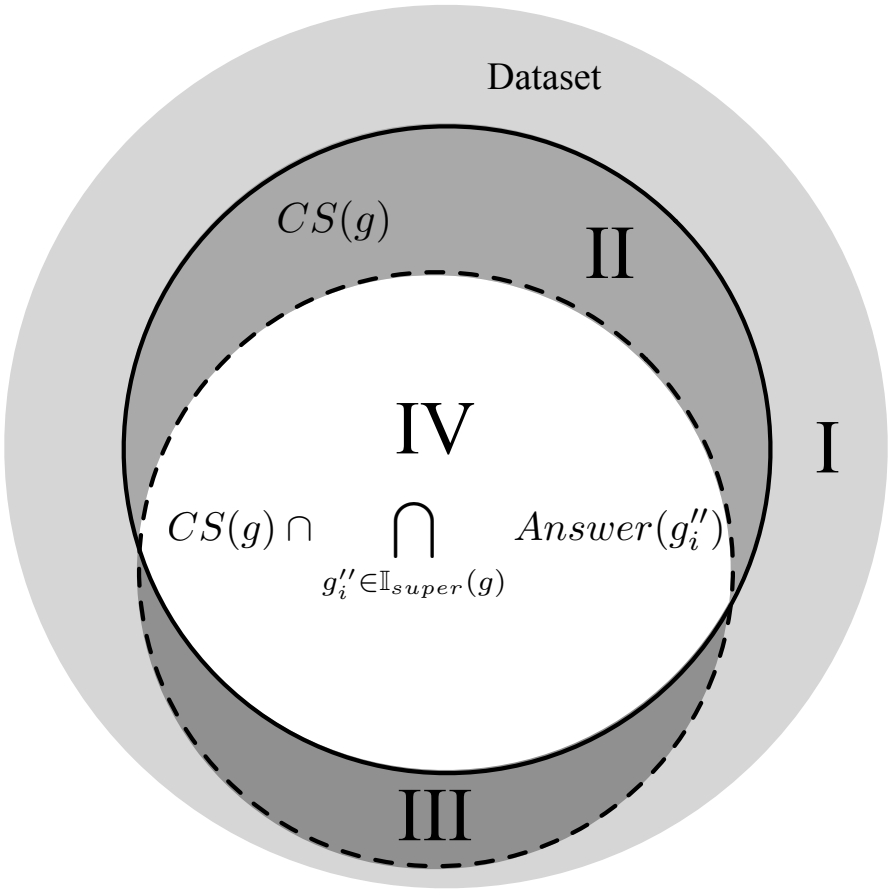


Figure 3.10: Benefit Analysis: iGQ Supergraph Case when g is a Q_{sub} (Areas Satisfy II $\subseteq$ I, III $\subseteq$ I, IV $\subseteq$ II and IV $\subseteq$ III; I, II and III are Round; Each Notation is inside its Area; II Has a Solid Border and III is Enclosed by Dashes.)

g. Therefore, for subgraph queries, iGQ supergraph case effects by subtracting candidates that do not fall into the area IV; hence smaller IV results higher benefit.

3.2.3 Two Optimal Cases

For subgraph queries, there are two special cases that warrant further emphasis, since they introduce the greatest possible benefits.

First, note that iGQ can easily recognize the case where a new query, g, is exactly the same as a previous query contained in $\mathbb{I}$. Specifically, this holds when $\exists g' \in \mathbb{I}$ such that $g \subseteq g'$ or $g \supseteq g'$, and g and g' have the same number of nodes and edges. When this holds, since $\mathbb{I}$ stores the result for g', we can return directly and completely avoid the subgraph isomorphism testing as the actual result for g is known! As the subgraph isomorphism test dominates the query execution time, this is expected to be a large performance improvement.

Second, consider the supergraph case of iGQ. If $\exists g' \in \mathbb{I}_{super}(g)$ such that $g' \subseteq g$ and $Answer(g') = \emptyset$, then we can completely omit the verification stage again: If there were

a dataset graph G such that $g \subseteq G$, since $g' \subseteq g$ we would conclude that $g' \subseteq G$, which necessarily implies that $G \in Answer(g')$, which contradicts the fact that $Answer(g') = \emptyset$. Thus, no such graph G can exist and it is safe to stop query processing at this stage.

3.3 iGQ Frameworks for Supergraph Queries

Turing our attention to the iGQ operations for supergraph queries, i.e., when $\mathbb{M}$ is a method for supergraph query processing (denoted $\mathbb{M}_{super}$). Similarly, method $\mathbb{M}_{super}$ builds its graph dataset index. The iGQ index, $\mathbb{I}$, starts off empty; it is then populated as queries arrive and are executed by $\mathbb{M}_{super}$.

Upon the arrival of a query g, three threads are set off: (i) a thread employing method $\mathbb{M}_{super}$'s algorithms to result a candidate set of graphs, $CS(g)$; (ii) another two threads checking whether the query graph is a subgraph/supergraph of previous query graphs.

The following shall proceed with the assumptions (3.1) and (3.2) stated in §3.2 and use the example of Figure 3.4, so as to illustrate the operations of each component in iGQ framework and how they are bundled together.

3.3.1 The Subgraph Case: $\mathbb{I}_{sub}$

Such case occurs when a new supergraph query g is a subgraph of a previous query g'. As shown in Figure 3.11, the supergraph query processing method $\mathbb{M}_{super}$ generates a candidate set $CS(g)$, which contains four graphs $\{G_1, G_2, G_3, G_4\}$. Running g through $\mathbb{I}_{sub}$ discovers a previous query graph g' such that $g \subseteq g'$. The answer set for g', $Answer(g') = \{G_1, G_{20}\}$ is also accompanied.

Here comes the reasoning process. First, consider graph $G_2 \in CS(g)$. We know from $\mathbb{I}_{sub}$ that $G_2 \notin Answer(g')$. If $g \supseteq G_2$ were to indeed be true, since $g \subseteq g'$, then it must follow that $g' \supseteq G_2$; that is, $Answer(g')$ would have to contain G_2 as well, which is a contradiction. Therefore, it is safe to conclude that $g \not\supseteq G_2$ and thus G_2 can be safely removed from $CS(g)$. Ditto for the removal of graphs G_3, G_4 from $CS(g)$, which reduces in this case the number of required subgraph isomorphism tests by 75%.

Next, consider $G_1 \in CS(g)$. Since $Answer(g') = \{G_1, G_{20}\}$, $g' \supseteq G_1$ is at hand. And $\mathbb{I}_{sub}$ has discovered that $g \subseteq g'$. However, as to the containment relationship between G_1 and g, it still bears uncertainties. Providing the queries g and g' in Figure 3.4, Figure 3.12 shows the possible instances of G_1 such that $g \supseteq G_{1y}$ and $g \not\supseteq G_{1n}$. Due to such uncertain status, G_1 will have to be left for subgraph isomorphism verification.

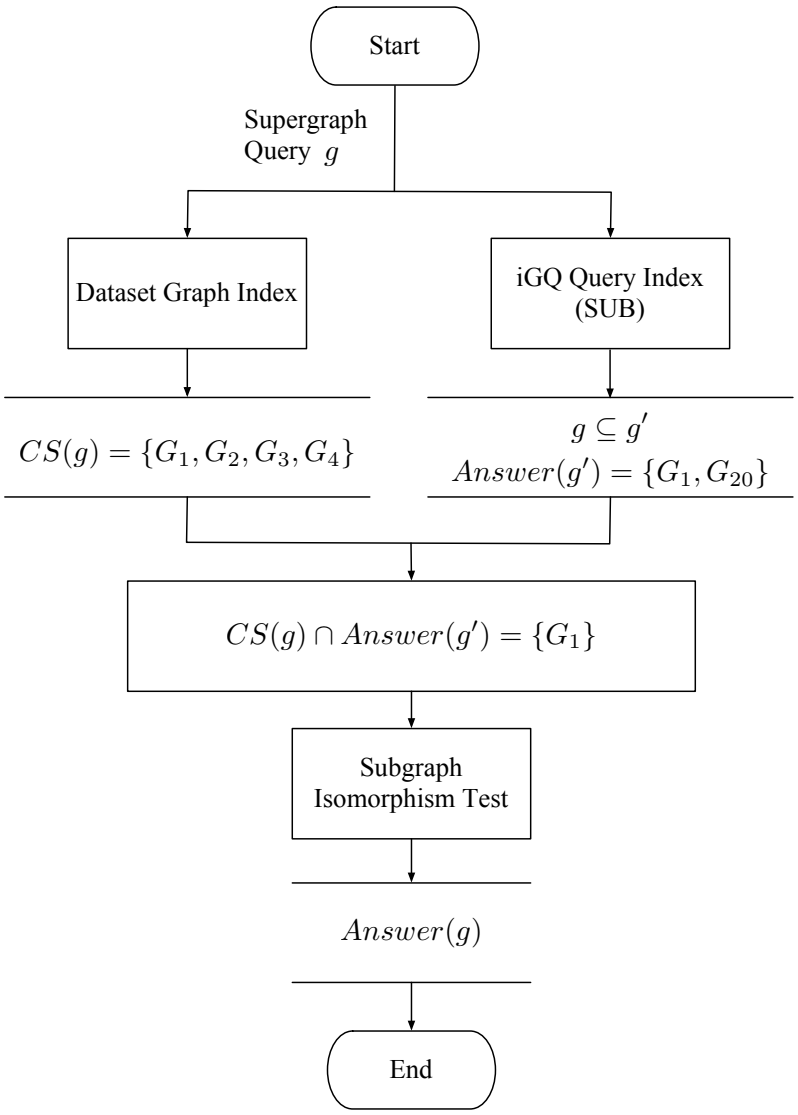


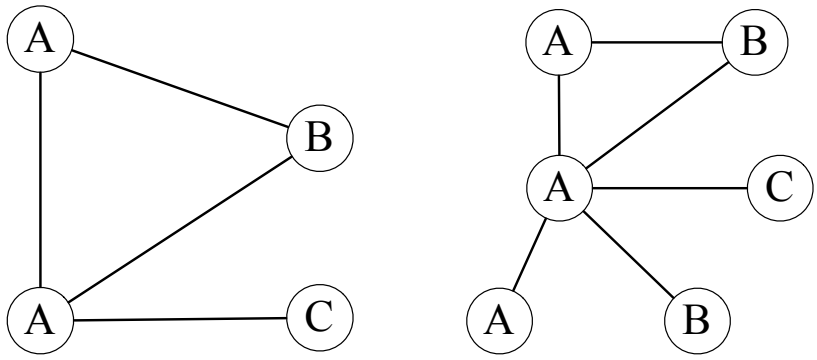
Figure 3.11: iGQ Subgraph Case for Supergraph Query Processing (when g is a Q_{super})

In general, g may be a subgraph of multiple previous query graphs g'_i in $\mathbb{I}_{sub}$. As illustrated, only those graphs appearing in the answer sets of all such queries g'_i may actually be subgraphs of g; hence dataset graphs submitted by iGQ for subgraph isomorphism testing is given by:

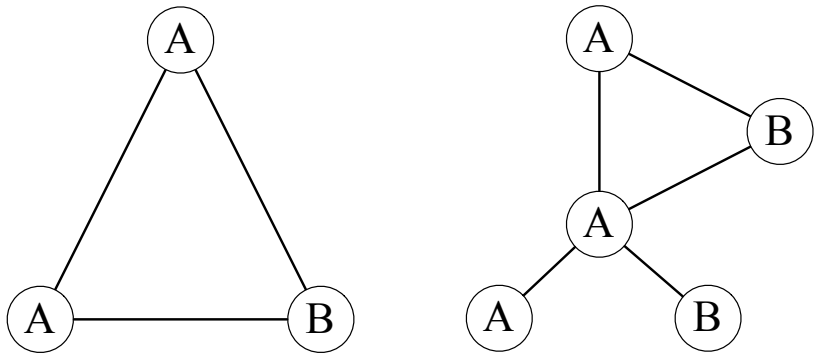
$$CS_{sub}(g) = CS(g) \cap \bigcap_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i) \tag{3.6}$$

where $g'_i \in \mathbb{I}_{sub}(g)$, with answer set $Answer(g'_i)$ (when g'_i was executed by $\mathbb{M}_{super}$). It is intersection on multiple sets $Answer(g'_i)$, as only those stored graphs being contained in all $g'_i \in \mathbb{I}_{sub}(g)$ could possibly be subgraphs of g. The final answer produced for supergraph query g by iGQ, $Answer(g)$, will be the subset of graphs in $CS_{sub}(g)$ that survive the subgraph isomorphism test.

Lemma 5. *The iGQ answer in the subgraph case does not contain false positives.*



(a) New Query g (b) Previous Query g'



(c) G_{1y} (d) G_{1n}

Figure 3.12: When g is a Q_{super}, the Uncertain Status of Dataset Graph G_1: $g \supseteq G_{1y}$ and $g \not\supseteq G_{1n}$

Proof. This trivially follows by construction as all graphs in $Answer(g)$ have passed through subgraph isomorphism testing at the final stage of processing. □

Lemma 6. *The iGQ answer in the subgraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by $\mathbb{I}_{sub}$; i.e., for some supergraph query g, $\exists G_{FN}$ such that $g \supseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. Method $\mathbb{M}_{super}$ does not produce in its candidate set any false negatives, hence $G_{FN} \in CS(g)$. Then, the only possibility for error is for iGQ to have removed graph G_{FN} from $CS_{sub}(g)$ with formula (3.6). This implies that $\exists g'$ such that $g' \in \mathbb{I}_{sub}(g)$ and $G_{FN} \notin Answer(g')$. But since $g' \in \mathbb{I}_{sub}(g)$, by equation (3.1), $g \subseteq g'$, and then $g \supseteq G_{FN}$ $\Rightarrow g' \supseteq G_{FN} \Rightarrow G_{FN} \in Answer(g')$ (a contradiction). □

Theorem 3. *The iGQ answer in the subgraph case of query processing is correct.*

Proof. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 5 and 6. □

Interpreting the Subgraph Case: Where Does Benefit Come From?

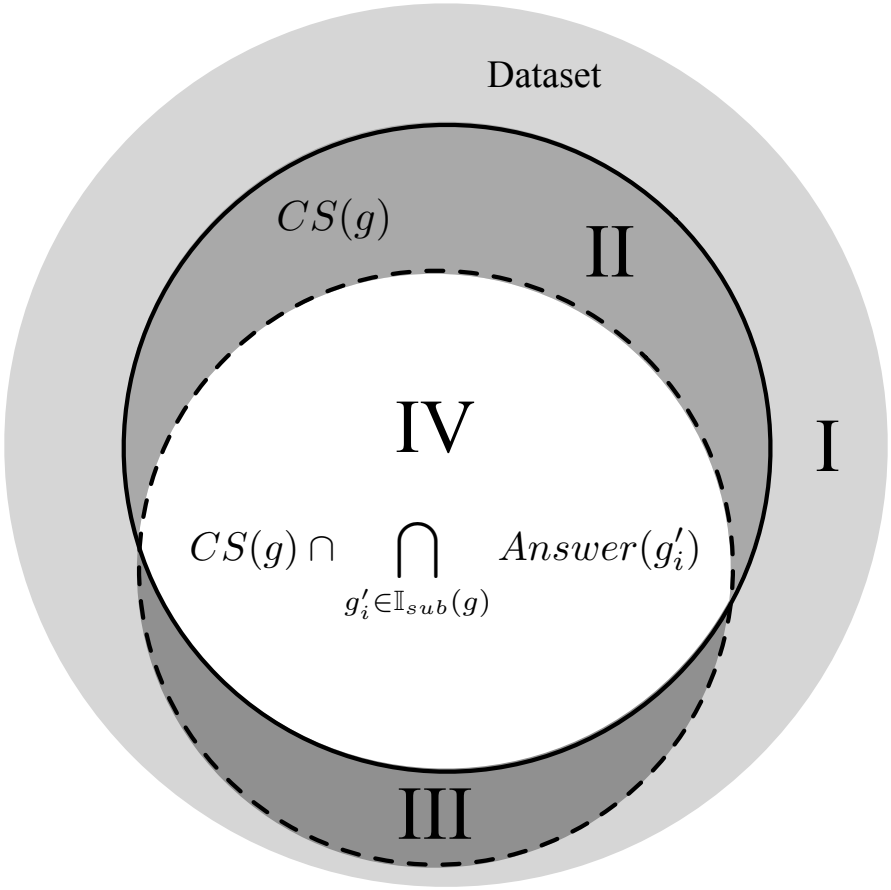


Figure 3.13: Benefit Analysis: iGQ Subgraph Case when g is a Q_{super} (Areas Satisfy II $\subseteq$ I, III $\subseteq$ I, IV $\subseteq$ II and IV $\subseteq$ III; I, II and III are Round; Each Notation is inside its Area; II Has a Solid Border and III is Enclosed by Dashes.)

Turning attention to the iGQ for supergraph query processing. Again, the benefit of iGQ is analyzed in the subgraph case and the supergraph case individually.

Figure 3.13 shows the benefit for subgraph case when g is a Q_{super}. Inside the area I that covers all dataset graphs, there are two round subsets, namely II and III. The former denotes the candidate set $CS(g)$; the latter is for dataset graphs in $\bigcap_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i)$. Their intersection results the area IV, i.e., $CS(g) \cap \bigcap_{g'_i \in \mathbb{I}_{sub}(g)} Answer(g'_i)$ in formula (3.6). IV includes graphs in the new candidate set, upon which subgraph isomorphism tests shall be performed to determine the final query answer of g. All in all, for supergraph query processing, iGQ subgraph case subtracts candidates that do not enter area IV – the smaller IV is, the larger benefit achieves.

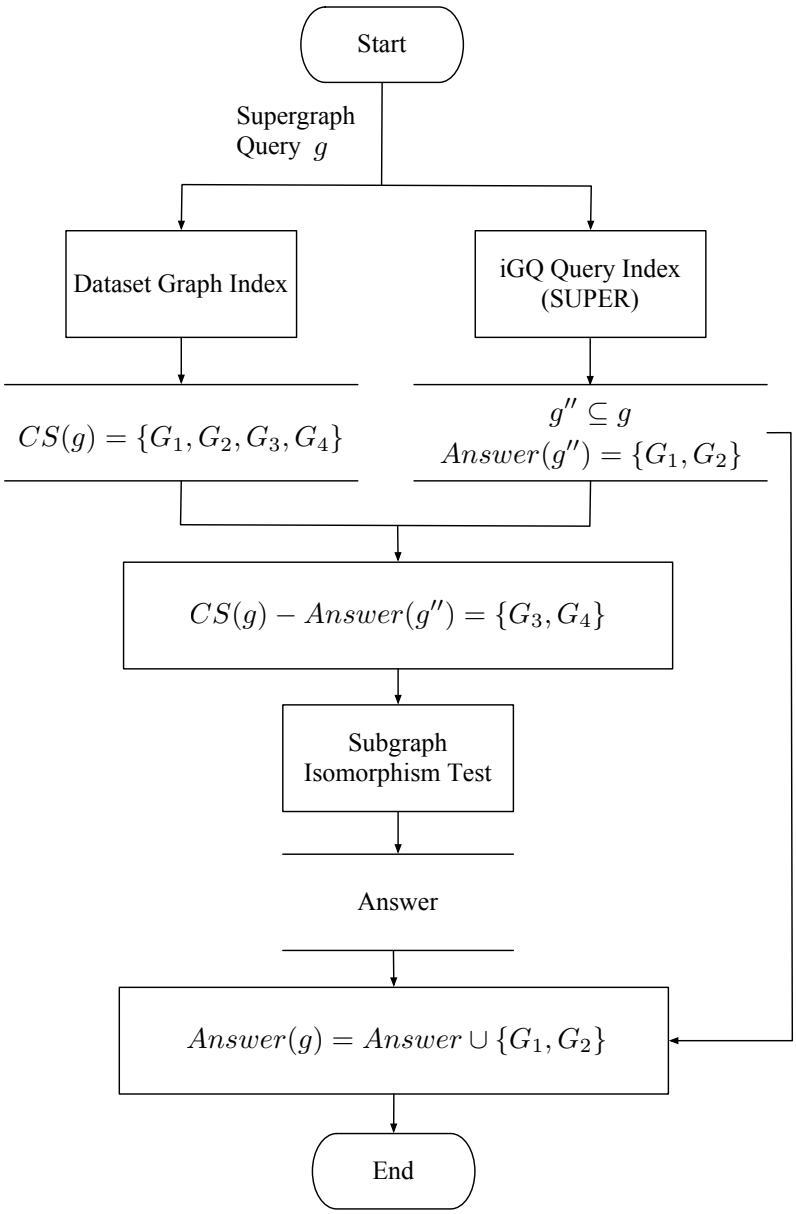


Figure 3.14: iGQ Supergraph Case for Supergraph Query Processing (when g is a Q_{super})

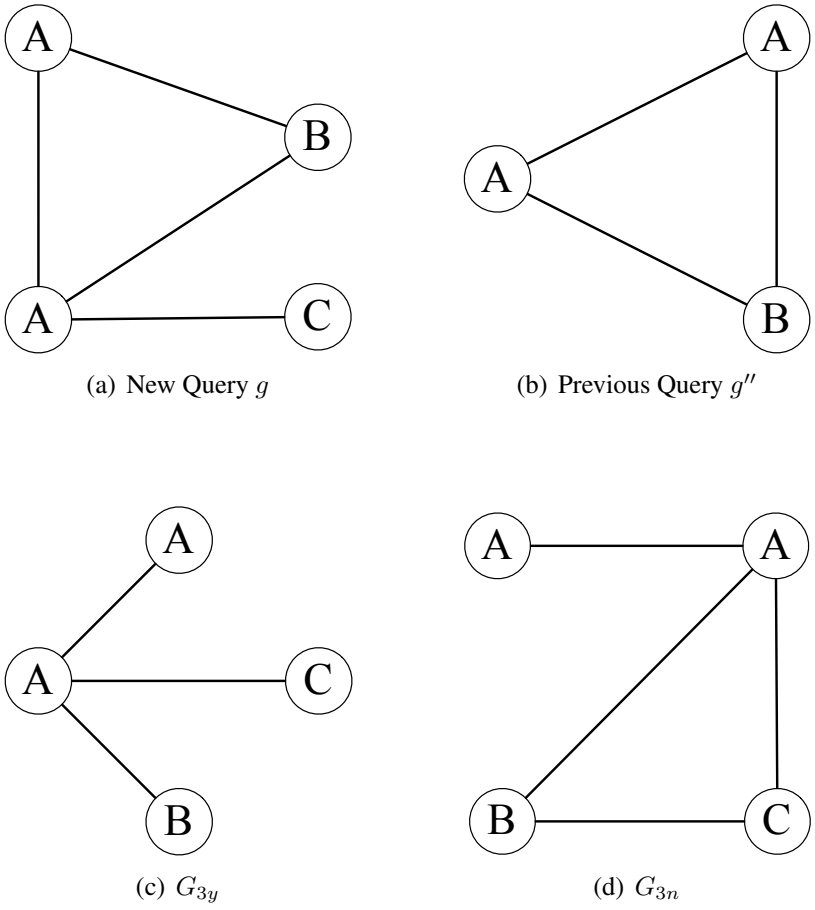
3.3.2 The Supergraph Case: $\mathbb{I}_{super}$

This case occurs when a new query g is a supergraph of a previous supergraph query g''. Figure 3.14 depicts an example for the supergraph case of iGQ in processing a supergraph query. For the new query g, method $\mathbb{M}_{super}$'s graph index produces a candidate set, $CS(g)$, which contains four graphs $\{G_1, G_2, G_3, G_4\}$. Meanwhile, $\mathbb{I}_{super}$ discovers that there exists a previous query g'', such that $g'' \subseteq g$. The answer set of g'' is then retrieved; in this case, $Answer(g'') = \{G_1,\ G_2\}$.

The reasoning proceeds as follows. Consider graph $G_1 \in CS(g)$. Since from $\mathbb{I}_{super}$ it has been concluded that $g'' \subseteq g$ and from the answer set of g'' we know that $g'' \supseteq G_1$,

it necessarily follows that $g \supseteq G_1$. Similarly, $g \supseteq G_2$ can be concluded. Hence, there is no point in testing G_1 or G_2 for subgraph isomorphism against g, as the answer is already known. Therefore, one can safely subtract graphs G_1, G_2 from $\mathbb{M}_{super}$'s candidate set, and test only the remaining graphs (reducing the number of subgraph isomorphism tests in this example by 50%). And G_1, G_2 are added to the answer set in the end.

Now consider the dataset graph G_3. Since $Answer(g'') = \{G_1, G_2\}$, it naturally follows $g'' \not\supseteq G_3$. Though $\mathbb{I}_{sup}$ has concluded that $g \supseteq g''$, whether G_3 is contained in g is uncertain. Again, queries g and g'' in Figure 3.4 are used for illustration. As shown by Figure 3.15, there could exist different instances of G_3 such that G_{3y} is contained in g whereas G_{3n} is not. Therefore, G_3 will go for subgraph isomorphism testing to determine the final status. Similarly, G_4 will be tested for subgraph isomorphism as well.



(a) New Query g (b) Previous Query g''

(c) G_{3y} (d) G_{3n}

Figure 3.15: When g is a Q_{super}, the Uncertain Status of Dataset Graph G_3: $g \supseteq G_{3y}$ and $g \not\supseteq G_{3n}$

In the general case, g may be a supergraph of several previous query graphs g_i'' in $\mathbb{I}_{super}$. Following the above reasoning, one can safely remove from $CS(g)$ all graphs appearing in the answer sets of all query graphs g_i'', as they are bound to be subgraphs of g. Hence, the set

of graphs submitted by iGQ for subgraph isomorphism verification is given by:

$$CS_{super}(g) = CS(g) \setminus \bigcup_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'') \tag{3.7}$$

where each graph g_i'' satisfies $g_i'' \in \mathbb{I}_{super}(g)$, with answer set $Answer(g_i'')$ (when g_i'' was executed by $\mathbb{M}_{super}$). The set union operation on $Answer(g_i'')$ rests on the fact that for any $g_i'' \in \mathbb{I}_{super}(g)$ with $G \in Answer(g_i'')$, it applies with $g \supseteq G$.

Finally, if $Answer_{super}(g)$ is the subset of graphs in $CS_{super}(g)$ verified to be contained in g through subgraph isomorphism testing, the final answer set for query g will be:

$$Answer(g) = Answer_{super}(g) \cup \bigcup_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'') \tag{3.8}$$

Lemma 7. *The iGQ answer in the supergraph case does not contain false positives.*

Proof. Assume that a false positive was produced by iGQ; particularly, consider the first ever false positive produced by $\mathbb{I}_{super}$, i.e., for some query g, $\exists G_{FP}$ such that $g \not\supseteq G_{FP}$ and $G_{FP} \in Answer(g)$. Note that G_{FP} cannot be in $Answer_{super}(g)$, as the latter contains only those graphs from $CS_{super}(g)$ that have been verified to be subgraphs of g after passing the subgraph isomorphism test, and hence $g \not\supseteq G_{FP} \Rightarrow G_{FP} \notin Answer_{super}(g)$. Therefore, by formula (3.8), $G_{FP} \in Answer(g) \Rightarrow \exists g''$ such that $g'' \in \mathbb{I}_{super}(g)$ and $G_{FP} \in Answer(g'')$. But according to equation (3.2), $g'' \in \mathbb{I}_{super}(g) \Rightarrow g \supseteq g''$; hence $G_{FP} \in Answer(g'') \Rightarrow g'' \supseteq G_{FP}$. It thus follows that $g \supseteq G_{FP}$ (a contradiction). □

Lemma 8. *iGQ in the supergraph case does not introduce false negatives.*

Proof. Assume that a false negative was produced by iGQ; particularly, consider the first ever false negative produced by $\mathbb{I}_{super}$, i.e., for some query g, $\exists G_{FN}$ such that $g \supseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. As method $\mathbb{M}_{super}$ is assumed to be correct, it cannot produce any false negatives when executing query g, hence $g \supseteq G_{FN} \Rightarrow G_{FN} \in CS(g)$. Then, the only possibility for error is that G_{FN} was removed using formula (3.7); i.e., $G_{FN} \notin CS_{super}(g)$. That implies that $\exists g''$ such that $g'' \in \mathbb{I}_{super}(g)$ and $G_{FN} \in Answer(g'')$. However, by equation (3.8), G_{FN} will be added to $Answer_{super}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). □

Theorem 4. *The iGQ answer in the subgraph case of query processing is correct.*

Proof. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 7 and 8. □

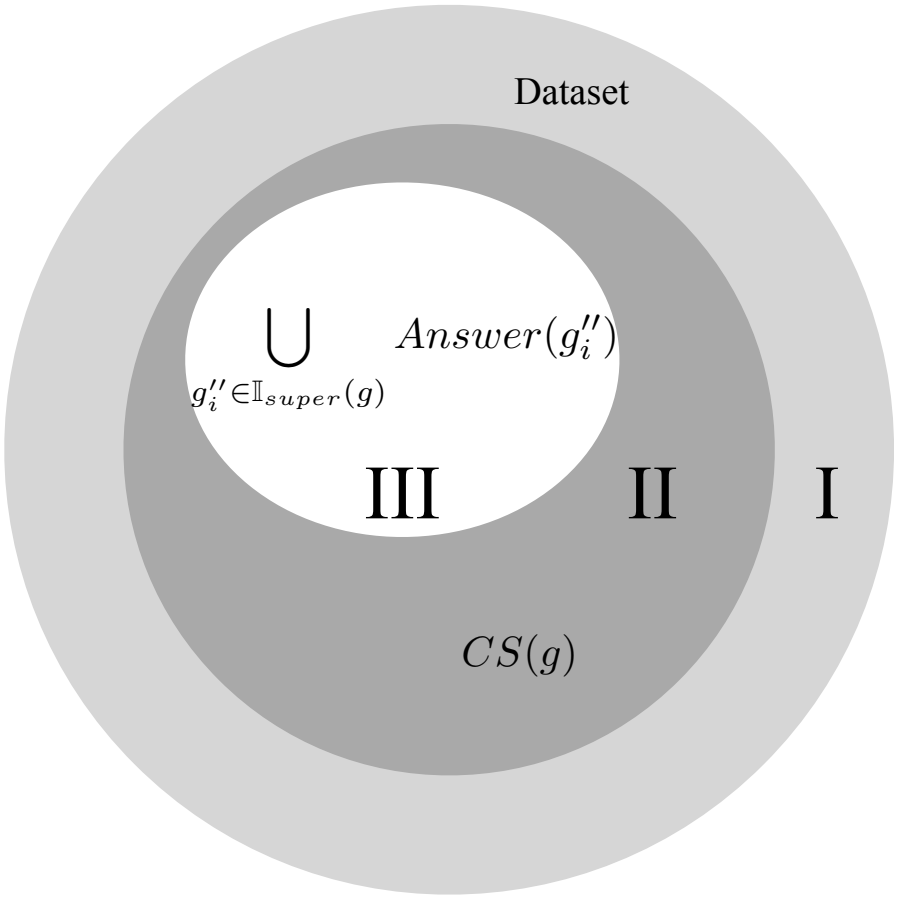


Figure 3.16: Benefit Analysis: iGQ Supergraph Case when g is a Q_{super} (Areas Satisfy III $\subseteq$ II $\subseteq$ I; Each Notation is inside its Area.)

Interpreting the Supergraph Case: Where Does Benefit Come From?

As to the supergraph case when g is a Q_{super}, Figure 3.16 shows the source of iGQ benefit. Inside the area I (the whole dataset), the candidate set $CS(g)$ takes the area II, of which a subset III covers the the minus part $\bigcup_{g_i'' \in \mathbb{I}_{super}(g)} Answer(g_i'')$ in equation (3.7). In the end, area III is added to the final answer of query g by formula (3.8). As a result, for supergraph query processing, iGQ supergraph case benefits by removing graphs in III from subgraph isomorphism testing – the larger III is, the higher profit obtains.

3.3.3 Two Optimal Cases

Similarly, iGQ renders two special cases for supergraph query processing. First, iGQ can easily discover the case where a new supergraph query, g, is exactly the same as a previous query contained in $\mathbb{I}$. Specifically, this holds when $\exists g' \in \mathbb{I}$ such that $g \subseteq g'$ or $g \supseteq g'$, and g and g' have the same number of nodes and edges. In such case, the stored answer set for g' can be returned directly, completely avoiding the subgraph isomorphism tests as the actual

result for g is known already! As the subgraph isomorphism test dominates the query time, a large performance improvement is expected.

Second, consider the subgraph case. If $\exists g' \in \mathbb{I}_{sub}(g)$ such that $g \subseteq g'$ and $Answer(g') = \emptyset$, the verification stage can be omitted again. The reasoning lies in that if there were a dataset graph g such that $g \supseteq G$, since $g \subseteq g'$ one can conclude that $g' \supseteq G$, which in turn follows $G \in Answer(g')$, being a contradiction with the fact that $Answer(g') = \emptyset$. Hence, no such graph G can exist and it is safe to return an empty answer set directly.

3.4 iGQ Algorithms and Structures

The proofs of correctness presented in the previous subsections assume that $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$ provide correct results; recall formulas 3.1 and 3.2. Now we shall discuss the associated mechanisms and prove that they hold.

3.4.1 Finding Supergraphs in $\mathbb{I}_{sub}$

This case represents a microcosm of our original subgraph query problem, where instead of indexing and querying dataset graphs, iGQ index and query previous query graphs. Hence, any approach from the related works can be adapted for this purpose. Actually, as iGQ can complement any existing approach, $\mathbb{M}_{sub}$, one can utilize $\mathbb{M}_{sub}$'s method for subgraph query processing for the subgraph case of iGQ, or any other method appropriate for iGQ's characteristics.

Note that the assumed correct method $\mathbb{M}_{sub}$ precludes false negatives and subgraph isomorphism testing of all candidates precludes false positives. Hence, formula (3.1)'s assumption is trivially satisfied.

3.4.2 Finding Subgraphs in $\mathbb{I}_{super}$

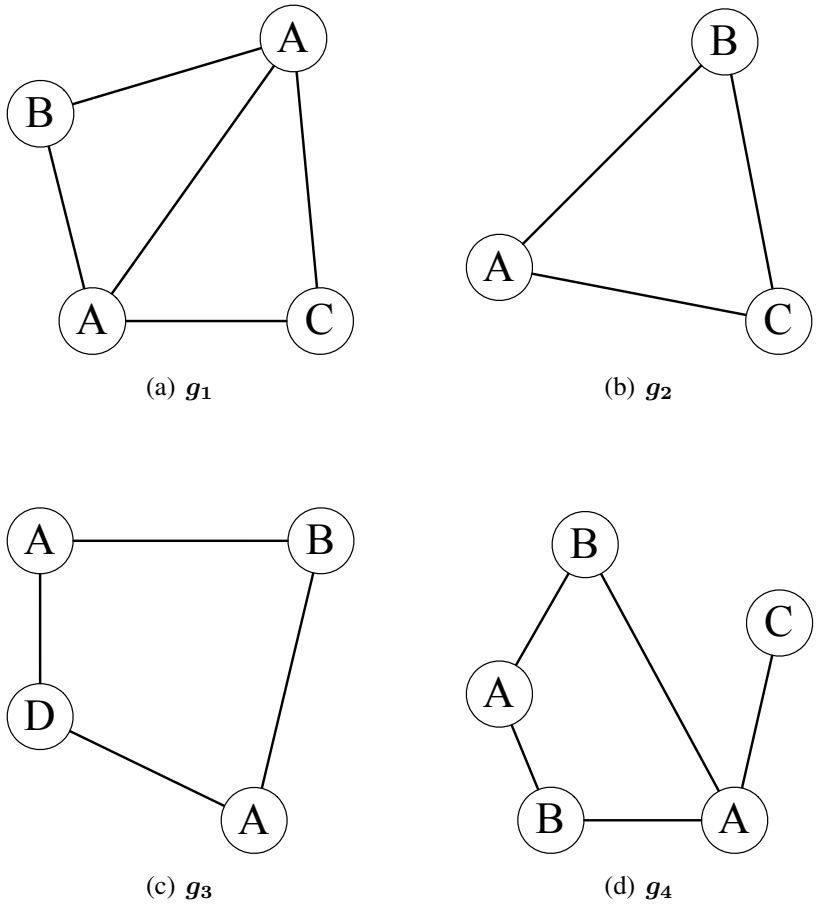
The problem of supergraph query processing has also received some attention (e.g., in [43, 51, 45, 46, 44]). In principle, any of these algorithms can be utilized for the task at hand within iGQ. However, this work chooses to propose a new approach, which is efficient yet simple and avoids the complexities and overheads involved in the above general approaches. An ideal such method should meet the demands that it can easily fit within the framework of iGQ and share the query index with $\mathbb{I}_{sub}$.

A Running Example

Inside the component $\mathbb{I}_{super}$ runs the supergraph query processing. Given a newly coming query g, finding subgraphs in $\mathbb{I}_{super}$ returns previous queries $\{g_i\}$ such that $g \supseteq g_i$. Following the FTV paradigm, the query processing is divided into two stages.

- First, by specific filtering rules, some of the previous queries are pruned out and the remaining consists of a candidate set $CS(g)$.
- Then, each graph in $CS(g)$ is tested by subgraph isomorphism to determine whether it is a subgraph of query g.

This work contributes the first step, by presenting a set of rules as to deriving the candidate set $CS(g)$. Next, we shall use an example to demonstrate such filtering process inside $\mathbb{I}_{super}$.



(a) g_1 (b) g_2

(c) g_3 (d) g_4

Figure 3.17: An Example with Four Previous Queries in iGQ: $\{g_1, g_2, g_3, g_4\}$.

To start with, previous queries stored in iGQ are preprocessed for their features. According to Figure 3.17, there are four such queries $\{g_1, g_2, g_3, g_4\}$. Features of these graphs are first extracted (see Table 3.1). For simplicity, features with no more than two vertices are

Table 3.1: Extracting Features from Queries $\{g_1, g_2, g_3, g_4\}$

query	feature	occurrence
g_1	A	2
g_1	B	1
g_1	C	1
g_1	AA	1
g_1	AB	2
g_1	AC	2
g_2	A	1
g_2	B	1
g_2	C	1
g_2	AB	1
g_2	AC	1
g_2	BC	1
g_3	A	2
g_3	B	1
g_3	D	1
g_3	AB	2
g_3	AD	2
g_4	A	2
g_4	B	2
g_4	C	1
g_4	AB	4
g_4	AC	1

considered – a vertex or a path, where each vertex is represented by its label, e.g., AB is a path bridged by two vertices with label A and B respectively. Features with occurrences in each graph are then inserted into an indexing structure of trie, where the route starting from root to each leaf constitutes the canonical form of a feature, as shown in Figure 3.19.

When the new query enters, it is also decomposed into a set of features, which are then compared with those of indexed graphs to determine the candidate set. Specifically, Figure 3.18 shows a query g, with its features $F(g)$ detailed in Table 3.2 where each feature $f \in F(g)$ is accompanied with its occurrence in query g (namely o).

Every feature f then looks up the trie in Figure 3.19, retrieving graphs $\{g_i\}$ with occurrences no more than o and inserting $\{g_i\}$ into a $multiset$ $\mathbb{G}$. Table 3.3 details the retrieved $\{g_i\}$ per feature $f \in F(g)$. Please note that g_4 is not included in the retrieval result of (AB,2) – the feature AB appears four times in g_4 (see the trie in Figure 3.19), violating the requirement that occurrence in g_i should not exceed that of query g (i.e., 2). Hence, the returned $multiset$ is $\mathbb{G} = \{g_1, g_1, g_1, g_1, g_1, g_1, g_2, g_2, g_2, g_2, g_2, g_3, g_3, g_3, g_3, g_4, g_4, g_4\}$.

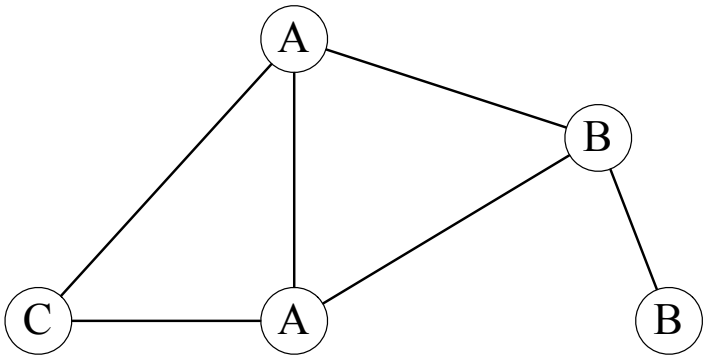


Figure 3.18: New Query g Entering the System

Table 3.2: Decomposing the New Query Graph g into Features

query	**feature**	**occurrence**
g	A	2
g	B	2
g	C	1
g	AA	1
g	AB	2
g	AC	2
g	BB	1

Table 3.3: Comparing Each Query Feature against the Trie

(f, o)	$\{g_i\}$
(A, 2)	$\{g_1, g_2, g_3, g_4\}$
(B, 2)	$\{g_1, g_2, g_3, g_4\}$
(C, 1)	$\{g_1, g_2, g_4\}$
(AA, 1)	$\{g_1\}$
(AB, 2)	$\{g_1, g_2, g_3\}$
(AC, 2)	$\{g_1, g_2, g_4\}$
(BB, 1)	$\emptyset$

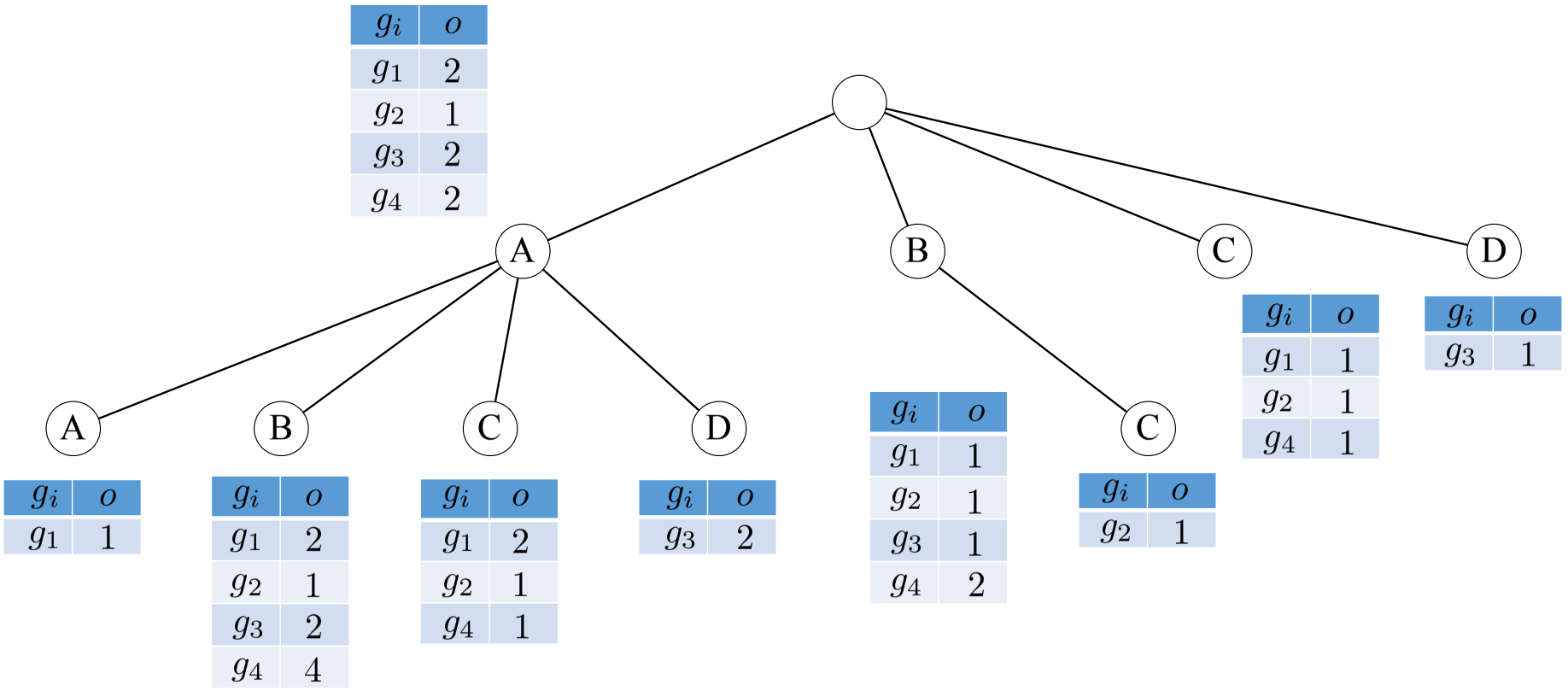


Figure 3.19: Indexing the Features of Previous Queries by a Trie

Table 3.4: Counting the Occurrences of Graphs in $\boldsymbol{multiset}$ $\mathbb{G}$

graph	count
g_1	6
g_2	5
g_3	3
g_4	4

Next, the derived $\boldsymbol{multiset}$ is analyzed, evicting the candidate graphs. First, each graph $g_i \in \mathbb{G}$ results a count of its occurrences in this $\boldsymbol{multiset}$, as shown in Table 3.4. Then, the count of graph g_i is compared with the number of distinct features in g_i (see the number of rows per graph in Table 3.1; namely $\boldsymbol{NF}[g_i]$). If $\boldsymbol{count}(g_i) == \boldsymbol{NF}[g_i]$ holds, g_i is added to the candidate set $\boldsymbol{CS}(g)$.

For example, the $\boldsymbol{multiset}$ $\mathbb{G}$ accommodates six occurrences of g_1. Providing the statistics in Table 3.1, g_1 consists of six unique features. Satisfying the requirement $\boldsymbol{count}(g_i) == \boldsymbol{NF}[g_i]$, g_1 enters the candidate set. Turning our attention to graph g_2. It appears five times in $\mathbb{G}$ but has six distinct features in total. Thus, g_2 is filtered out, as it possesses an extra feature that query g does not have. Similarly, g_3 and g_4 are precluded by the filtering process.

In overall, when generating the candidate set $\boldsymbol{CS}(g)$ among previous queries g_i, our approach is characterized by the two following rules.

- Previous query g_i should not possess extra features that do not appear in the query g.
- For each feature shared by g_i and g, occurrence of the former should be no more than the latter.

Violating any of the two requirements would preclude g_i from the candidate set $\boldsymbol{CS}(g)$. Each graph in $\boldsymbol{CS}(g)$ is then verified for subgraph isomorphism, returning subgraphs of the query g.

Algorithms with Proof of Correctness

Algorithm 1 shows how $\mathbb{I}_{\boldsymbol{super}}$ is created. Briefly, $\mathbb{I}_{\boldsymbol{super}}$ is a trie, storing features of queries. For each feature f it stores a pair $\{g_i, o\}$ for each graph g_i in which f appears, where o is its number of occurrences in g_i. For each g_i it also stores the number of distinct features ($\boldsymbol{NF}[g_i]$) it contains.

Algorithm 2 illustrates how $\mathbb{I}_{\boldsymbol{super}}$ identifies candidates $\boldsymbol{CS}$ that are potential subgraphs of query g. The idea is to find those graphs that contain $\boldsymbol{only}$ features included in the query

Algorithm 1 The Supergraph Index in iGQ

```
1: Input: Set ℚ of (previous) queries g_1, g_2, ..., g_n
2: Output: Supergraph index of previous queries 𝕀_super
3:
4: Initialize 𝕀_super to an empty TRIE
5: for all g_i ∈ ℚ do
6:     Extract all features of g_i and insert them in set F(g_i)
7:     NF[g_i] = |F(g_i)|
8:     for all features f ∈ F(g_i) do
9:         o = number of occurrences of f in g_i
10:        𝕀_super.insert(f, {g_i, o})
11:    end for
12: end for
13: return 𝕀_super
```

graph g (lines 19–22; the check for $count(g_i)$ on line 20, ensures that all individual features of g_i are contained in g), and where for each such graph g_i a feature f occurs at most as many times as f occurs in g (line 12). Last, the graphs in CS are tested for subgraph isomorphism to verify that $g_i \subseteq g$.

Lemma 9. *Finding subgraphs in $\mathbb{I}_{super}$ does not contain false positives.*

Proof. This trivially follows by construction as all found subgraphs have passed through subgraph isomorphism testing at the final stage of processing. □

Lemma 10. *Finding subgraphs in $\mathbb{I}_{super}$ does not introduce false negatives.*

Proof. Assume there were a false negative g_i such that $g_i \subseteq g$ and $g_i \notin CS$. Since $g_i \subseteq g$, any feature f in g_i appears no more times than f appears in g, thus g_i would be added to $\mathbb{G}$ on every execution of line 12. As $g_i \subseteq g$, all of g_i's features must appear in g. Then g_i would pass the if-clause at line 20 and be added to CS (contradiction). □

Theorem 5. *Finding subgraphs in $\mathbb{I}_{super}$ is correct.*

Proof. There are only two possibilities for error; finding subgraphs in $\mathbb{I}_{super}$ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 9 and 10. □

Hence, formula (3.2)'s assumption holds.

Algorithm 2 Supergraph Query Processing in iGQ

```
1: Input: Query graph g and 𝕀_super
2: Output: Candidate set CS of potential subgraphs of g
3:
4: Initialize multiset 𝔾 = ∅
5: Extract all features of query graph g, F(g)
6: for all features f ∈ F(g) do
7:     O[f, g] = number of occurrences of f in g
8: end for
9: for all features f ∈ F(g) do
10:    if f ∈ 𝕀_super then
11:        for all {g_i, o} ∈ 𝕀_super.get(f) do
12:            if o ≤ O[f, g] then
13:                𝔾.insert(g_i)
14:            end if
15:        end for
16:    end if
17: end for
18: for all graphs g_i ∈ 𝔾 do
19:    count(g_i) = number of occurrences of g_i in 𝔾
20:    if count(g_i) == NF[g_i] then
21:        CS.add(g_i)
22:    end if
23: end for
24: return CS
```

3.5 Summary

This chapter has contributed a novel perspective and solution of expediting graph queries, which departs from related work in three ways: First, it constructs the indices for queries, as opposed to simply relying on the index of dataset graphs, which especially facilitate cases when the query shares little characteristics of graph dataset; Second, it maintains the knowledge the system produced when executing previous queries to accelerate graph query processing; Third, it can be used to expedite both subgraph and supergraph queries, substantially demonstrating iGQ's elegance of killing two birds using one stone.

Contents of this chapter are mainly pertaining to the three aspects as follows.

- Demonstrating a new perspective to the problem of subgraph/supergraph query processing, with insights as to how the work performed by the system when executing queries can be appropriately managed to improve the performance of future queries.
- Presenting the iGQ framework as to how iGQ complements the existing approaches with two components that are responsible of discovering the subgraph/supergraph sta-

tus between queries and manages to reduce the number of subgraph isomorphism tests to be performed.

- Showing the details of iGQ approach such as the query index structure and accompanying algorithms.

Furthermore, iGQ offers food for thought.

- It "mines" the relationships among queries. By indexing query graphs, in addition to queries that are frequently submitted to system, it also benefits queries that share subgraph/supergraph relationships with those executed. This is significant as it leads to further interesting research topics for graph query processing.
- It presents a simple yet efficient solution to deal with supergraph query processing. By following a set of straightforward logics, this approach swiftly avoids heavy overhead and other unnecessary sophisticated issues. Its elegance is further emphasized by the formally proved correctness.

Chapter 4

GraphCache: A Caching System for Graph Queries

Underpinned by the proposed iGQ framework, this chapter shall put forth GraphCache, a full-fledged caching system for graph queries. To obtain the inspiration of designing GraphCache, this work starts off with a quick overview of the well-established cache principles and applications. By considering the specific characteristics of graph queries, the design issues of GraphCache are figured out, each with the corresponding solution. It then follows the system architecture of GraphCache, featured by well defined subsystems and interfaces, allowing for the flexible plug-in of any general subgraph/supergraph query method in the literature, be it an FTV algorithm or SI solution.

Essentially, GraphCache is a semantic graph cache that is capable of harnessing both subgraph and supergraph cache hits, extending the traditional exact-match-only hit and hence leading to significant speedups for graph queries. GraphCache incorporates iGQ framework, where previously executed graph queries benefit future query processing. As queries arrive continuously and the memory space is finite, GraphCache requires mechanisms to efficiently deal with cache replacement.

To address this problem, an instant idea is to employ existing cache replacement policies directly. However, none of the policies proposed so far had taken the particularity of graph queries into consideration. GraphCache hence contributes a number of replacement strategies that are graph query aware, emphasizing a novel hybrid dynamic policy for its competitive performances. To the best of our knowledge, GraphCache is the first caching system in the literature for general subgraph/supergraph queries.

4.1 System Design and Architecture

4.1.1 Overview of Cache Issues

Caching is a prevalent technique in designing computer systems. Typically, cache functions by exploiting locality, which could have two cases, temporal locality and spatial locality. In a program having good temporal locality, a memory location that is accessed once is likely to be frequently accessed in the near future. And in a program with good spatial locality, if a memory location has been accessed, then in the near future, its nearby location is likely to be accessed. Fundamentally, the two cases share the same principle, i.e., using the past to predict and facilitate the future – essence of caching.

In a modern computer system, caching is well utilized. At the bottom hardware layer, small fast memories (namely cache memories) hold the most recently referenced blocks of data and instruction items, allowing the high speed access of main memory. The intermediate operating system layer allows the main memory to cache the most recently used blocks in the disk file. On the top layer of application programs, caching is also widely employed. For example, Web browsers cache recently visited documents on a local disk and high-volume Web servers hold the recently requested documents in a front-end disk cache.

Cache contents differ in various systems. Microsoft SQL Sever caches query plans and pages from database files. For example, the world-wide distinguished library in University of Glasgow boasts more than 25 million books. Say there is a query SELECT DISTINCT country FROM *book* GROUP BY country. Microsoft SQL Sever will scan the whole table of *book* with over 25 million records, on which the aggregation operations return a few entries only. When this query is resubmitted, Microsoft SQL sever will reuse the query plan and scan the table cached in memory from scratch. Obviously, it is not efficient enough, especially in the era of big data when queries come in an overwhelming rush.

For this reason, increasing numbers of application systems extend their architectures by plugging in a data cache, where query results are stored in a RAM named Memcached [88]. As a distributed memory object caching system, Memcached is highlighted by alleviating database load and accelerating dynamic web applications. Memcached is widely employed in mainstream big data applications such as LinkedIn [89] and Facebook [90]. Interestingly, Memcached also dedicates the elegant principle of **being simple yet powerful** – its simple design promotes quick deployment, ease of development, and swift solution towards many problems in large data caches.

All in all, these examples demonstrate good practices of designing cache systems and hence pave the way for this chapter, which targets at developing a graph cache system GraphCache to accelerate graph query performance. Following the fundamental principle of cache, a

quick idea at hand is to use previous graph queries to facilitate future query processing. Then, an interesting issue emerges: how to make good use of the previous graph queries? By the nature of locality, cache effects when the same query is submitted again. Also, recall the graph applications with subgraph/supergraph status of queries. Therefore, as to the designing of GraphCache, the inspiration obtained is – besides the traditional cache hits (caused by isomorphic queries), it is significant to consider and exploit the cache hits of subgraph/supergraph cases.

4.1.2 Designing GraphCache

In designing GraphCache, a set of design issues and goals are first identified, pertaining to the characteristics of (i) the query workloads, (ii) the underlying graph datasets, and (iii) the algorithmic and system context within which GraphCache will operate (e.g., categories of research methods GraphCache will complement). GraphCache is intended to expedite graph queries whatever the algorithm of choice may be and across a wide variety of query workloads and graph datasets.

Query Workloads As with any caching system, the assumption is that previous queries can help expedite future queries. This is reasonable, given the example applications mentioned in §1. Most works [16, 42, 13, 18, 40] test algorithms for queries directly generated from dataset graphs. Though this is of particular interest, workloads should also include queries that are not guaranteed to have any answer. Furthermore, in general, of particular interest to any caching system is the probability distribution of possible queries. For GraphCache this in effect refers to the popularity of query graphs or of regions of the dataset graphs. GraphCache should thus be able to deal effectively with various skewness levels of this distribution (e.g., from uniform to highly skewed Zipf distributions). Finally, a practical problem emerges when creating workloads: it must contain a large number of queries so as to obtain reliable results on the performance of any method. But subgraph isomorphism is NP-Complete. This leads to queries with possibly very long execution times, regardless of the heuristic used, making the experiments very time consuming. Nevertheless, the current work shall utilize well over 6 million queries for performance evaluation.

Graph Datasets Fortunately there exist a number of real-world graph datasets that are commonly used in related research. These of course help concretize the effects of any solution on real-world data and allow direct comparison of methods and result repeatability. For this reason, evaluations conducted over three popular such graph datasets will be reported; namely, AIDS[86], PDBS[87], and PCM[91]. However, it is worth creating additional synthetic datasets so to perform evaluations under characteristics unseen in the real-

world datasets. Specifically, this thesis shall use a synthetic dataset [14] presenting interesting characteristics regarding the number, size and node degrees of graphs in the dataset.

Algorithmic Context GraphCache is intended to be a general-purpose front-end for graph query processing. GraphCache entails a query indexing strategy that, as explained in §3, can accommodate both subgraph and supergraph queries. In addition, the design of GraphCache must be able to accommodate both FTV methods and SI algorithms; its current implementation comes bundled with well-established FTV methods and SI algorithms. In fact, any such algorithm is viewed as a pluggable component into the architecture, allowing any future algorithm to be incorporated.

4.1.3 System Architecture

GraphCache is designed from the ground up as a scalable semantic cache for subgraph/supergraph queries, capable of expediting any SI or FTV method (henceforth denoted *Method M*). Figure 4.1 shows the main architectural components of GraphCache, comprising three major subsystems: Method M, Query Processing Runtime, and Cache Manager. The last two are internal subsystems of GraphCache; the first is the method that GraphCache is called to expedite and hence external to GraphCache.

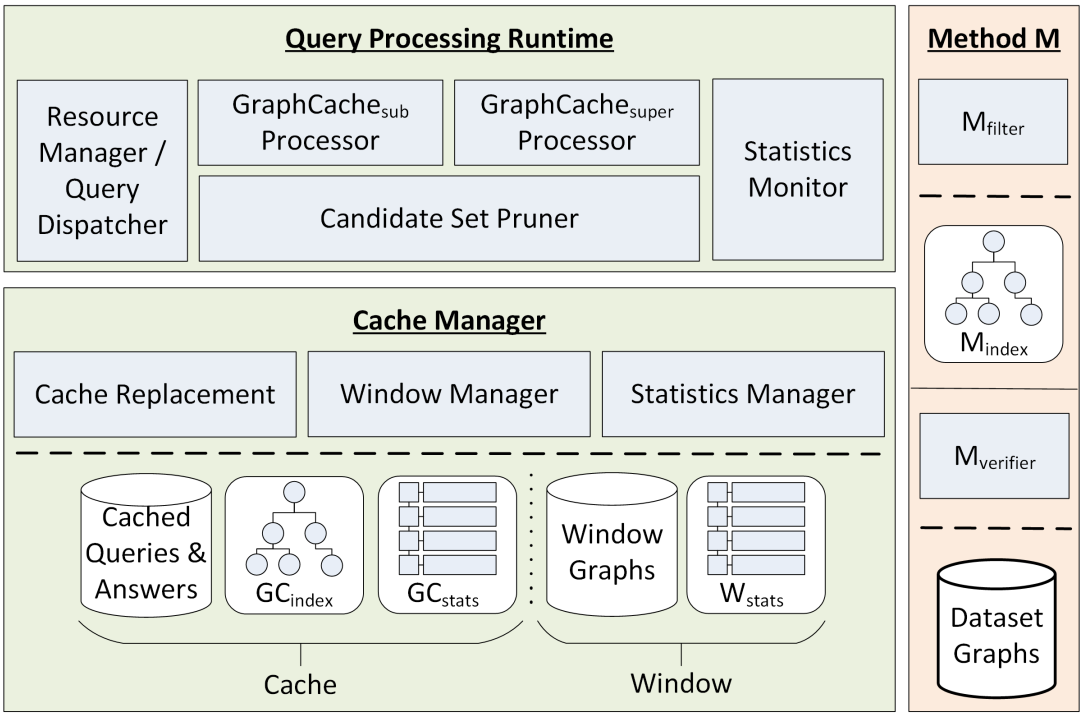


Figure 4.1: GraphCache System Architecture

Method M Subsystem

The Method M subsystem includes, at a minimum, the base graph dataset and a sub-iso test implementation, denoted $M_{verifier}$. Additionally, if M is a FTV method, then it also features its index, denoted M_{index}, and a filtering component, M_{filter}. The index is built in a pre-processing step, by using Method M's indexing component (not shown in Figure 4.1 for simplicity).

When GraphCache is not used, subgraph/supergraph query processing proceeds by first using M_{index} through M_{filter} to prune away dataset graphs definitely not containing (or contained in) the query, thus forming its candidate set, M_{CS}. Then $M_{verifier}$ executes a sub-iso test against all graphs in M_{CS}, reading their structure directly from the graph dataset store. For SI methods, M_{CS} contains all graphs in the dataset.

Query Processing Runtime Subsystem

Within GraphCache, the Query Processing Runtime is responsible for the execution of queries and the monitoring of key operational metrics. The key components inside the Query Processing Runtime subsystem consist of:

- A resource/thread manager that is in charge of dispatching queries to the various filtering/verification modules.
- The internal subgraph/supergraph query processors, which evict the set of previous queries that contain or are contained in the new query, respectively.
- The logic for GraphCache's candidate set pruning, discovering the how GraphCache leverages the knowledge of previous queries to expedite graph query processing.
- A statistics monitor taking the key measurements for the Query Processing Runtime subsystem.

These components communicate with Method M and the Cache Manager via well-defined APIs. Section 4.2 shall present the Query Processing Runtime in detail.

Cache Manager Subsystem

In turn, the Cache Manager deals with the management of data and metadata stored in the cache. Specifically, the Cache Manager subsystem comprises the following components:

- The cache replacement mechanisms that are designed and developed towards the specific characteristics of graph queries.

- A Window Manager that is responsible for the cache admission control and the maintenance of cache contents.
- A Statistics Manager taking charge of metadata, pertaining to past or current queries.
- The stores for all GraphCache-related data, including cached queries and their answer sets, currently executing (not cached) queries, metadata/statistics for both past and current queries.

These components constitutes the fundamental of the Cache Manager subsystem. More details shall be provided in section 4.3.

System Data and Control Flow

Figure 4.2 depicts the flow of control and data in GraphCache during processing of a query, following the below procedures:

- The query first arrives at the Resource Manager (1) and is then dispatched to M_{filter} and GC's filtering processors in parallel (2). At the same time, a copy of the query is added to the set of currently processed queries, called the Window Manager (to be discussed shortly).
- The filtering components use their respective indexes to produce intermediate candidate sets (3). More specifically, M_{filter} uses M_{index}, while the two GraphCache processors use GC_{index}, the set of cached graph queries and their answer sets.
- The results of this stage are then fed to the Candidate Set Pruner which produces the final candidate set GC_{CS} (4); at the same time, statistics regarding GC_{CS} and the contribution of cached graphs are gathered by the Statistics Monitor and forwarded to the Statistics Manager.
- The final candidate set then undergoes sub-iso testing using $M_{verifier}$(5); metadata pertaining to the verification time are also gathered by the Statistics Monitor and sent to the Statistics Manager.
- When the Window is full, the Window Manager selects the set of current queries to be considered for admission in the cache (6) and invokes the cache replacement algorithm (7); i.e., updates to the Cache are batched through the Window, instead of refreshing the cache on every query arrival.

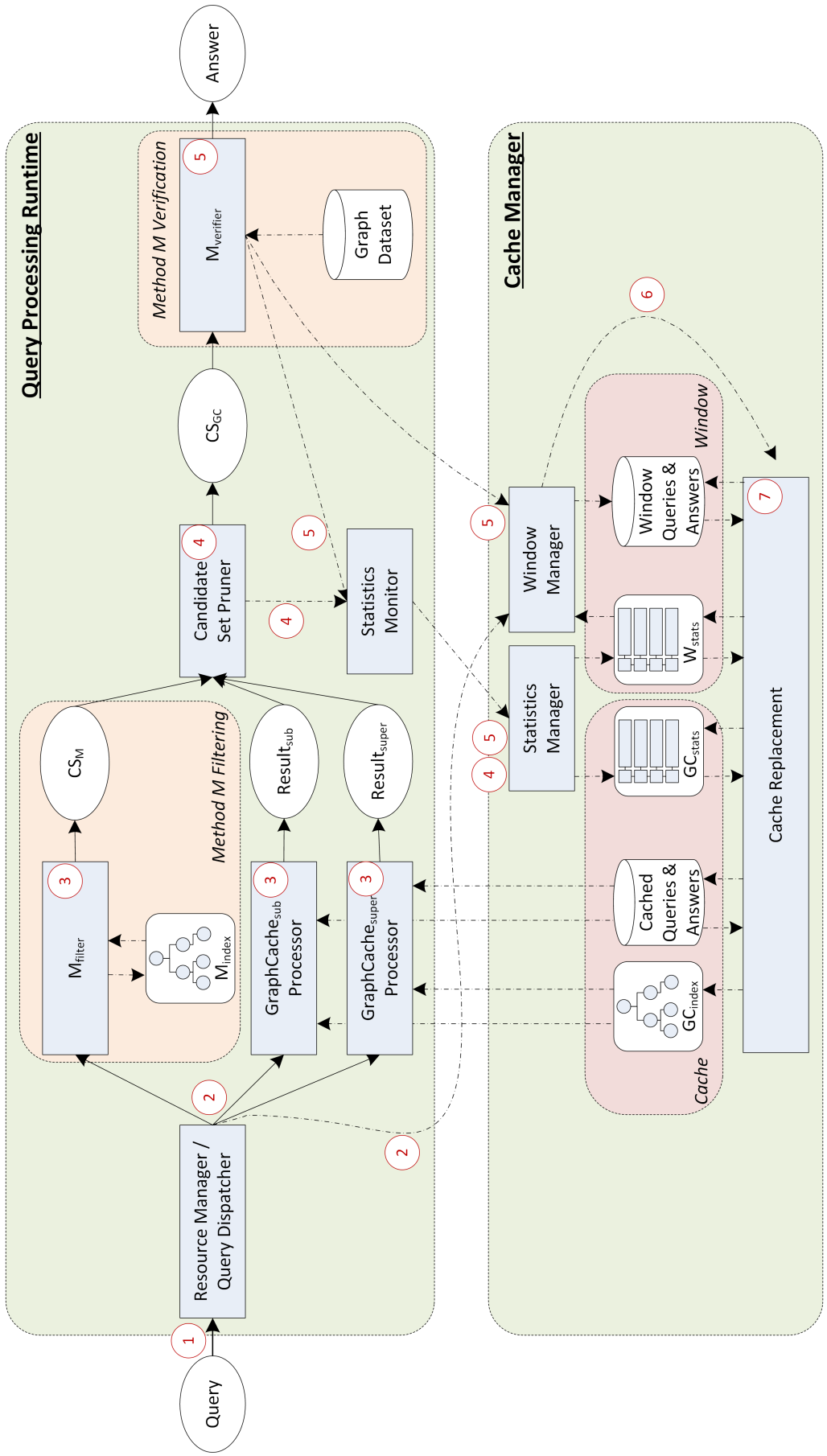


Figure 4.2: The Data and Control Flow in GraphCache

4.2 Query Processing

This section shall discuss the design and implementation of GraphCache's Query Processing subsystem, which is responsible for the execution of queries and the monitoring of key operational metrics. To be clear, it shall first describe the operations of GraphCache when caching subgraph queries and then illustrate how GraphCache can be used for supergraph queries as well.

4.2.1 Candidate Set Pruning

For completeness, this subsection shall overview the essence of iGQ [92], mapping the operations of iGQ to the components of GraphCache. With respect to more details and the formal proofs of correctness, please refer to §3.

Initially, if Method M is a FTV method, its indexing subsystem is used to build its graph dataset index as per usual. The GraphCache's data stores are initially all empty and are then populated as queries arrive and are processed. When a query g arrives at the system, M_{filter} is used to produce a first candidate set. Concurrently, GraphCache checks whether the query graph is a subgraph or supergraph of previous query graphs, through its GC_{sub}/GC_{super} Processors, to be discussed separately.

GraphCache$_{sub}$ Processor

The GraphCache$_{sub}$ Processor is responsible for identifying when a new query g is a subgraph of a previous query g'. It can be assumed that, when g' was first executed by the system, GC indexed g''s features in GC_{index} and stored its result set and relevant statistics in the cache data stores.

Figure 4.3 shows an example of such case. The new query g is processed through M_{filter}, producing candidate set $CS_M(g)$ (with four graphs $\{G_1, G_2, G_3, G_4\}$). Similarly, g is processed by the GC_{sub} Processor, which determines that there exists a previous query g', such that $g \subseteq g'$. GC then retrieves g''s cached answer set, $\{G_1, G_2\}$.

Now, consider graph $G_1 \in CS_M(g)$. Since $g \subseteq g'$ and from the answer set of g' we know that $g' \subseteq G_1$, it necessarily follows that $g \subseteq G_1$ (and, similarly, $g \subseteq G_2$). Hence, there is no point in testing g for subgraph isomorphism against G_1 or G_2 as the answer is already known. Therefore, one can safely remove $\{G_1, G_2\}$ from $CS_M(g)$, sub-iso test only the remaining graphs, and add them directly to the final answer set.

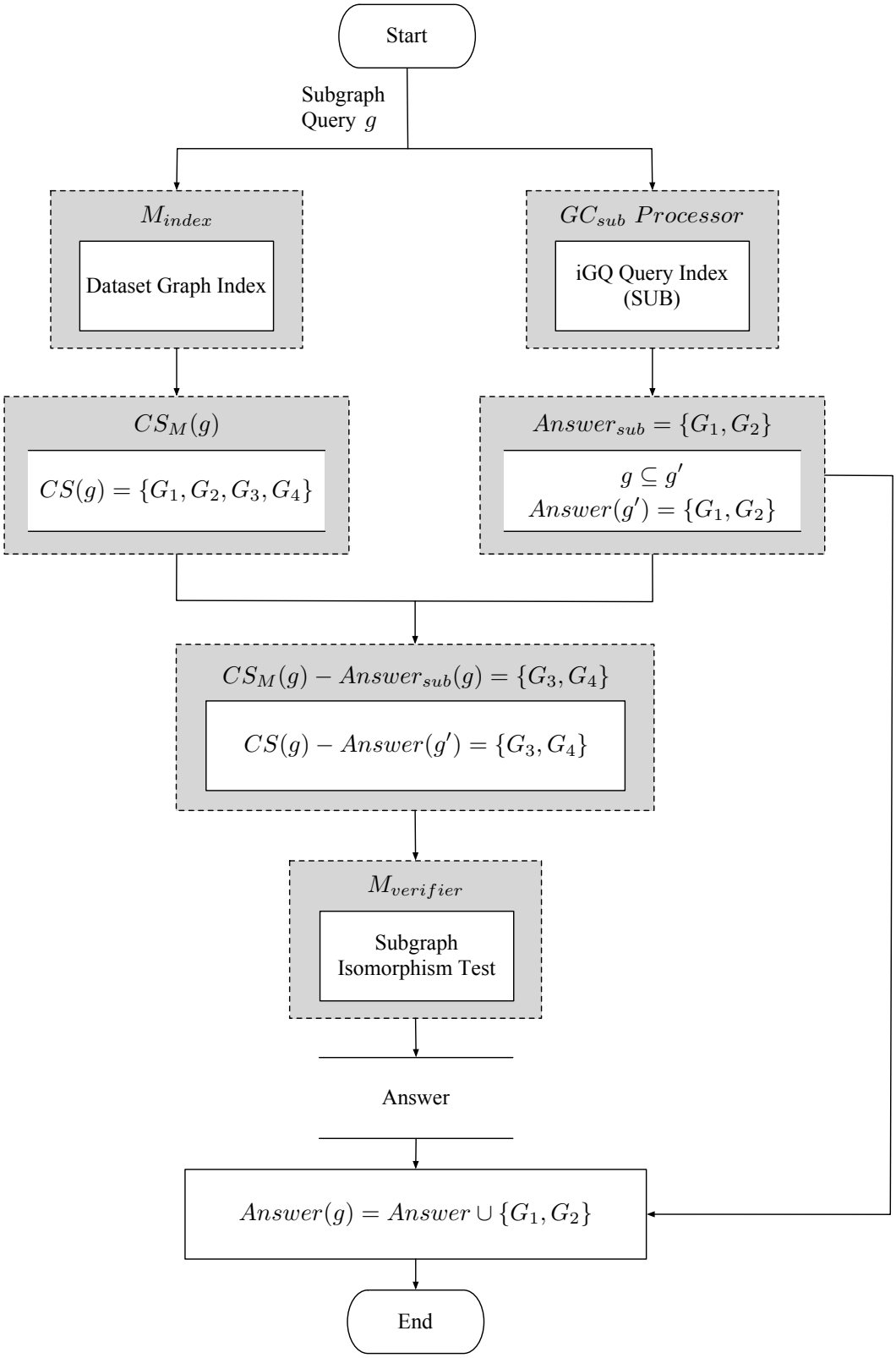


Figure 4.3: Mapping iGQ Operations to GraphCache Components: Using GraphCache$_{sub}$ Processor to Deal with the Subgraph Case

In the general case, g may be a subgraph of multiple previous query graphs g'_i. Then, the set of graphs that need be sub-iso tested is given by:

$$CS_{GC_{sub}}(g) = CS_M(g) \setminus \bigcup_{g'_i \in Result_{sub}(g)} Answer(g'_i) \tag{4.1}$$

where Result$_{sub}(g)$ contains all query graphs currently in GC$_{index}$ of which g is a subgraph. Finally, if $Answer_{GC_{sub}}(g)$ is the set of graphs in $CS_{GC_{sub}}(g)$ verified to be containing g through subgraph isomorphism testing, the final answer set for query g will be:

$$Answer(g) = Answer_{GC_{sub}}(g) \cup \bigcup_{g'_i \in Result_{sub}(g)} Answer(g'_i) \tag{4.2}$$

GraphCache$_{super}$ Processor

In turn, the GraphCache$_{super}$ Processor is responsible for identifying when a new query g is a supergraph of a previous query g'', as illustrated in Figure 4.4. Again, Method M produces its candidate set, $CS_M(g)$ (e.g., $\{G_1, G_2, G_3, G_4\}$). GC$_{super}$ then determines that there exists a previous query graph g'' such that $g'' \subseteq g$ and whose cached answer set is $\{G_1, G_{20}\}$.

The reasoning then proceeds as follows. Consider graph $G_2 \in CS_M(g)$. We know from the cached answer set above that G_2 is not in the answer set of g''. Since $g'' \subseteq g$, if $g \subseteq G_2$ were to be true then it should also hold that $g'' \subseteq G_2$; i.e., the answer set of g'' would contain G_2, which is a contradiction. Therefore, it is safe to conclude that $g \nsubseteq G_2$ and thus G_2 can be removed from $CS_M(g)$. Similarly, we can also safely remove graphs G_3 and G_4 from $CS_M(g)$, and only run a sub-iso test for G_1.

In the general case, g may be a supergraph of multiple previous query graphs g''_j. By the above reasoning, only those graphs appearing in the answer sets of all queries g''_j may possibly be supergraphs of g; Then, the set of graphs tested for sub-iso by GC is:

$$CS_{GC_{super}}(g) = CS_M(g) \cap \bigcap_{g''_j \in Result_{super}(g)} Answer(g''_j) \tag{4.3}$$

where Result$_{super}(g)$ contains all query graphs currently contained in GC$_{index}$ of which g is a supergraph. The final answer produced for query g by GC, $Answer(g)$, will be the set of graphs in $CS_{GC_{super}}(g)$ that pass the sub-iso test.

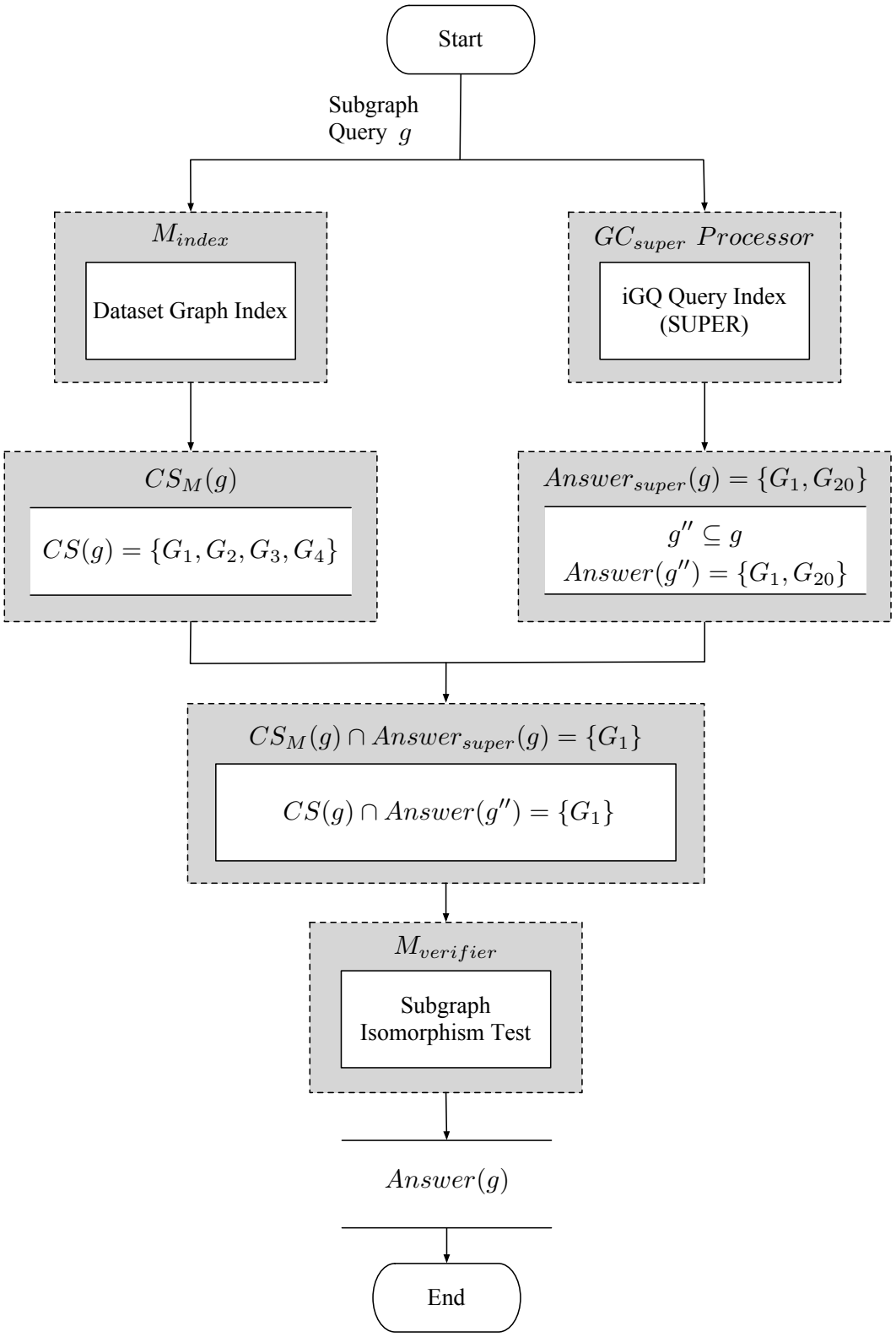


Figure 4.4: Mapping iGQ Operations to GraphCache Components: Using GraphCache$_{super}$ Processor to Deal with the Supergraph Case

Putting It All Together

The Candidate Set Pruner collects CS_M and the results of the above two Processors; it then first applies equation (4.1) on CS_M, then applies (4.3) on the result of the previous operation. The end result is a reduced candidate set $\boldsymbol{CS_{GC}}$, which is given by:

$$\boldsymbol{CS_{GC}(g)} = (\boldsymbol{CS_M(g)} \setminus \bigcup_{\boldsymbol{g'_i \in Result_{sub}(g)}} \boldsymbol{Answer(g'_i)}) \cap \bigcap_{\boldsymbol{g''_j \in Result_{super}(g)}} \boldsymbol{Answer(g''_j)} \tag{4.4}$$

$\mathrm{M}_{\boldsymbol{verifier}}$ then performs subgraph isomorphism test on $\boldsymbol{CS_{GC}}$, within which graphs containing query $\boldsymbol{g}$ are identified, coined $\boldsymbol{Result_{GC}}$. Finally, the subgraph query answer of $\boldsymbol{g}$ is returned as:

$$\boldsymbol{Answer(g)} = \boldsymbol{Result_{GC}} \cup \bigcup_{\boldsymbol{g'_i \in Result_{sub}(g)}} \boldsymbol{Answer(g'_i)} \tag{4.5}$$

Two Special Cases

Additionally, there are two cases that warrant further emphasis, since they introduce the greatest possible gains.

First, note that GraphCache can easily recognize the case where a new query, $\boldsymbol{g}$, is isomorphic to a previous cached query. Specifically, for connected query graphs, this holds when $\exists \boldsymbol{g'} \in \boldsymbol{GC_{index}}$ such that $\boldsymbol{g} \subseteq \boldsymbol{g'}$ or $\boldsymbol{g} \supseteq \boldsymbol{g'}$, and $\boldsymbol{g}$ and $\boldsymbol{g'}$ have the same number of nodes and edges. In this case, GraphCache can return the cached result of $\boldsymbol{g'}$ directly and completely avoid any further processing, including the subgraph isomorphism testing. As the subgraph isomorphism test dominates the query execution time, this is expected to be a large performance improvement.

Second, consider that $\exists \boldsymbol{g'} \in \boldsymbol{Result_{super}(g)}$ (i.e., $\boldsymbol{g'} \subseteq \boldsymbol{g}$) and $\boldsymbol{Answer(g')} = \emptyset$; then GraphCache can directly return with an empty result set. The reason is that if there were a dataset graph $\boldsymbol{G}$ such that $\boldsymbol{g} \subseteq \boldsymbol{G}$, since $\boldsymbol{g'} \subseteq \boldsymbol{g}$ we would conclude that $\boldsymbol{g'} \subseteq \boldsymbol{G}$, which implies that $\boldsymbol{G} \in \boldsymbol{Answer(g')}$, contradicting the fact that $\boldsymbol{Answer(g')} = \emptyset$; thus, no such graph $\boldsymbol{G}$ can exist and the final result set is necessarily empty.

Supergraph Query Processing

As mentioned earlier, GraphCache can expedite both subgraph *and* supergraph query processing. In the latter case, the filtering components of GraphCache remain unchanged, but the handling of the return answer sets is the exact inverse of what happens for subgraph queries. Briefly, given a supergraph query processing Method M and a supergraph query

g, the union of the answer sets of graphs in $\boldsymbol{Result}_{super}(g)$ are removed from $\boldsymbol{CS}_M(g)$ and added to $\boldsymbol{Answer}_{GC_{super}}(g)$, and the graphs not appearing in the intersection of the answer sets of graphs in $\boldsymbol{Result}_{sub}(g)$ are completely subtracted from $\boldsymbol{CS}_M(g)$.

Also, the first special case still holds, but for the second special case processing terminates when $\exists g' \in \boldsymbol{Result}_{sub}(g)$ such that $\boldsymbol{Answer}(g') = \emptyset$. The intuition behind this design follow the same reasoning as those in subgraph queries and are not repeated here.

4.2.2 Statistics Monitoring

The final component of Query Processing Runtime subsystem is the Statistics Monitor. This is a lightweight layer, implemented as a wrapper library allowing components of this subsystem to record various statistics (see §4.3.1) and to communicate them to the Statistics Manager component of the Cache Manager subsystem.

As to the GraphCache system, the Statistics Monitoring component works for two purposes: (i) measuring the quantities of query processing for performance evaluation; (ii) collecting metrics to provide support for the cache management. Hence, statistics pertaining to these two categories shall be demonstrated separately as follows.

Measurement of Query Performance

Metrics in this category focus on the performance of the newly coming query that is executed with the aid of previous queries. Currently the following quantities are monitored:

- Static metrics regarding the characteristics of the query graph, including the number of nodes, edges and distinct labels in the query.
- The number of dataset graphs in the candidate set and answer set.
- Total filtering time pertaining to executing the three filtering components in parallel, including M_{filter}, GraphCache$_{sub}$ Processor and GraphCache$_{super}$ Processor, as well as the break-down time of each said component.
- Total verification time of the query, i.e., testing each dataset graph in the candidate set so as to determine whether it appears in the answer set of the query graph.
- The number of times that the query was identified being subgraphs of previous queries by GraphCache$_{sub}$, ditto for the number of matches discovered by GraphCache$_{super}$.
- Signals indicating whether the new query benefits from the two special cases and hence warrant the exemption of any sub-iso tests in the verification stage.

Metrics for Cache Management

As mentioned earlier, the processing of each new query utilizes the knowledge derived from those stored queries. Hence, the following shall present relevant measurements with regard to the contribution of cached queries.

Table 4.1: Metrics Pertaining to Cached Queries in GraphCache

metric	description
hit num	the number of times that a cached graph has been hit
last hit	the most recent time of a cached graph being hit
CS_M reduction	total reduction in the candidate set of new queries
SI cost reduction	total time saving for new queries

According to Table 4.1, metric **hit num** counts the occurrences when a cached graph facilities query processing, either the hits happen in GraphCache$_{sub}$ Processor or GraphCache$_{super}$ Processor. In turn, **last hit** is constantly refreshed by the most recent hit time of a graph in cache. The said time of hit is expressed by the serial no. of query that benefits from this exact cached graph.

Metric **CS_M reduction** quantifies the contribution of a cached query in terms of the reduction on the candidate set of new queries. Hits in GraphCache$_{sub}$ Processor and GraphCache$_{super}$ Processor shall render the reductions with different principles.

More specifically, when a cached query g' is identified by GraphCache$_{sub}$ Processor such that the new query g is a subgraph of g', g' shall benefit the query processing of g, by subtracting some candidate graphs from verification, i.e., reducing sub-iso tests of $CS_M(g)$. Such reduction, namely R_{sub}, is due to the hit of g' in GraphCache$_{sub}$ Processor:

$$R_{sub}(g', g) = \bigcup_{G_i \in Answer(g')} G_i \tag{4.6}$$

The reasoning proceeds as follows. To determine the reduction of cached graph g' towards the sub-iso tests for new query g, two situations are required to consider, i.e., the sub-iso tests pertaining to g with and without the alleviation of g'. As to the former situation, according to formula (4.1), the number of sub-iso tests is given by:

$$CS_M(g) \setminus \bigcup_{G_i \in Answer(g')} G_i \tag{4.7}$$

And the latter comes with $CS_M(g)$ by nature. Hence, the gap between these two situations consists of the reduction on the sub-iso tests of g, as shown in equation (4.6). For more analyses and proofs of correctness, please refer to §3.

Similarly, when a cached query g'' is detected by GraphCache$_{super}$ Processor such that the new query g is a supergraph of g'', g'' will reduce sub-iso tests of $\text{CS}_M(g)$ as well. Reduction of this case is coined R_{super}, which is given by:

$$R_{super}(g'', g) = \bigcup_{G_i \in CS_M(g) \setminus CS_M(g) \cap Answer(g_i'')} G_i \tag{4.8}$$

The intuition behind follows the similar reasoning process as that of equation (4.6), through applying formula (4.3) specifically.

Finally, the metric **CS$_M$ reduction** in Table 4.1 covers the reductions captured by processors of both GraphCache$_{sub}$ and GraphCache$_{super}$.

$$CS_M\ reduction(x) = \sum\nolimits_{g_i} |R_{sub}(x, g_i)| + |R_{super}(x, g_i)| \tag{4.9}$$

where x is a cached graph and $\{g_i\}$ denotes the set of new queries that benefit from x.

The quantity **SI cost reduction** further extends metric **CS$_M$ reduction** by taking time savings into consideration. This statistic is computed as the sum of the estimated costs of all sub-iso tests alleviated. Regarding the estimation of the individual sub-iso test time $c(g, G)$ for a query graph g against a dataset graph G, it uses the formula:

$$c(g, G) = \frac{N \times N!}{L^{n+1} \times (N - n)!} \tag{4.10}$$

where L is the number of distinct labels, n is the number of nodes in g, and N the number of nodes in G having $N \geq n$. Appendix §A shall provide more details of formula (4.10).

4.3 Cache Management

In GraphCache, the Cache Manager subsystem, which runs in parallel with the Query Processing Runtime subsystem, is dealing with the management of the data and metadata stored in the cache. We shall first discuss the various data stores handled by this subsystem and then dive into the design of its various components.

4.3.1 Data Layer

The Cache Manger maintains a number of complementary data stores, conceptually bundled together into two groups: the Cache stores and the Window stores.

Cache Stores

In overall, there are three components in the Cache stores.

First, a component storing copies of cached queries (i.e., the actual graph submitted as a query to GraphCache), alongside their result sets – the sets of dataset graph IDs containing (for subgraph queries) or being contained in (for supergraph-queries) the query graph. This component is implemented as an in-memory hash table, loaded from disk on startup and written back to disk on shutdown of the Cache Manager subsystem. In said hash table, the serial number of the query is used as the key and the query graph and result set as the value. At startup, an upper limit is set on the size of this hash table (expressed in number of records); the Cache is deemed full when this upper limit is reached.

Second, a combined subgraph/supergraph index, indexing the aforementioned query graphs to expedite subgraph/supergraph matching of future queries against past queries. We have loosely based our query index design on the GraphGrepSX subgraph query index[16], augmented with additional metadata to allow for the processing of supergraph queries. This index is loaded on startup and written back on shutdown of the Cache Manager subsystem. Our index design allows us to have a single index for both subgraph and supergraph queries, thus providing for lower disk space and I/O overhead, and a memory footprint low enough to allow for the index to be easily resident in main memory throughout the lifetime of the Cache Manager process.

Third, a component storing statistics for each cached query, implemented as an in-memory key-value store, loaded from disk on startup and written back on shutdown of GraphCache. The query serial number is again used as the key, pointing to a variable size array of columns, sorted by column name. Columns in this store include, but are not limited to:

- static query metrics such as the number of nodes, edges and distinct labels in the query;
- total filtering and verification time of the query when first executed;
- count of times the query was matched by either of the GC_{sub}/GC_{super} Processors plus number of optimal matches (see §4.2.1);
- last (most recent) time a query was matched by either Processors, expressed as the serial number of the matching query;
- total contribution of the cached query in reducing the candidate sets and processing times of future queries, expressed respectively as the number of dataset graphs removed from the candidate set of queries and the cumulative sub-iso test time alleviated.

Window Stores

On the other hand, the Window stores include two components.

First, a component storing new graph queries and their result sets, implemented in the same manner as the first component of the Cache stores above. An upper limit on the size of this store is also configured at startup; the Window is deemed full when said limit is reached.

Second, a component storing statistics for each query in the previous component, also implemented as an in-memory key-value store like the statistics component of the Cache stores. In this case, the statistics include only static information regarding the new queries, including the number of nodes, edges and distinct labels in the query, as well as the total filtering and verification time of the query.New queries are sent to the Window Manager directly from the Query Dispatcher to be added to the appropriate store, while their answer sets are added at the end of their processing.

All updates to the query statistics stores are performed through the Statistics Manager using values supplied by the Statistics Monitor. The Statistics Manager is currently implemented as a lightweight wrapper library, encapsulating accesses to the statistics stores. The design of this subsystem has explicitly been abstract enough to allow for an easy replacement of the data stores with other in-memory, on-disk or even remote/distributed stores without requiring changes to the rest of our code.

The Statistics Manager exposes an interface akin to that of contemporary key-value stores; i.e., it stores triplets of the form {key, column name, column value}, accessible either by key (returns a "row" with all triplets with the given key), or by column name alone (returns a "column" with all triplets with the given column name), or by key and column name (returns a single triplet).

4.3.2 Window Manager with Admission Control

The principle of batch management is prevalently employed for cache designs and implementations. Consider the memory hierarchy of modern computer systems. Every level caches the most recently used contents of the next level in a block-wise manner, instead of per record or per byte. Similarly, in GraphCache system, there is a specific component named Window Manger dealing with the batch management of issued queries.

The Window Manager, implemented as a separate thread, is the brain of the Cache Manager subsystem. In overall, role of the Window Manager centers in the following two aspects:

- Keeping track of the queries in the current Window and invokes the Cache Admission Control algorithm to decide whether each new query should be considered as a candidate for addition to the cache.

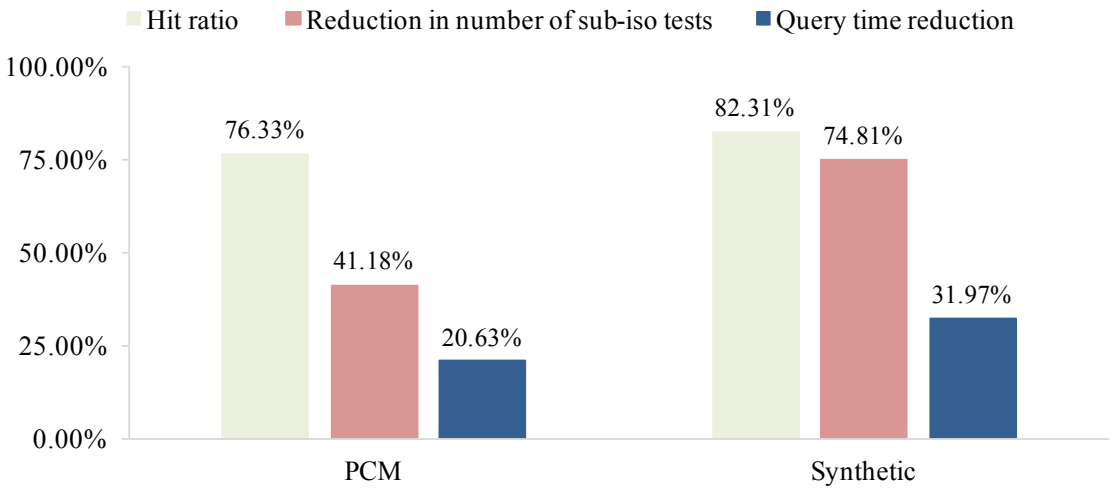


Figure 4.5: Performance of GraphCache for PCM and Synthetic Datasets

- Executing the Cache Replacement algorithms when the Window is full, and rebuilds GC_{index} to reflect any changes in the cached queries store.

The driving force behind this design was the fact that, much like all index-based graph-matching methods, our current version of GC_{index} does not support dynamic concurrent updates for the time being. Nevertheless, the design of GraphCache allows for low-latency/high-throughput processing of new queries, even while the index is rebuilt, and incurs minimal locking overhead (i.e., only for the swapping of old and new cache contents/index structures, implemented as simple in-memory reference (pointer) swaps), trading off some possible cache hits.

Cache Admission Control in GraphCache

Among the various experiments, we discovered that for some common graph datasets and workloads, the overall query time reduction attained by graph caching was very low, despite the fact that it benefited the majority of queries. Such phenomenon is coined **cache pollution**. For example, Figure 4.5 depicts the overall hit ratio (i.e., the % of queries benefiting from the cache), and the reduction in number of sub-iso tests and overall query processing time, against the "filter-then-verify" approach [42] which was found to be a top performer in [14], for two datasets: a real-world dataset PCM [91], containing 200 graphs with 377 nodes and 4,340 edges per graph on average, and a synthetic dataset consisting of 1,000 graphs with 892 nodes and 7,991 edges per graph on average. Results for other top-performing algorithms (e.g., GGSX[16]) are similar and hence omitted.

As we can see, for PCM 76.33% of executed queries were cache hits, leading to a 41.18% reduction in the number of sub-iso tests executed across the whole workload. However,

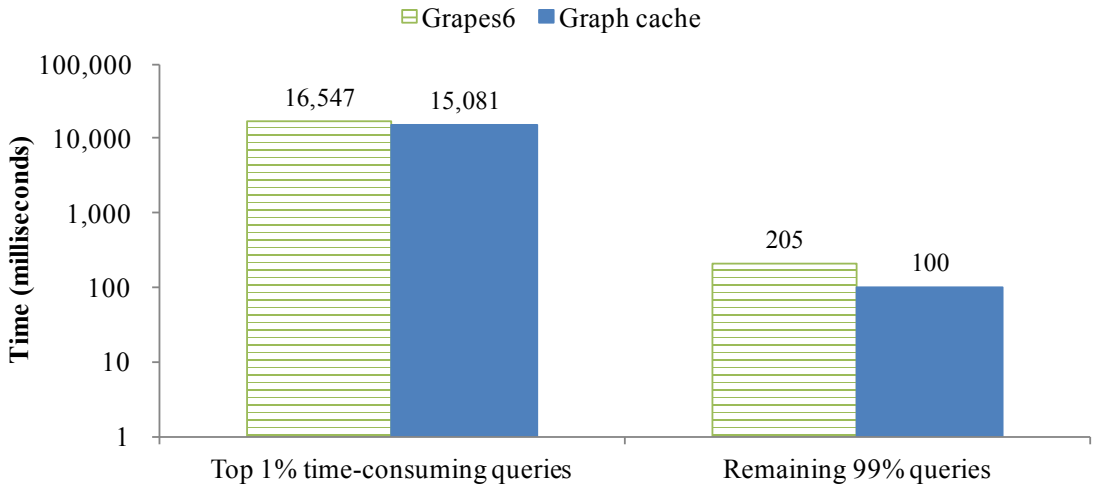


Figure 4.6: Query Times for Grapes6 on the Synthetic Dataset: with and without the Cache

the reduction in overall query time was only 20.63%. Similarly for the Synthetic dataset, a 82.31% hit ratio translated to a 74.81% overall reduction in the number of sub-iso tests, but a mere 31.97% reduction in overall query processing time. Of course, we did not expect the cache hit ratio to translate into equal reductions in sub-iso tests or query time, but the difference between said numbers is noteworthy. These results for both the PCM and Synthetic datasets across several workloads are further verified.

Figure 4.6 shows an analysis of individual query times for one of the workloads against the Synthetic dataset shown in Figure 4.5. We can see that the average query processing time of the top 1% most time-consuming queries is 2 orders of magnitude higher than that of the remaining 99% queries in the workload. The vanilla GraphCache does reduce the query time of the latter queries by half; however, for the top 1% queries, which dominate the overall query processing time, the reduction in query time was negligible. This observation explains the results in Figure 4.5 and echoes the phenomenon of graph cache pollution: that is, the graph cache is filled with graphs from the 99% part, leaving little space for graphs from the top-1%, which leads to low overall query time reduction.

To further explore the problem of graph cache pollution, more datasets with different characteristics are used. An interesting finding is that datasets with higher average degree of nodes tend to exacerbate cache pollution more than datasets having lower node degrees. Figure 4.7 depicts the coefficient of variation (i.e., standard deviation over the absolute value of the mean) of the query time for 6 workloads (W1, W2, ..., W6) against 3 different datasets (PCM, Synthetic and PDBS [87]). These workloads, each with 5,000 queries, are generated following the typical procedure for Type B workloads (see section 6.1.2 for detail).

More specially, there are two parameters being considered for the workload generations: (i) $\alpha = 1.4$ and 1.1 for the Zipf distribution (probability density function: $p(x) = \frac{x^{-\alpha}}{\zeta(\alpha)}$, where

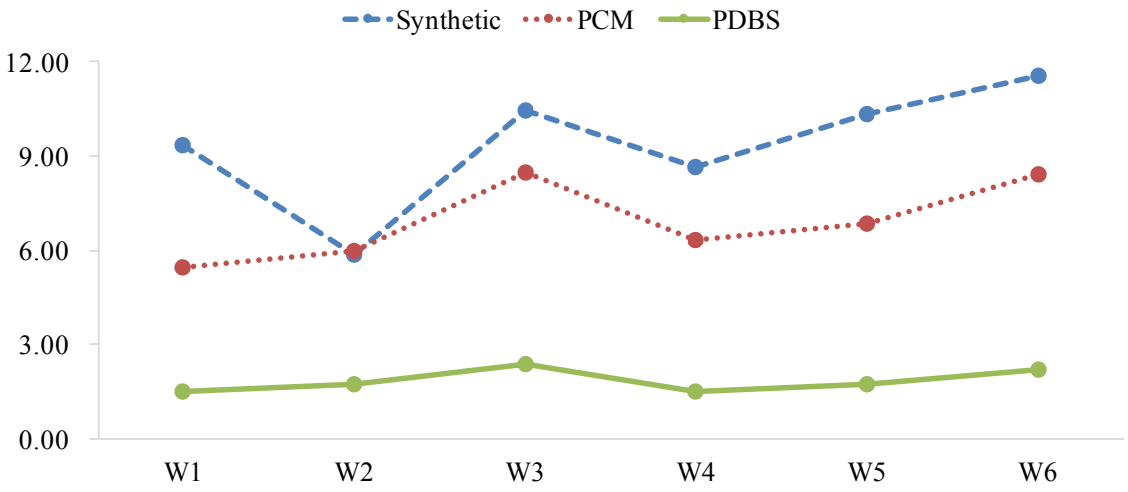


Figure 4.7: Coefficient of Variation of Query Time for PCM, Synthetic and PDBS Datasets

ζ is the Riemann Zeta function[93]). (ii) the percentage of no-answer queries – 0%, 20% and 50% Hence, each dataset results $\mathbf{2 \times 3 = 6}$ workloads (named W1, W2, ..., W6) in the end. Apparently, datasets pertaining to higher average node degrees (PCM, Synthetic) exhibit a considerably higher coefficient of variation in query time, compared with the counterpart having lower node degrees (PDBS), echoing the situation depicted in Figure 4.6.

The above analysis explained that plain GraphCache produces only small reduction in query processing time when under the influence of graph cache pollution. In order to alleviate this situation, a key principle is to effectively manage the contents of the graph cache. Recall that the essence of caching is that past popular items are expected to be popular in the future. Then, a natural conjecture for graph cache management is that past expensive queries (time-wise) are more likely to benefit later coming expensive queries. This thesis therefore proposes an **admission control** mechanism rewarding expensive queries to tackle the problem of graph cache pollution.

Such mechanism works orthogonally to the vanilla GraphCache system – new queries are batched in *windows*; when a window is full, its contents replace the least "useful" graphs in the cache, identified using a utility score. With respect to the graph cache, it is optimized by adding a switch to prevent inexpensive queries from polluting the cache.

To quantify the expensiveness of a query graph, the ratio of its verification time over its filtering time is employed. Each executed query is thus assigned an "expensiveness" score and only queries with such a score above a threshold are considered as candidates for entering the graph cache. To compute said threshold, the said mechanism examines the queries in the first few *windows* and computes an expensiveness value which would result in a predefined percentage of queries (set to 10% in the experiments) being classified as expensive.

The reasoning behind the above lies in the fact that, given a graph query processing frame-

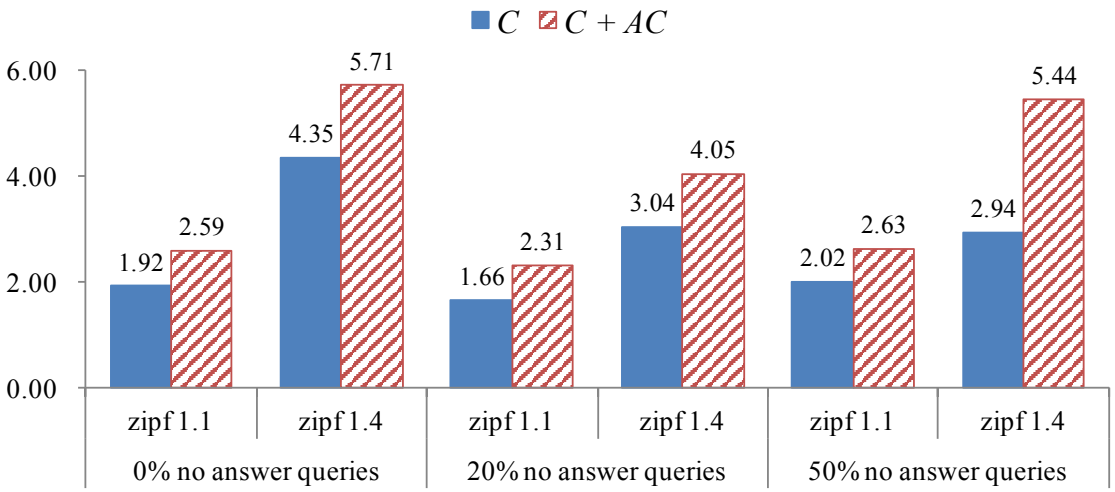


Figure 4.8: Query Time Speedups: Grapes6/PCM Dataset

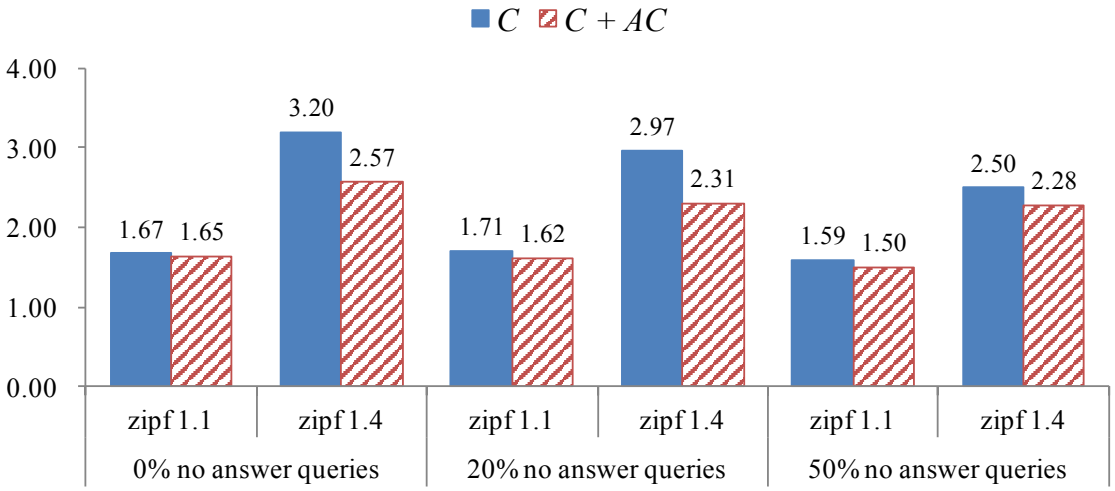


Figure 4.9: Reduction (Speedup) in Number of Sub-Iso Tests: Grapes6/PCM Dataset

work, the filtering time is relatively constant across queries (e.g., retrieving trie index [42], checking fingerprints [18], etc.), in contrast to the dramatic variance of verification times. Moreover, the verification stage dominates the query time, as has been shown in related work [13, 14], and the larger the verification time the more overwhelming this dominance is. For example, recall Figure 3.1 where the verification of queries against the PDBS [87] dataset (with averaged 3,064 edges per graph) nearly covers the total query processing time. Thus, the above expensiveness score and admission control mechanism, are a **simple yet effective** technique to guarantee that more complex queries are prioritized in the cache.

The effectiveness of cache admission control is tested using Grapes[42] and GGSX[16], for being top-performers in literature [14]. For Grapes, two alternatives with 1 (6) threads are used, denoted by Grapes1 and Grapes6. We report performance gains in terms of query processing time and of the number sub-iso tests performed, expressed as a speedup; that is,

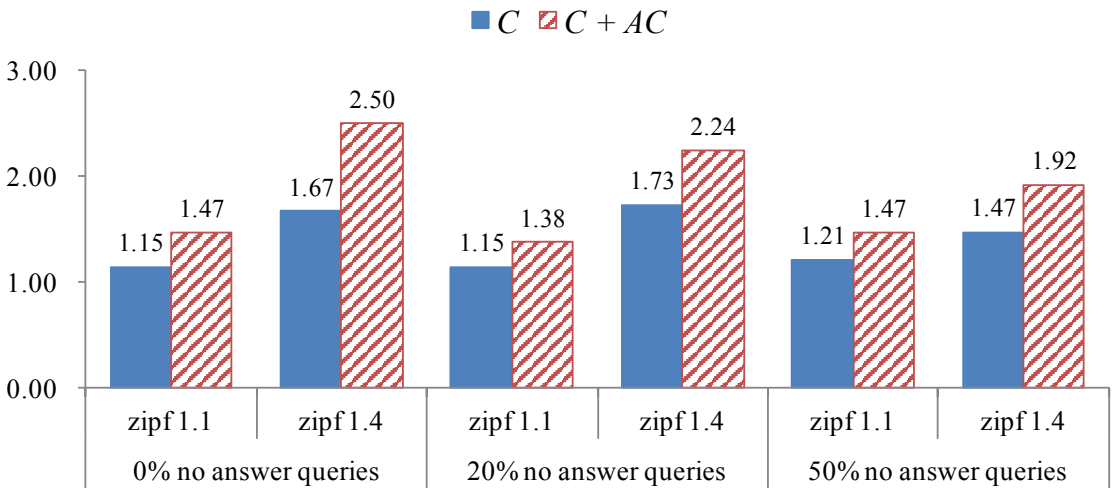


Figure 4.10: Query Time Speedups: Grapes6/Synthetic Dataset

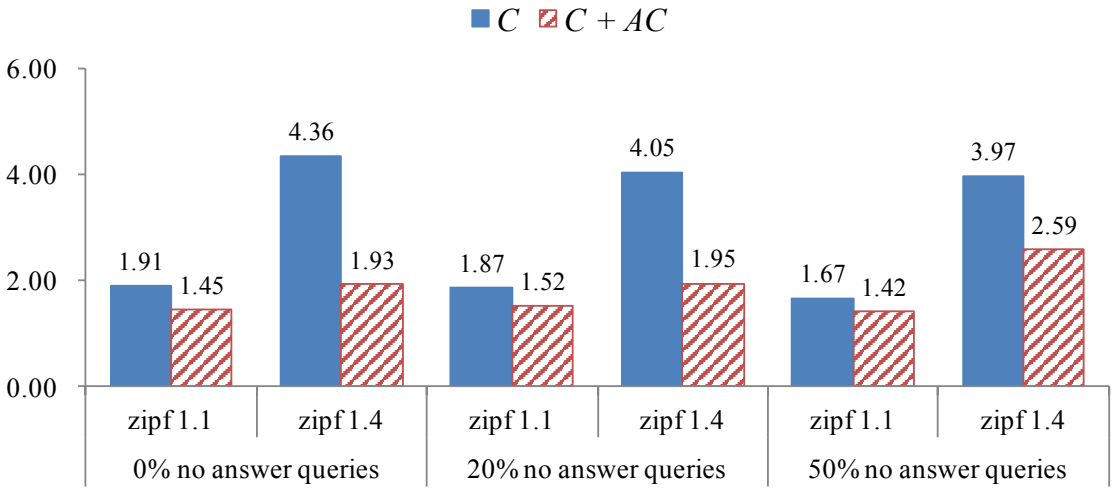


Figure 4.11: Reduction (Speedup) in Number of Sub-Iso Tests: Grapes6/Synthetic Dataset

as the ratio of the average performance (query time, number of sub-iso tests) of the base Method M, over the average performance of our solution when deployed over Method M (i.e., a speedup of X>1 indicates an improvement by a factor of X). The reported speedups are regarding both the plain GraphCache system, denoted by $\boldsymbol{C}$, and the same cache with the addition of our admission control mechanism, denoted by $\boldsymbol{C+AC}$.

Figures 4.8 and 4.9 show the speedup in query time and number of sub-iso tests for the PCM dataset. As expected, in both cases one can see that the cache performs better for higher values of the Zipf α parameter and thus more skewed query distributions. With the admission control mechanism turned on, the speedup of query times increases considerably across all workloads. It is interesting to note, though, that the corresponding speedup in the number of sub-iso tests is reduced. This validates our claim that prioritizing expensive queries over inexpensive ones can improve the overall query processing time, even if it leads

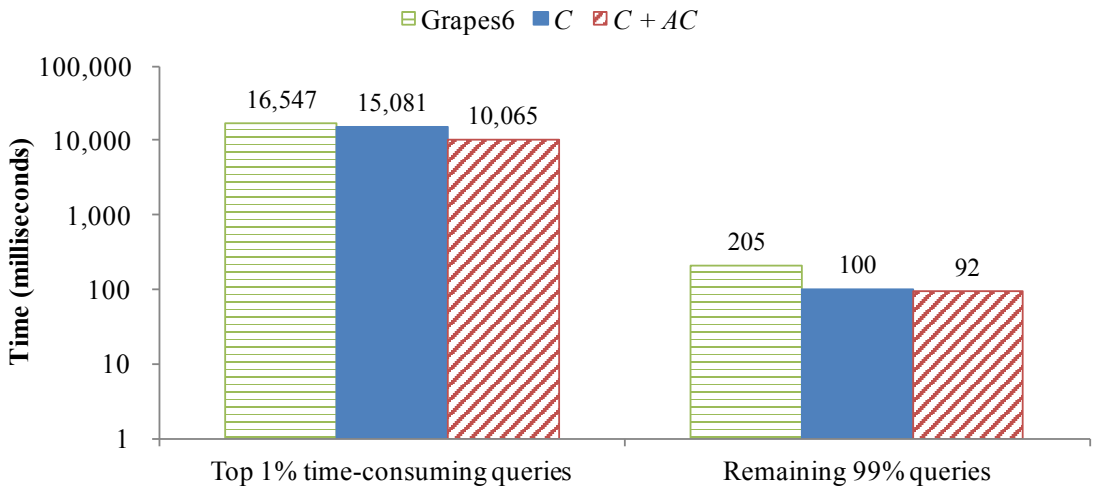


Figure 4.12: Query Times for Grapes6, C and $C+AC$ on the Synthetic Dataset

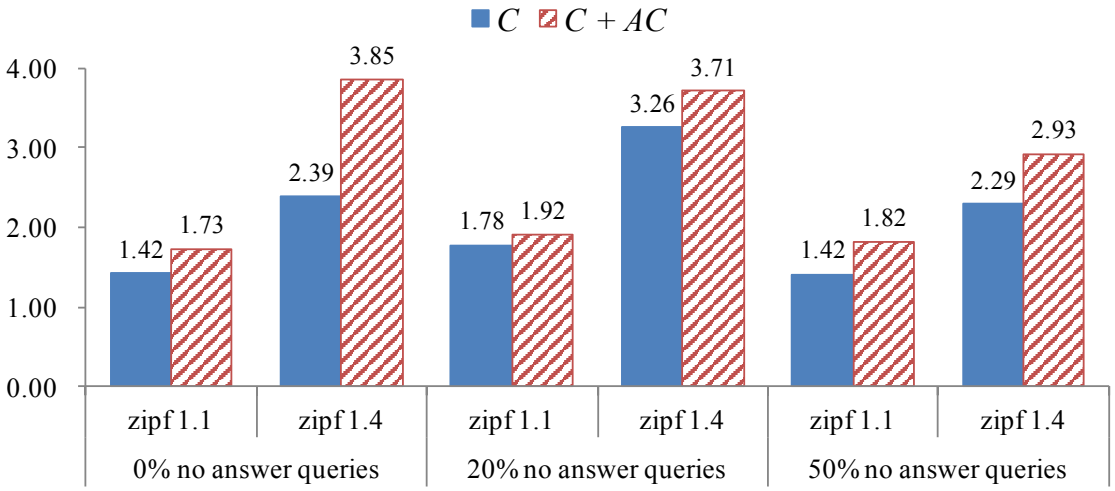


Figure 4.13: Query Time Speedups: Grapes1/Synthetic Dataset

to a number of inexpensive sub-iso tests not alleviated.

Similarly, Figures 4.10 and 4.11 depict the speedups of C and $C+AC$ against Grapes6 on the Synthetic dataset. Again, the cache admission control mechanism leads to further improvements in query time over the plain graph cache, despite the somewhat higher number of sub-iso tests performed overall.

To better understand these trends, let us concentrate on the results for 50% no-answer queries and Zipf α=1.4 in Figures 4.10 and 4.11, corresponding to the analysis shown in Figure 4.6 and now extended in Figure 4.12. The plain graph cache yields a speedup as high as 3.97 in the number of sub-iso tests; however, the resulting query time speedup is only 1.47, as the top time-consuming queries hardly benefit from the polluted cache. With the admission control mechanism, despite the lower reduction (speedup) in sub-iso tests, the overall query time

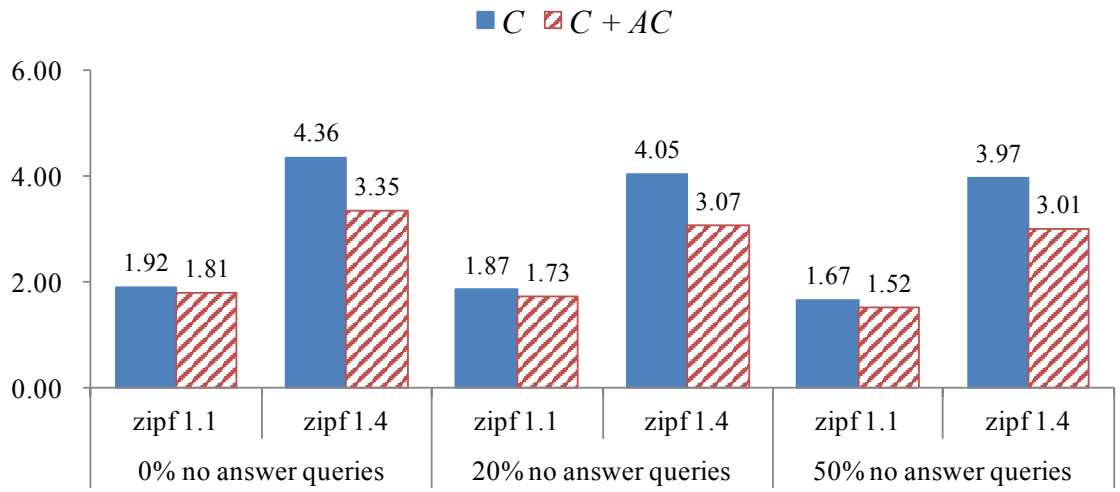


Figure 4.14: Reduction (Speedup) in Number of Sub-Iso Tests: Grapes1/Synthetic Dataset

speedup is increased due to accelerating the top time-consuming queries in the workload.

Last, Figures 4.13 and 4.14 present results against Grapes1 for the Synthetic dataset, where we can see the same trends as in the case of Grapes6. Results of other cases and for GGSX[16] are similar and hence omitted. In overall, the proposed cache admission control mechanism is effective in further improving the query time speedup of graph caches, as showcased with different workloads on both real-world and synthetic datasets.

4.3.3 Cache Replacement Policies

With queries continuously arriving and the limited memory for cache stores, GraphCache requires strategies to handle the replacement. This section shall first describe a general framework of GraphCache in dealing with the cache replacement and then propose a set of replacement policies exclusively for GraphCache, using an example to illustrate the different tradeoffs of various polices.

GraphCache Framework for Cache Replacement

GraphCache is designed with a framework allowing for the flexibility of accommodating various strategies. Figure 4.15 shows the operations in such framework, among which there exist two “black boxes” that are dependent with the replacement policy, i.e., steps accompanied with notations of “Step 2” and “Step 11”.

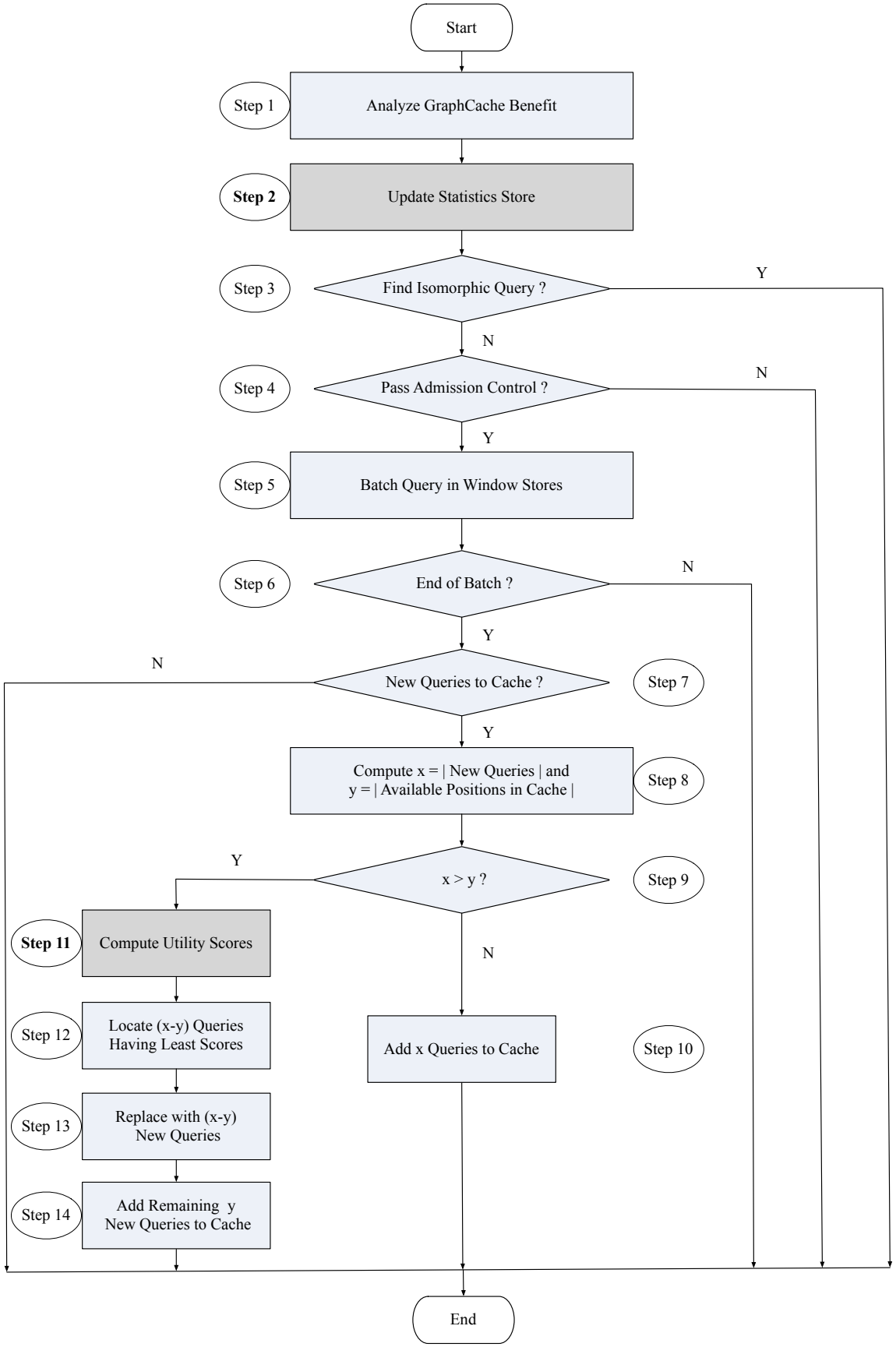


Figure 4.15: GraphCache Framework for Cache Replacement

- **Step 1:** The benefit of GraphCache is analyzed, providing the matches in GraphCache$_{sub}$ Processor and GraphCache$_{super}$ Processor when executing the query $\boldsymbol{g}$. Briefly, matched graphs $\{\boldsymbol{g'}\}$ are further categorized into three sets: (i) $\mathbf{S_1}$ contains id of graphs that are matched in GraphCache$_{sub}$ Processor only; (ii) $\mathbf{S_2}$ consists of (id of) graphs being matched in GraphCache$_{super}$ Processor exclusively; (iii) $\mathbf{S_3}$ covers (id of) those matched in the two above processors.

- **Step 2:** Specific to the replacement policies, statistics regarding the savings of cached graphs are updated, such as **hit num**, **last hit**, **CS$_M$ reduction** and **SI cost reduction** (see their interpretations in §4.2.2).

- **Step 3:** The system determines whether the query $\boldsymbol{g}$ is isomorphic to a cached graph $\boldsymbol{g'}$. This is easily achievable, provided directly by a signal indicating whether query $\boldsymbol{g}$ benefits from the first special case. If yes, there is no reason to cache $\boldsymbol{g}$, as its isomorphic graph $\boldsymbol{g'}$ has been cached. Hence, further steps can be avoided and the procedure is ended.

- **Step 4:** Query $\boldsymbol{g}$ is checked whether it satisfy the requirements of passing the cache admission control (please refer to §4.3.2 for detail). If not, the procedure comes to an end as well.

- **Step 5:** The query graph $\boldsymbol{g}$, together with its result set and static statistics (detailed in §4.3.1), is batched in Window Stores.

- **Step 6:** According to the experiment setting for the upper limit of window size, it is determined whether the current query $\boldsymbol{g}$ is at the end of batch. If not, the procedure is ended.

- **Step 7:** Window Stores are checked whether there are new queries being batched. An extreme case is that for each query in current batch, either it is isomorphic to those in cache already, or it does not survive the cache admission control, which in turn leads to an end of the procedure. Otherwise, further steps are triggered.

- **Step 8:** Two quantities pertaining to the number of new queries in Window Stores (represented by $\boldsymbol{x}$) and of the available positions in cache (in term of $\boldsymbol{y}$) are calculated.

- **Step 9:** Numeric comparison between $\boldsymbol{x}$ and $\boldsymbol{y}$ is performed. If $\boldsymbol{x}$ is smaller, go to **Step 10**. Otherwise, go to **Step 11**.

- **Step 10:** Move $\boldsymbol{x}$ queries directly from Window Stores to Cache Stores and then end the procedure.

- **Step 11:** Since there are not enough positions in cache to hold new queries, replacement will take place. During this process, utility scores of cached graphs are first computed, following the formula defined by each replacement policy (to present shortly).
- **Step 12:** Then, $(x - y)$ cached graphs with the lowest scores are located.
- **Step 13:** Cache Stores occupied by these located graphs are replaced directly by $(x - y)$ new queries in Window Stores.
- **Step 14:** The remaining y new queries in Window Stores are cached in. Thus finishes the branch involving cache replacement, as well as the whole procedure of cache management.

Next, the "black boxes" shall be revealed in each concerned replacement policy. To tackle the cache replacement, GraphCache is bundled with a number of novel strategies (POP, PIN, PINC and HD), each offering different trade-offs and performance characteristics for various datasets and query workloads. These strategies shall be described here and their relative performance is left for §6. In all cases, it will access query statistics through the Statistics Manager's key-value store interface and return the IDs of queries to be cached out, i.e., queries are assigned a "utility" value and those with the lowest such values are cached out.

To help illustrate all cache replacement algorithms considered for GraphCache, Table 4.2 presents a snapshot of GC_{stats} for a number of hypothetical cached queries, as an example, say, the cache now is full of 6 entries and window size w is 2. In all cases, assume that the replacement algorithm is invoked at time point 99 (i.e., right after the query with serial number 99 was executed [1]) and needs to remove two entries from the cache, thus has to find the two entries with the lowest utility value.

Least Recently Used (LRU)

LRU discards the least recently used items from the cache. Hence, the utility of each cached graph is its last "hit" time – the serial no. of the last query that are matched and accelerated by the said cached query, either in GraphCache_{sub} Processor or $\text{GraphCache}_{super}$ Processor.

Please note that in GraphCache there exist cases that cached graphs are matched however not rendering savings to the query processing. For example, when a cached graph g' is hit by query g in GraphCache_{sub} Processor only and the answer set of g' in cache stores is empty, g' shall not provide any real help for the processing of query g. Therefore, the definition of **last hit time**, as well as other metrics regarding the cache management of GraphCache, stresses the real savings. [2]

[1] The serial number used in GraphCache starts from zero by default, including the id of graph and of batch.

[2] In GraphCache system, if without any specific illustration, hits of a cached graph by default refer to those rendering real savings for query processing.

Table 4.2: An Example: Cached Query Statistics

SerialNo / Query ID	Last Hit	Number of Hits (H)	CS_M Reduction (R)	SI Cost Reduction (C)
11	91	23	170	2600
13	51	32	80	1200
37	69	26	376	780
53	78	13	210	360
82	90	5	120	150
91	95	4	10	270

Turning attention to the two black boxes in Figure 4.15. For LRU policy, the metric concerned in **Step 2** and **Step 11** is **last hit time** of each cached graph. There are two stages involved in **Step 2**: (i) among cached graphs, those rendering savings for the query g are first identified; (ii) the entry **last hit time** of each identified graph is updated by the ID of g.

Algorithm 3 details the procedures for the first stage. More specifically, a BitSet B_s is initialized with length of the upper limit of cache size (each bit is false by default). Providing the various matching categories, the strategies of determining real savings are different. For each cached graph in set S_1, it can introduce query savings only with non-empty answer set, which is represented by a BitSet structure (lines 5–9). Processing set S_2 is more complicated, where both the candidate set of query g and answer set of cached graph g' are considered (lines 10–18). Intersection of these two sets is performed efficiently through one BitSet operation. Comparing the cardinality of said candidate set and intersection result, only if the former is properly larger can g' be bound to render savings. As to each graph g' in set S_3, as long as the candidate set of query g is non-empty, g' contributes by removing all the sub-iso tests pertaining to g (lines 19–23).

The returned BitSet B_s is then used by the second stage of **Step 2** in a straight-forward manner. That is, for each bit having value of true, its corresponding graph in cache refreshes the entry **last hit time** with the ID of query g. In turn, the utility score of LRU policy could be retrieved directly from **last hit time** and the black box of **Step 11** hence has none additional computations.

In the running example, cached queries with serial number 13 and 37 would be cached out. LRU is a simple and very popular policy in several traditional caches. However, it builds on the assumption that the longer a query has not been posed, the less probable it is to see this query again in the future. It thus fails to identify cases of queries which have contributed huge savings to query processing although not having been used in a while. In the said

Algorithm 3 Identifying Cached Graphs Rendering Real Savings for a Query g

Input: Set S_1, Set S_2 and Set S_3 containing id of graphs that are matched in GraphCache$_{sub}$ Processor only, GraphCache$_{super}$ Processor only and both, respectively
Output: BITSET B_s with each bit indicating whether a cached graph renders real savings for executing g

Initialize BITSET B_s with length of the upper limit of cache size (each bit is set false, i.e., 0)
for all $g' \in S_1$ **do**
    if g'.answer.cardinality() > 0 **then**
        B_s.set(g')
    end if
end for
for all $g' \in S_2$ **do**
    BITSET tmp = g.candidate
    tmp.and(g'.answer)
    x = g.candidate.cardinality()
    y = tmp.cardinality()
    if $x > y$ **then**
        B_s.set(g')
    end if
end for
for all $g' \in S_3$ **do**
    if g.candidate.cardinality() > 0 **then**
        B_s.set(g')
    end if
end for
return B_s

example, one can see that query 13 has been matched the most times, but still is evicted.

Popularity-based Ranking (POP)

Ideally, we would prefer a replacement policy that would take into account the **popularity** of queries. This leads to the second policy considered in GraphCache system: POP (short for Popularity-based Ranking).

More specifically, the utility of each cached graph is assigned by:

$$U_{pop} = \frac{H}{A}, \tag{4.11}$$

where H is the number of times a cached query was hit and A reflects its age given by:

$$A = (\lfloor \frac{g_{id}}{w} \rfloor - \lfloor \frac{g'_{id}}{w} \rfloor) \times w \tag{4.12}$$

Table 4.3: POP Replacement Policy: Looking into the Example of Table 4.2

SerialNo/ Query ID	Number of Hits (H)	Age (A)	Utility (U)
11	23	88	0.26
13	32	86	0.37
37	26	62	0.42
53	13	46	0.28
82	5	16	0.31
91	4	8	0.5

in which w is the upper limit of window size (i.e., number of queries per batch); $\lfloor \frac{g_{id}}{w} \rfloor$ and $\lfloor \frac{g'_{id}}{w} \rfloor$ pertain to the serial batch number of query g and cached query g' separately. In turn, A could be interpreted as the number of queries that g' has "witnessed" (i.e., bearing the opportunity to expedite) since it is being cached. Take an extreme example. If there exists a super beneficial query g' in the cache such that it has been hit by every following query, then by equation (4.11), the said g' shall possess a POP utility score of 1.

As to the black boxes in Figure 4.15, POP implements as follows. Similar as LRU, POP implements **Step 2** in two stages. (i) again, Algorithm 3 is called to identify cached queries that have contributed the execution of query g; (ii) for each identified graph, its entry of **number of hits** is increased by 1. By using formulas (4.11) and (4.12), **Step 11** returns the utility score for each cached query, which will then serve as the criteria for replacement of POP.

In overall, the POP replacement policy manages to take both popularity and age into the consideration of utility score. Table 4.3 shows the break-down analysis of POP policy regarding the example in Table 4.2, in which the eviction apply for queries 11 and 53.

POP + Number of Sub-Iso Tests (PIN)

As mentioned before, unlike traditional exact-match caching schemes in which each cache hit saves one disk/network IO, cache hits in GraphCache may result vastly different reductions in query processing times. One reason lies in that cache hits reduce the candidate set of the coming query by possibly vastly different amounts. However, neither LRU nor POP (actually, none of the known replacement policies) take this into account. This gives rise to the next, exclusive to GraphCache, replacement policy: PIN (short for Popularity and sub-Iso test Number).

To this end, PIN extends the utility by further considering the different savings in number of subgraph isomorphism tests rendered by various cache hits. For each cached graph, the

Algorithm 4 Calculating the Savings of Each Cached Query in Number of Sub-iso Tests Pertaining to the Execution of a Query g

Input: Set S_1, Set S_2 and Set S_3 containing id of graphs that are matched in GraphCache$_{sub}$ Processor only, GraphCache$_{super}$ Processor only and both
Output: HASHMAP T_s with key of each cached query g' (id) and value of a number reflecting the amount of sub-iso tests that g' has reduced when executing g

Initialize T_s with an empty HASHMAP
for all $g' \in S_1$ **do**
    if g'.answer.cardinality() > 0 **then**
        T_s.put(g', g'.answer.cardinality())
    end if
end for
for all $g' \in S_2$ **do**
    BITSET tmp = g.candidate
    tmp.and(g'.answer)
    x = g.candidate.cardinality()
    y = tmp.cardinality()
    if $x > y$ **then**
        T_s.put(g', $(x - y)$)
    end if
end for
for all $g' \in S_3$ **do**
    if g.candidate.cardinality() > 0 **then**
        T_s.put(g', g.candidate.cardinality())
    end if
end for
return T_s

utility of PIN is assigned by:

$$U_{pin} = \frac{R}{A}, \tag{4.13}$$

where R is the total number of subgraph isomorphism tests alleviated by the cached query and A is the same aging factor as above by equation (4.12). The utility formula of PIN (4.13) can also be rewritten as:

$$U_{pin} = \frac{H}{A} \times \frac{R}{H} \tag{4.14}$$

which can be interpreted as the probability of the cached query being hit (i.e., its popularity), times the average savings in number of subgraph isomorphism test per hit. The second factor expresses PIN's concern regarding the said saving difference in number of subgraph isomorphism tests.

Recall the two black boxes in Figure 4.15. The implementation of PIN follows the similar paradigm as LRU and POP, however bearing different metric of interest CS_M **Reduction**. Again, **Step 2** of PIN is further divided into two stages: (i) savings in number of subgraph

Table 4.4: PIN Replacement Policy: Looking into the Example of Table 4.2

SerialNo/ Query ID	CS_M Reductions (R)	Age (A)	Utility (U)
11	170	88	1.93
13	80	86	0.93
37	376	62	6.06
53	210	46	4.57
82	120	16	7.5
91	10	8	1.25

isomorphism tests are calculated per cached graph; (ii) for each concerned query in cache, the entry of CS_M **Reduction** is updated.

Algorithm 4 shows the operations as to the first stage of **Step 2**. Different from Algorithm 3 that targets at detecting cached graphs with savings, Algorithm 4 further looks into how many these savings are, in term of the subgraph isomorphism test numbers. To this end, per cached query is accompanied with a number reflecting the said saving amount (see line 7, 16 and 21 for the details pertaining to different categories of cache hit).

The second stage of **Step 2** is in charge of updating the cache statistics with the HASHMAP T_s returned by Algorithm 4. More specifically, for each concerned graph g' (key part of T_s), its entry of CS_M **Reduction** is added by a number that associates with g' (value part of T_s). And in **Step 11**, PIN computes the utility score by applying equations (4.13) and (4.12), in which the R element is directly retrieved from metric CS_M **Reduction**.

Turing attention to the example in Table 4.2. When the PIN replacement policy is evoked, Table 4.4 presents the detail of processing. According to the principle of evicting queries with least utilities, graphs with ID of 13 and 91 will be cached out.

PIN + Sub-Iso Tests Costs (PINC)

PIN takes into account the number of sub-iso tests alleviated. Another GraphCache-exclusive replacement policy PINC (abbreviation of Popularity, sub-Iso test Number, and time Cost) is proposed, which further considers the possibly vast differences in query execution times. PINC assigns each cached query a utility value equal to:

$$U_{pinc} = \frac{C}{A}, \tag{4.15}$$

where A is the same aging factor as mentioned, and C is the total decrease in query processing time due to the cached query. Alas, this figure cannot be computed unless the concerned

subgraph isomorphism tests are performed, which is a moot point in our case; instead, we use a heuristic to estimate this cost (see Appendix §A for the detail).

PINC may improve upon PIN's utility value computation by considering the actual (estimated) time cost of alleviated subgraph isomorphism tests instead of deeming them all equivalent. PINC's utility formula can be rewritten as:

$$\boldsymbol{U_{pinc}} = \frac{\boldsymbol{H}}{\boldsymbol{A}} \times \frac{\boldsymbol{R}}{\boldsymbol{H}} \times \frac{\boldsymbol{C}}{\boldsymbol{R}} \tag{4.16}$$

which could be interpreted as the probability of a cached graph being hit, times the average savings in number of sub-iso tests per hit, times the average estimated time cost per saved sub-iso test.

Regarding the black boxes in Figure 4.15, PINC rests on a more sophisticated structure of savings, which shall be presented shortly. As per usual, **Step 2** is carried out in two stages: (i) discovering the details of each reduced sub-iso test; (ii) updating the relevant cache statistics.

The first stage is implemented by Algorithm 5, which returns the structure mentioned above – a HASHMAP D_s having key of each cached query $\boldsymbol{g'}$ and associated value of a VECTOR containing dataset graphs such that their sub-iso testings with regard to the query $\boldsymbol{g}$ are reduced by $\boldsymbol{g'}$. In the second stage, there are two metrics to update. Besides the $\boldsymbol{CS_M}$ **Reduction** as per usual, PINC maintains an extra metric of $\bar{\boldsymbol{c}}$, which records **the average (estimated) time cost per saved sub-iso test by each cached graph**. More specifically, for each said $\boldsymbol{g'}$, the associated value part in D_s (in term of VECTOR) is retrieved. For each element $\boldsymbol{G}$ in the VECTOR, it means that the sub-iso test of query $\boldsymbol{g}$ against dataset graph $\boldsymbol{G}$ is removed by the cached query $\boldsymbol{g'}$; hence the $\boldsymbol{CS_M}$ **Reduction** of $\boldsymbol{g'}$ is added by one and the said average time cost $\bar{\boldsymbol{c}}$ of $\boldsymbol{g'}$ is updated by:

$$\bar{\boldsymbol{c}} = \frac{\boldsymbol{R_0} \times \bar{\boldsymbol{c}}_0 + \boldsymbol{c}(\boldsymbol{g}, \boldsymbol{G})}{\boldsymbol{R}} \tag{4.17}$$

where the $\boldsymbol{R_0}$ and $\boldsymbol{R}$ represent the value of $\boldsymbol{CS_M}$ **Reduction** before/after the described adding operation; $\bar{\boldsymbol{c}}_0$ is the currently stored value for $\boldsymbol{g'}$ (before the update). And the estimated time cost of $\boldsymbol{c}(\boldsymbol{g}, \boldsymbol{G})$ is given by:

$$\boldsymbol{c}(\boldsymbol{g}, \boldsymbol{G}) = \frac{\boldsymbol{N} \times \boldsymbol{N!}}{\boldsymbol{L}^{\boldsymbol{n+1}} \times (\boldsymbol{N} - \boldsymbol{n})!}, \tag{4.18}$$

where $\boldsymbol{L}$ is the number of distinct labels, $\boldsymbol{n}$ is the number of nodes in $\boldsymbol{g}$, and $\boldsymbol{N}$ the number of nodes in $\boldsymbol{G}$ having $\boldsymbol{N} \geq \boldsymbol{n}$ (to be detailed in Appendix §A).

As to the **Step 11** in Figure 4.15, it performs the computation of the metric of **SI Cost Reduction (C)**, by returning the product of the said $\boldsymbol{CS_M}$ **Reduction (R)** and $\bar{\boldsymbol{c}}$. And the utility score of each cached graph is then calculated using formula (4.15).

Algorithm 5 Discovering Each Sub-iso Test Reduced from the Execution of a Query g

Input: Set S_1, Set S_2 and Set S_3 containing id of graphs that are matched in GraphCache$_{sub}$ Processor only, GraphCache$_{super}$ Processor only and both
Output: HASHMAP D_s with key of each cached query g' (id) and value of a VECTOR storing the id of dataset graphs that are reduced from candidate set of query g

Initialize D_s with an empty HASHMAP
for all $g' \in S_1$ **do**
    if g'.answer.cardinality() > 0 **then**
        D_s.put(g', g'.answer)
    end if
end for
for all $g' \in S_2$ **do**
    BITSET tmp_0, tmp_1 = g.candidate
    tmp_0.and(g'.answer)
    tmp_0.flip()
    tmp_1.and(tmp_0)
    if tmp_1.cardinality() > 0 **then**
        Initialize an empty VECTOR tmp_2
        for all $i \in tmp_1.index$ **do**
            if tmp_1.get(i) **then**
                tmp_2.add(i)
            end if
        end for
        D_s.put(g', tmp_2)
    end if
end for
for all $g' \in S_3$ **do**
    if g.candidate.cardinality() > 0 **then**
        D_s.put(g', g.candidate)
    end if
end for
return D_s

PINC also follows the principle of discarding queries with lowest utility score. Recall the example in Table 4.2. Table 4.5 shows the process of figuring out the two graphs to evict. And in the end, it is queries 53 and 82 to be replaced.

The Hybrid Dynamic Policy (HD)

As the cost component in PINC is only an estimation, using it does not always lead to improvements in GC's net query processing time. As a matter of fact, we have observed through a large number of experiments, that when the values of the R utility component exhibit a high variability, they are discriminative enough on their own. In such cases, taking the estimated cost into account can actually lead to lower time gains (i.e., PIN performing better than

Table 4.5: PINC Replacement Policy: Looking into the Example of Table 4.2

SerialNo/ Query ID	SI Cost Reductions (C)	Age (A)	Utility (U)
11	2600	88	29.55
13	1200	86	13.95
37	780	62	12.58
53	360	46	7.83
82	150	16	9.38
91	270	8	33.75

PINC). However, when the values of $\boldsymbol{R}$ exhibit a low variability, adding in the $\boldsymbol{C}$ component leads to considerable query processing time improvements.

Thus, the last replacement policy considered in this work (also exclusive to GraphCache), coined the **hybrid dynamic** policy (HD), coalesces both PIN and PINC. More specifically, when the HD policy is invoked, it first retrieves the $\boldsymbol{R}$ component from GC_{stats} and computes its variability [94] by using the (squared) coefficient of variation ($\boldsymbol{CoV}$).

$\boldsymbol{CoV}$ is defined as the ratio of the (square of the) standard deviation over the (square of the) mean of the distribution. When $\boldsymbol{CoV} > \mathbf{1}$, the associated distribution is deemed of high variability, as exponential distributions have $\boldsymbol{CoV} = \mathbf{1}$ and typically hyper-exponential distributions (which capture many high-variance, heavy tailed distributions) have $\boldsymbol{CoV} > \mathbf{1}$. In this case, HD performs cache eviction using PIN's scoring scheme; otherwise, it uses PINC's scoring scheme.

To this end, HD reuses the implementation of PIN and PINC pertaining to the two black boxes in Figure 4.15. **Step 2** directly leverages that of PINC, refreshing the two metrics of $\boldsymbol{CS_M}$ **Reduction** and $\bar{c}$ (recording the average estimated time cost per saved sub-iso test by each cached graph). On the other hand, **Step 11** rests on a hybrid paradigm to determine the utility score, that is:

- For all graphs in the current cache stores, their entries of $\boldsymbol{R}$ produce the value of $\boldsymbol{CoV}$.
- If the returned $\boldsymbol{CoV}$ is bigger than 1, HD goes to the implementation of PIN that uses formula (4.13) for scoring; otherwise, it switches to that of PINC, which employs formula (4.15) for the computation of utility score.

As shown in Figure 4.15, **Step 11** is repeated for each batch of queries. The decision processes among various batches are independent, echoing the dynamic essence of HD replacement policy.

Recall the example shown in Table 4.2. The mean $\boldsymbol{R}$ value is $\boldsymbol{\mu = 161}$ and its standard deviation $\boldsymbol{\sigma \approx 126}$; then $\boldsymbol{CoV = \sigma/\mu \approx 0.78 < 1}$ and thus HD will use PINC and evict queries 53 and 82.

4.3.4 Running in Parallel with Query Processing

A significant characteristic of GraphCache is that the cache management is designed to run in parallel with query processing. For this purpose, the implementation of Cache Manager is encapsulated in a thread handling the related operations.

Once the query execution of a query graph $\boldsymbol{g}$ finishes (i.e., returning the answer set against dataset graphs), GraphCache shall start a thread to deal with the cache management issues regarding $\boldsymbol{g}$. The said thread consists of three modules, each is responsible for performing a set of operations. In Overall, the processing inside each module is as follows.

- Update statistics store. Since the query processing of $\boldsymbol{g}$ benefits from cached graphs, the contribution pertaining to each relevant graph is recorded. Currently, GraphCache is designed to use the first batch queries to warm the cache. Hence, this module is turned off when query $\boldsymbol{g}$ is in the first batch, during which the cache is empty.

- Batch query in window stores. Unless there is isomorphic query in cache already or $\boldsymbol{g}$ fails the admission control, the query $\boldsymbol{g}$ is batched in window stores, being prepared to enter the cache and leverage benefits for future queries.

- Update cache and rebuild GraphCache index. Instead of refreshing cache contents per query graph, GraphCache updates the cache at the end of each batch, except the extreme case that none of queries in this batch had managed to enter the window stores. In this module, new queries in window are forwarded to the cache stores, where the cache replacement is usually incurred. Then the GraphCache index is rebuilt to serve future query processing, by identifying the subgraph/supergraph cases between the coming query and cached graphs (see §4.2 for detail).

As aforementioned, each executed query $\boldsymbol{g}$ evokes a thread going through the Cache Manager. In GraphCache, the main thread of system flow shall not be blocked this thread, unless $\boldsymbol{g}$ is the ending query of the batch. When query with ID $\boldsymbol{i}$ is being executed, there may exist several such threads pertaining to previous queries (with ID of $\boldsymbol{i-1}$, $\boldsymbol{i-2}$, ..., etc) chasing the same data stores within Cache Manager. This in turn results the following consequence:

- On the one hand, the efficiency of query processing is guaranteed, echoing the fundamental design concerns of GraphCache.

- On the other hand, locking mechanism is required to ensure the mutual exclusion for relevant operations. Calling the module for updating cache and rebuilding index is exclusively incurred when the query g ends a batch. Under such circumstance, the described thread shall block the system flow and hence hold the exclusive access for relevant data stores by nature. Therefore, the module for updating cache and rebuilding index does not need extra locking solutions. However, regarding the modules for updating statistics store and batching query in window stores, each has to launch a lock to ensure the mutual exclusion of the operations.

Furthermore, time overhead in GraphCache is generated for blocking the main thread of system flow, during which operations of updating cache contents and rebuilding index for cached graphs are performed. Quantified evaluation for such time overhead shall be detailed during the break-down analysis of query time in §6.

4.4 Summary

This chapter has presented GraphCache [95], to the best of our knowledge the first full-fledged caching system for general subgraph/supergraph query processing. In overall, the contribution of GraphCache system lies in that:

- The elegance afforded by the double use of GraphCache in accelerating both subgraph and supergraph query processing is unique.
- GraphCache can be used to expedite all current FTV and SI methods, bridging these two separate threads of research so far.
- Designed from ground up as a semantic graph cache, GraphCache manages to harness subgraph/supergraph cache hits, expanding the traditional exact-match-only hit.
- GraphCache is bundled with a number of GC exclusive graph-query-aware cache replacement policies and a novel cache admission control mechanism to enhance the performance gains.

Despite the large span, contents of this chapter are properly fit into three sections as follows, proceeding with a clear clue.

- By combining the general rationale of a caching system and the particularity of graph queries, it first identifies the design goals and builds the system architecture of GraphCache, which consists of two subsystems, namely the Query Processing Runtime and the Cache Manager.

- Regarding graph query processing, it has materialized iGQ framework by GraphCache components. Also, it monitors the statistics for purposes of both query performance evaluation and cache management support.

- As to the central section of cache management, it has demonstrated the stores in data layer, presented the window manager with novel mechanism of admission control, proposed a number of GraphCache exclusive cache replacement policies and “run” the subsystem of cache management in parallel with that of query processing, concluding a full-fledged caching system.

Chapter 5

Ensuring Consistency in Graph Cache for Graph-Pattern Queries

Following the established research in literature, the proposed GraphCache system handles graph queries against a static dataset. That is, throughout the continual arrival and execution of queries, all graphs in the underlying dataset remain unchanged. However, real-world applications indicate that the graph dataset could evolve over time. This poses a significant challenge for the current graph caching technique and hence emerges the requirement of advanced systems that could deal with dataset changes.

To address this problem, a straightforward and efficient solution is to employ the existing caching systems in literature, which in turn points to GraphCache. Therefore, this chapter shall present an upgraded graph caching system coined GraphCache+, highlighting the newly plugged-in subsystems and components on top of the vanilla GraphCache.

Two GC+ exclusive cache models will be developed, positioning different designs of ensuring graph cache consistency. Furthermore, GraphCache+ places emphasis on its novel logics in pruning candidate set for subgraph/supergraph queries with dataset changes; each is accompanied by the formally proved correctness.

To the best of our knowledge, GraphCache+ is the first study that discusses the topic of graph cache consistency and provides a full-fledged system that manages to accelerate the general subgraph/supergraph query processing over dynamic dataset.

5.1 Exploring Graph Cache Consistency

5.1.1 Consistency in Caches

Consistency issues arise from the parallel operations on shared resources. When one alters the state of the said resource, it must assure that no other can harbor a misunderstanding about the freshness of relevant data [96].

Web applications are aware of this and having solutions to ensure the consistency of web cache. For example, on the booking system of National Rail Enquiries [97], it is convenient to buy train tickets from various providers. To compete for the market share, providers endeavour to render attractive services such as seat reservation. The current available seats are retrievable through the front-end enquiry system. Say, at some time, regarding one specific train from Glasgow to Edinburgh, among all the seats left, only one is by window that is viewed and then clicked by 100 customers simultaneously. If the problem pertaining to consistency were not solved, 100 people might be allocated the same seat on the same train.

A modern computer system is also bundled with mechanisms for cache consistency. An example is just at hand. As I type this thesis, the new version is stored in the main memory of computer, while the hard disk holds a version of stale data. When I click to save the file, the disk and the main memory shall contain the same contents, until I start to type again. In overall, cache consistency is the term given to the problem of assuring that the contents of the cache memory and those of main memory for all caches in a multiple cache system are either identical or under tight enough control that stale and current data are not confused with each other [96].

In the realm of graph caching system, however, little work had been done. The established research had been focused on graph query processing against static underlying dataset, including the most recent proposed GraphCache system.

But among real-world applications, there are an abundance of situations bearing dynamic graph datasets. In social networking, the relations/interactions among people in a group could be easily modeled by one graph, where the edges bridge relevant nodes. And a graph dataset contains several graphs, e.g, interactions among fans of The Big Bang Theory [98] in Glasgow could form one graph and the global fans in turn develop a graph dataset (say, tagged with *FansOfTheBigBangTheory*). There are millions of such groups on applications of Facebook [90] and WeChat [99]. As time goes by, newly added groups, break-up of existed groups, and the changed relations/interactions among group members are frequently happening. Consider another example. In biochemical datasets, it is common that datasets are continually refreshed by newly-translated, disregarded or transformed proteins, for either application or research purpose. All in all, such changes could be modeled as graph addition

(ADD), graph deletion (DEL), graph update by edge addition (UA) and graph update by edge removal (UR) in graph dataset analytics.

Recall the current two research threads of graph query processing in literature. SI algorithms could accommodate these changes on the fly as each dataset graph shall undergo subgraph isomorphism test eventually, whereas FTV methods additionally require an updatable indexing mechanism to evict proper candidate set. To the best of our knowledge, however, none of the FTV algorithms proposed so far has been bundled with updatable index or similar solutions to tackle dataset changes.

As a result, it turns to SI methods, where each dataset graph is painstakingly verified for subgraph isomorphism. On the other hand, caching is proved efficient in accelerating the general subgraph/supergraph queries (see GraphCache system in §4). Therefore, it naturally follows an approach of using graph cache to alleviate the costly SI methods for subgraph/supergraph queries with dataset changes – a topic that has not been discussed yet. In the end, it is to create an upgraded system GraphCache+, which is capable of ensuring the consistency in graph cache amid the continual changes over the underlying dataset.

5.1.2 Designing GraphCache+

Like in GraphCache, a number of design goals are first identified for GraphCache+ system. Regarding the query workloads and graph datasets, GraphCache+ shares that of GraphCache (see details in §4.1.2). This section shall only present the particular issues in designing GraphCache+.

Dataset Change Plan

As mentioned, the underlying graph dataset could evolve over time. Hence, GraphCache+ system requires a change plan that could properly model the various mutations of graph dataset. Fundamentally, the generation of such dataset change plan must meet the following demands: First, it could well express the different categories of graph changes, such as {ADD, DEL, UA, UR}; Second, it should assure the flexibility that every mutation could occur at any time over any possible graph in the dataset. Much like the cache management in GraphCache system, GraphCache+ shall deal with the graph changes in batches, so as to guarantee the efficiency of query processing; more details shall be provided in §6.

Algorithmic Context

Similar as GraphCache, GraphCache+ is intended to be a general-purpose front-end for graph query processing, such that methods of both subgraph and supergraph queries could

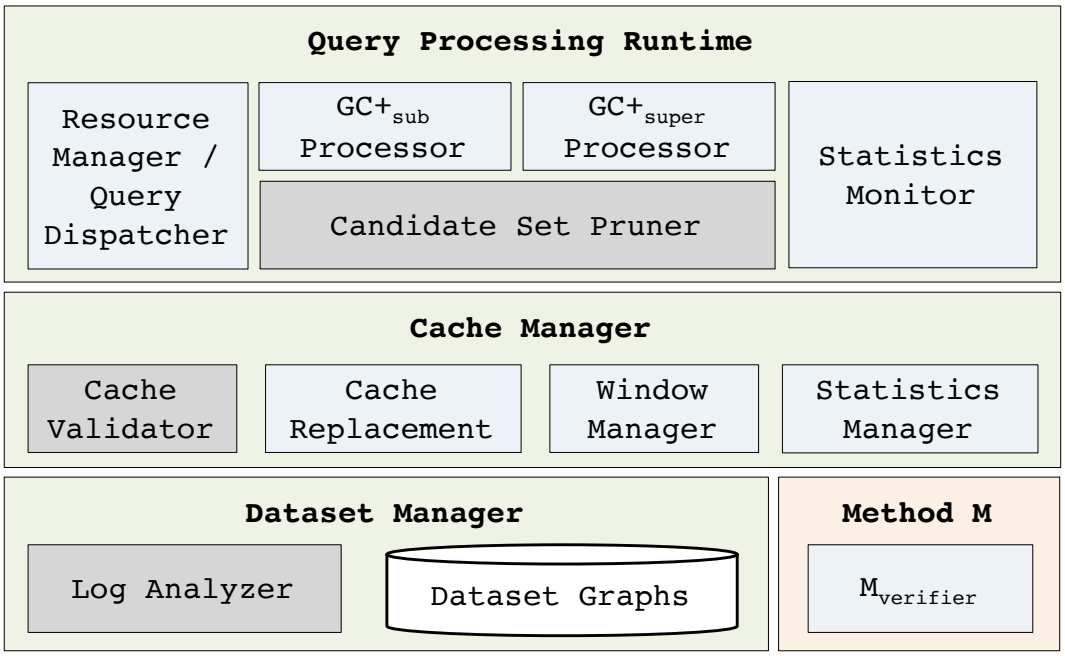


Figure 5.1: System Architecture of GraphCache+

be flexibly plugged in.

As aforementioned, over dynamic graph datasets, FTV methods require extra mechanisms to handle an updatable index. Unfortunately there is a lack of such FTV algorithms in literature. Whereas SI algorithms could accommodate the dataset changes on the fly, as each graph in the dataset shall undergo the subgraph isomorphism verification.

To this end, GraphCache+ is designed to be capable of supporting all the SI algorithms in literature. Again, GraphCache+ shall use three well-established SI methods with good performance, including GraphQL [17] as provided by [13], a modified version of VF2 [10] (denoted VF2+) provided by [18] and the plain VF2 [10] that is widely used in FTV implementations [16, 42, 13].

5.1.3 System Architecture of GraphCache+

Similar as GraphCache, GraphCache+ is a scalable semantic cache for subgraph/supergraph queries. GraphCache+ consists of four subsystems, namely Data Manager, Cache Manager, Query Processing Runtime and Method M, as shown by Figure 5.1. The first three are GraphCache+ internal and Method M is the external SI method that GraphCache+ is called to expedite. Method M subsystem includes an SI implementation, denoted $M_{verifier}$, which performs the subgraph isomorphism testing of the candidate set M_{CS} (the whole dataset when GraphCache+ is not used).

GC+ Specific Subsystems/Components

Dataset Manager subsystem is GraphCache+ exclusive, containing a component coined Log Analyzer to handle dataset logs. According to the specific designs of different cache models (see details in §5.2 and §5.3), Log Analyzer shall launch different operations of dealing with graph cache consistency.

Cache Manager is responsible for the management of data and metadata stored in the cache. Besides the components that work as usual as in GraphCache (such as Cache Replacement, Window Manager and Statistics Manager), the Cache Manager subsystem in GraphCache+ has an additional component named Cache Validator. The significant characteristic of GraphCache+, i.e., ensuring graph cache consistency, rests on the said components of Cache Validator and Log Analyzer; both pertain to mechanisms that are cache model dependent.

Query Processing Runtime subsystem of GraphCache+ is in charge of query execution and metrics monitoring. Like in GraphCache, it comprises a resource/thread manager, the internal subgraph/supergraph query processors, the logic for candidate set pruning, and a statistics monitor – these components of Query Processing Runtime communicate with Method M and the Cache Manager via well-defined APIs. Note that GraphCache+'s logic for Candidate Set Pruner is different and more complicated than that of GraphCache, though the former could be viewed as adapting from the latter. Details of GraphCache+ logics regarding subgraph/supergraph query processing shall be demonstrated in §5.3.3 and §5.3.4 shortly.

Data and Control Flow in GraphCache+

Figure 5.2 shows the flow of data and control in GraphCache+ system in processing a query.

- When a query g arrives, Dataset Manager subsystem first use the Log Analyzer component (1) to identify whether the dataset has been changed recently and send out the result of checking.
- If the result sent by Log Analyzer stating no recent change over the graph dataset, Cache Validator shall do nothing. Otherwise, providing the selected cache model, Cache Validator shall launch corresponding operations to reflect these changes to previous queries that are stored in cache and window (2).

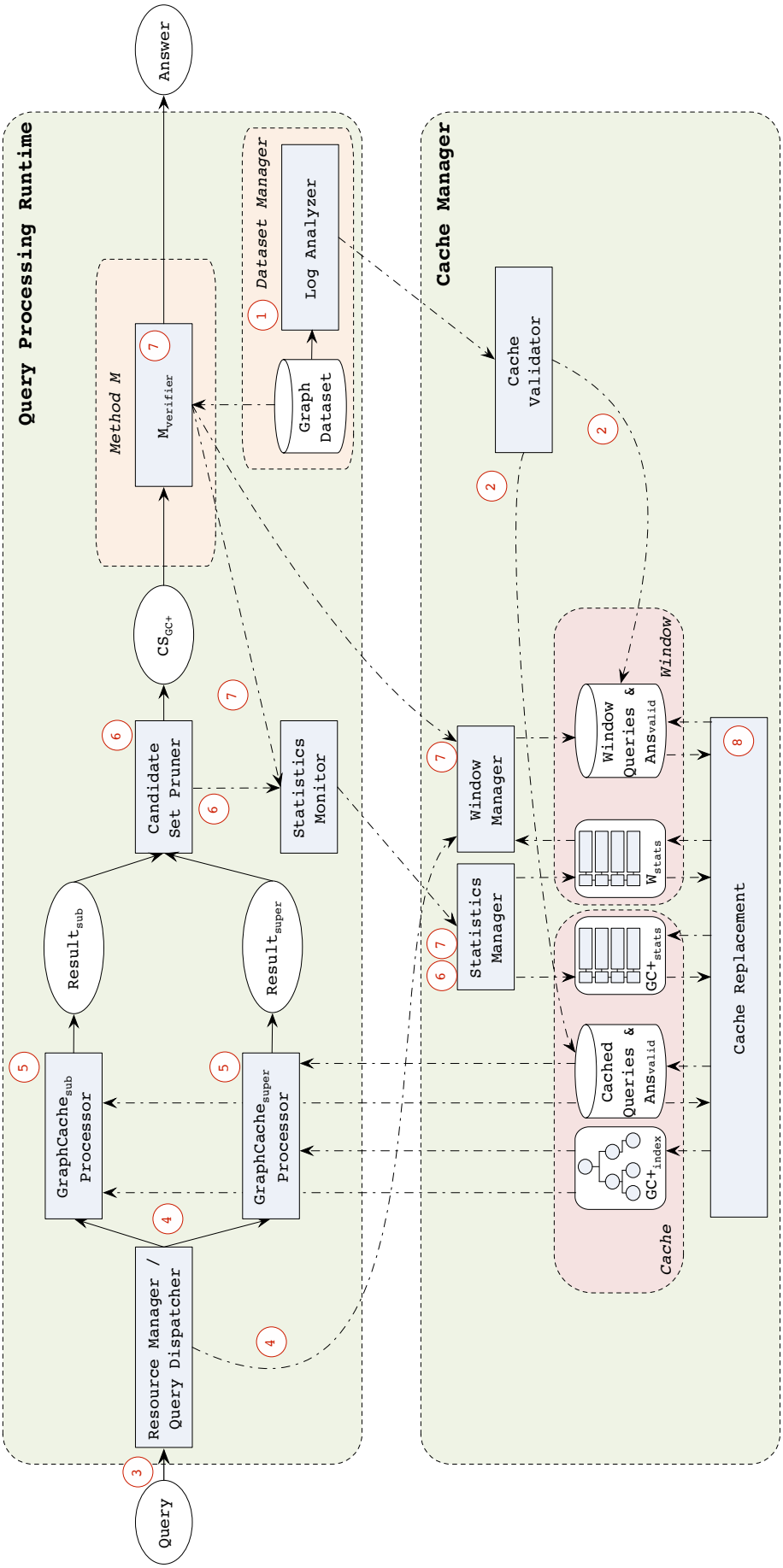


Figure 5.2: GraphCache+ System: The Data and Control Flow

- Then, g is sent to Query Processing Runtime subsystem for query execution. The query g arrives at the Resource Manager (3) and is dispatched to the processors GC+$_{sub}$ and GC+$_{super}$ in parallel (4) as per usual. Meanwhile, a copy of the query is added to the Window Manager. The time used by each processor, as well as their overall execution time, is gathered by the Statistics Monitor.

- The intermediate results of the processors GC+$_{sub}$ and GC+$_{super}$ (5) are forwarded to the Candidate Set Pruner (6) that generates the final *candidate set* GC+$_{CS}$, time of which is gathered by the Statistics Monitor. And statistics pertaining to GC+$_{CS}$ and the contribution of cached graphs are collected by the Statistics Monitor and fed to the Statistics Manager.

- Each dataset graph in GC+$_{CS}$ is then verified for subgraph isomorphism (7), resulting the *answer set* of query g. Metadata regarding the verification time is also gathered by the Statistics Monitor. And the resulted *answer set* is collected by the Statistics Monitor and sent to the Statistics Manager.

- When the Window Manager is full of currently executed queries, it shall perform cache admission control and forward admitted queries to window stores. Cache Manager subsystem shall then invoke cache replacement (8), concurrently with the Query Processing Runtime subsystem executing subsequent queries.

Note that GraphCache+ is capable of dealing with both subgraph and supergraph query processing. They share the above data and control flow, despite following different logics in reducing the candidate set (see §5.3.3 and §5.3.4 for detail).

5.2 Brute Force Approach: EVI Cache

To address the challenge arising from dynamic graph dataset, a straightforward solution is to abandon the vague cache, i.e., evicting (*EVI*) graph cache whenever dataset has encountered changes. Hence, operations of processing each query graph are figured out.

- **Step 1:** The Log Analyzer of Dataset Manager subsystem first checks whether the graph dataset has changed. If so, the Log Analyze shall raise a flag, broadcasting the message that there has been some changes over the graph dataset.

- **Step 2:** Knowing the dataset has changed, the Cache Validator then clears the contents of cache stores and window stores indiscriminately.

- **Step 3:** The query graph goes through the Query Processing Runtime subsystem for execution, with a cold graph cache that does not render any speedup.

- **Step 4:** The executed query, together with its metadata, is forwarded to the Cache Manager subsystem, preparing to facilitate future query processing.

In this way, the caching system GraphCache in §4 could be easily adapted to tackle graph queries with dataset changes, as the cleared cache will never produce error for future query processing. But the limitation is obvious – EVI cache has to warm from scratch upon every change over the graph dataset.

The problem of EVI cache lies in that it fails to differentiate the validity of stored answers pertaining to executed queries. For example, AIDS [86] dataset consists of 40K graphs; the current graph cache is filled with 100 executed queries, each having an *answer set* that reflects the containment relationship of the query g against dataset graphs $\{G\}$. Hence, the cache stores 4 million (40K $\times$ 100) units of information, where each unit is regarding a graph pair (g, G). Say, 100 dataset graphs undergo changes and some of the cache contents in turn become invalid. Indeed, invalid contents cannot be leveraged to accelerate future queries and should be abandoned. However, EVI abandons the whole cache, including those 3.99 million (39.9K $\times$ 100) information units that are still valid towards state of the art dataset. All in all, the purge in EVI cache throws away contents with good potential in expediting future query processing, making the cache efficiency truncated.

5.3 Advanced CON Cache

To address the aforementioned problem of EVI, this work contributes another cache model named *CON*, which targets at preserving useful information at full steam. Then the key issue turns to identifying the knowledge from a vague graph cache. Different from EVI that handles the graph cache as a whole, CON ensures consistency on a fine-grained level – to each unit of (g, G), i.e., per executed query against per dataset graph.

5.3.1 Interpreting the Rationale of CON

Problem Analysis

Before reaching GraphCache+, recall GraphCache system that handles graph queries against static dataset. For each executed subgraph query, its containment relationship with per dataset graph could be fully expressed by one bit, i.e., either 0 (the query is not a subgraph of the dataset graph) or 1 (the query is a subgraph of the dataset graph). Despite queries' continual arrival and execution, such state (0 or 1) is fixed and utilized by future query processing.

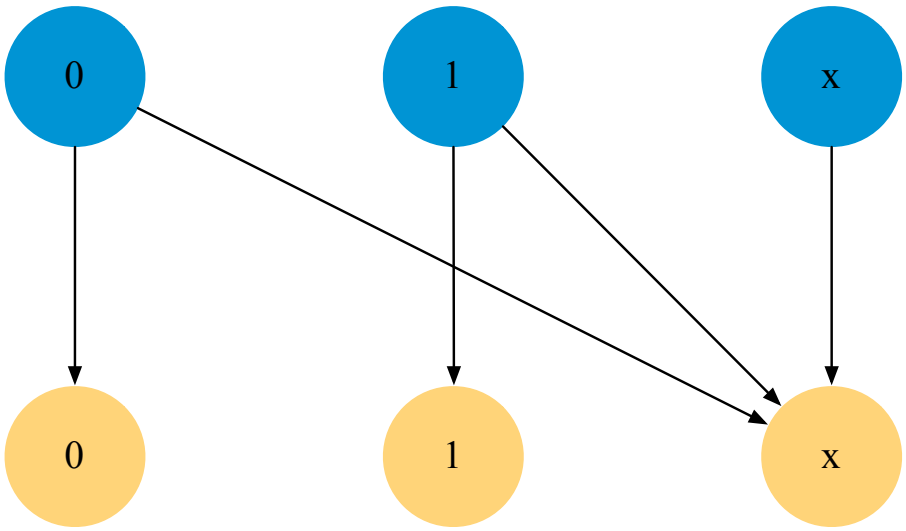


Figure 5.3: State Transitions of Containment Relationship in GraphCache+

Table 5.1: Mapping Each State to the Containment Relationship of an Executed Query g' versus a Dataset Graph G

query category of g'	state	description
subgraph query	0	$g' \nsubseteq G$
	1	$g' \subseteq G$
	x	unknown status
supergraph query	0	$g' \nsupseteq G$
	1	$g' \supseteq G$
	x	unknown status

Turing attention to GraphCache+. Things become more complicated. Since the underlying dataset is dynamic, the state that represents the query status over every dataset graph may undergo changes as well. More specially, state that once was 0 or 1 may become unknown, which is coined x in Figure 5.3. On the other hand, the state may remain as it was before. Such uncertainties exacerbate the complex situation of GraphCache+ system.

Nevertheless, Figure 5.3 manages to enumerate the state transitions that could take place in GraphCache+, in which a specific instance of each state is pertaining to an executed query and a dataset graph. For different categories of query processing, the states have different interpretations. Table 5.1 shows the meaning of such state values (i.e., 0, 1 or x) for subgraph queries and supergraph queries.

An Example of CON cache

CON cache model is developed to deal with the complicated cases of GraphCache+ system. We shall first use an example to illustrate how the CON model works for subgraph queries. With respect to supergraph queries, the CON model follows the similar principles and shall be detailed shortly in §5.3.4.

Figure 5.4 depicts an example when GraphCache+ is processing subgraph queries. GraphCache+ starts off with a dataset containing four graphs $\{G_0, G_1, G_2, G_3\}$ and an empty CON cache. At time T_1, query g' arrived at GraphCache+ system and was executed. Assuming that g' passed the admission control and entered the cache later. In turn, CON cache recorded the containment relationship of g' with each dataset graph, as $g' \nsubseteq G_0$, $g' \nsubseteq G_1$, $g' \subseteq G_2$ and $g' \subseteq G_3$.

Think of the dynamic graph dataset: mutations as to the mentioned containment relationships may occur. Therefore, CON cache employs an extra structure coined GC_{valid} for each executed query, storing the validity of its status with each dataset graph. GC_{valid} employs a BitSet structure and swiftly implements the aforementioned state x by assigning the relevant bit to 0. At time T_1, it naturally follows that $GC_{valid}(g')$ covers all graphs in current dataset, i.e., with IDs of 0, 1, 2, 3.

Then, at time T_2, dataset was changed by adding a new graph G_4, and an update on G_3 of edge removals. Obviously, there is no clue of G_4 containing g' or not, i.e., g' does not hold validity of its status with the newly coming G_4. As to G_3, there was $g' \subseteq G_3$, which becomes unknown as removing edge may result $g' \nsubseteq G_3$. Hence, the validity of g' pertaining to G_3 is turned off.

Subsequently, at time T_3, another query g'' was executed and later entered cache, holding the validity towards each graph in state-of-the-art dataset that contains five graphs. Again, the dataset changed at time T_4 – graph G_0 was deleted and G_1 was updated by edge additions. As to the current dataset $\{G_1, G_2, G_3, G_4\}$, $g' \nsubseteq G_1$ is not guaranteed, since adding edges may introduce $g' \subseteq G_1$. g', as well as g'', thus loses the validity over G_1.

Therefore, when the new query g comes, it would be facilitated by cached graphs g' and g'', each with the up-to-date valid info pertaining to all current dataset graphs, ensuring the cache consistency of CON model.

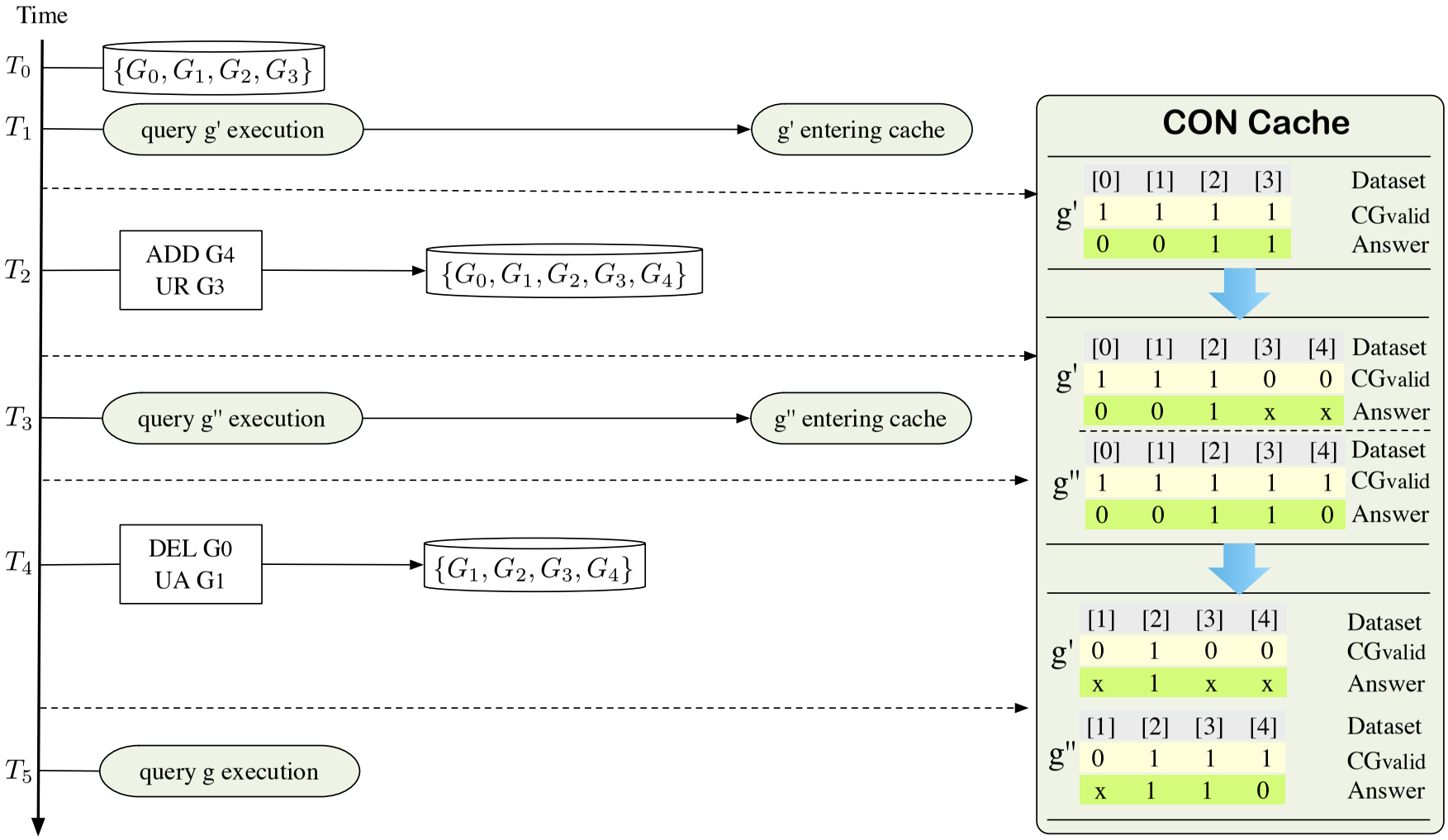


Figure 5.4: CON Cache Model: An Example with Timeline

Algorithm 6 Analyzing Log for the CON Cache

```
1: Input: Dataset update log L
2: Output: A container C with counters to categorize operations performed on dataset
   graphs
3:
4: Initialize C with an empty HASHMAP per counter (C_T, C_A and C_R)
5: Extract the incremental records ℝ from L
6: for all r ∈ ℝ do
7:     i = id of the dataset graph G in r
8:     t = operation type in r
9:     switch t do
10:        case UA
11:            C_A.get(i) += 1
12:            break
13:        case UR
14:            C_R.get(i) += 1
15:            break
16:    C_T.get(i) += 1
17: end for
18: return C
```

5.3.2 Algorithms and Structures

GraphCache+ is designed to warrant CON cache possessing the potential to benefit queries at full steam. To this end, both Dataset Manager subsystem and Cache Manager subsystem are accompanied with CON specific mechanisms.

Analyzing Dataset Log

Dataset Manager subsystem employs the component Log Analyzer to categorize dataset changes, acting as a preprocessing step for validating CON cache. Algorithm 6 illustrates how the corresponding analysis is performed.

Briefly, the incremental records that have not been reflected in cache are first extracted from dataset log. Log Analyzer then launches a container with three counters, implemented by HashMap with key of dataset graph id and value of count for the operations on this graph. The said three counters (C_T, C_A and C_R) (line 4) are responsible for counting the total, UA and UR operations, respectively. Each aforementioned record identifies the related dataset graph and its operation type (lines 7–8). Exhausting these records (lines 9–16) hence results the total counter C_T, UA counter C_A and UR counter C_R.

Algorithm 7 Refreshing a cached graph's validity indicator

1: Input: Counter container C (containing C_T, C_A and C_R), currently maximum graph id m in dataset, stored info of a cached graph (with its dataset-graph-validity-indicator CG_{valid} and query result $Answer$, both structured by BITSET)
2: Function: Updating CG_{valid}
3:
4: **if** $(m+1) > CG_{valid}.size$ **then**
5: Extend CG_{valid} to length $(m+1)$ by assigning false to extended bits
6: **end if**
7: **for all** $i \in C_T.keySet()$ **do**
8: $t_c = C_T.get(i)$
9: $ua_c = !C_A.containsKey(i)? \ 0 : \ C_A.get(i)$
10: $ur_c = !C_R.containsKey(i)? \ 0 : \ C_R.get(i)$
11:
12: **if** $t_c == ua_c \ \wedge \ CG_{valid}.get(i) \ \wedge \ Answer.get(i)$ **then**
13: continue
14: **else if** $t_c == ur_c \ \wedge \ CG_{valid}.get(i) \wedge !Answer.get(i)$ **then**
15: continue
16: **else**
17: $CG_{valid}.set(i, false)$
18: **end if**
19: **end for**

Validating CON Cache

The counter container returned by Dataset Manager subsystem is forwarded to Cache Manager subsystem, where Cache Validator refreshes the dataset-graph-validity-indicator for cached graphs. CON cache is designed at startup to augment the performance of graph query processing at full steam. The Cache Validator hence strives to exploit useful previous query results for CON.

In GC+, once a query is executed, its *answer set* is finalized, which snapshots the query status over dataset at the execution time – even the dataset would undergo changes later, GC+ will not repeat processing previous queries. Therefore, to deal with dataset changes, GC+ employs a BitSet indicator CG_{valid} per cached query, with each bit identifying the up-to-date validity of the query status towards a dataset graph.

Algorithm 7 depicts how the CG_{valid} of a cached graph g is refreshed by Cache Validator. To start with, CG_{valid} is checked whether it contains all the bits required (line 4 where dataset graph id starts from 0). If not, it implies that there are newly-added dataset graphs. Obviously, the status of g over those new dataset graphs is unknown and those extended bits are thus assigned false (line 5). The idea is to make recent dataset changes take effect on relevant bits of CG_{valid} (lines 7–19).

Specifically, for each concerned dataset graph G_i (identified by i in line 7), its numbers

of total operations (t_c), UA (ua_c) and UR (ur_c) are retrieved from the input counters (lines 8–10). If operations on dataset graph G_i are exclusively of UA category ($t_c == ua_c$ in line 12), together with valid ($CG_{valid}.get(i)$ in line 12) query result $g \subseteq G_i$ ($Answer.get(i)$ in line 12), such validity still holds ($g \subseteq G_i$ is not bothered by adding edges to G_i). Similarly, if operations on dataset graph G_i are exclusively of UR category, the query result $g \nsubseteq G_i$ ($!Answer.get(i)$ in line 14) remains valid. Whereas other operations shall make false the value of g on G_i if applicable (line 17).

By harnessing the validity to the level of per cached query and dataset graph, as well as the optimizations in UA-exclusive and UR-exclusive cases, CON model manages to enhance the cache efficiency in expediting graph queries.

5.3.3 CON Expediting Subgraph Query Processing

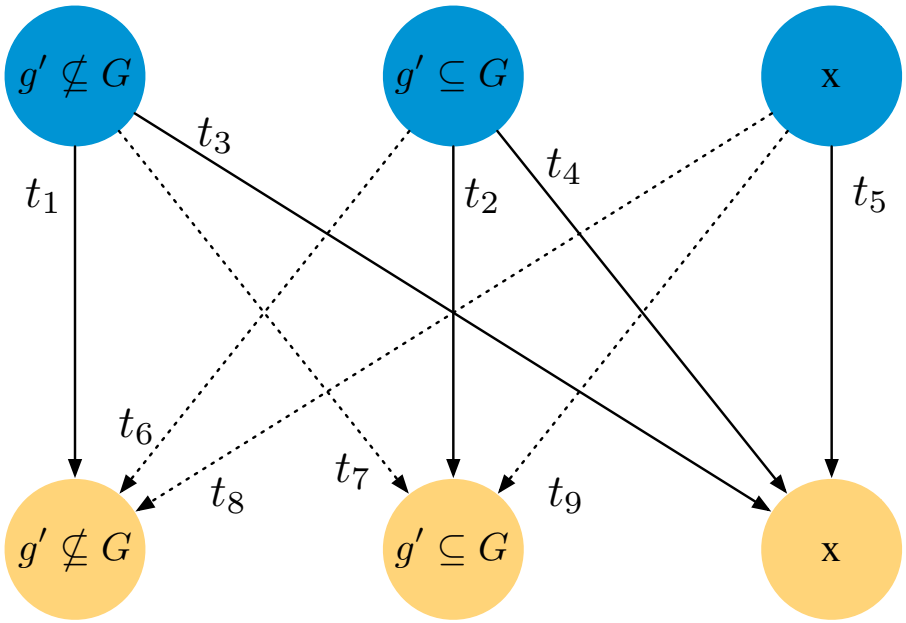


Figure 5.5: State Transitions in GraphCache+ for Subgraph Query Processing

Figure 5.5 uses solid arrows to show the transitions that may take place in GraphCache+ when processing subgraph queries. Each transition is triggered by certain circumstances.

- t_1: $g' \nsubseteq G$ still holds. This happens either since the dataset graph G has not been changed, or G is updated exclusively by edge removals (UR).
- t_2: $g' \subseteq G$ still holds, either since the dataset graph G has not been changed, or G is updated exclusively by edge additions (UA).
- t_3: The previous containment relationship $g' \nsubseteq G$ becomes unknown. This could be attributed to a number of reasons; nevertheless, they can be concluded as the com-

plementary setting of that for t_1, i.e., G has some updates that do not satisfy the requirement of t_1.

- t_4**:** The former status $g' \subseteq G$ becomes unknown. There exists a number of possible causes, which could be covered by the complementary scenarios of that for t_2, i.e., G has some updates that do not meet the requirement of t_2.

- t_5**:** The subgraph status of g' against G keeps unknown, due to the fact that GraphCache+ shall not launch further subgraph isomorphism tests to verify the uncertain status after the execution of g'.

To be complete, Figure 5.5 also lists transitions that are not of interest in GraphCache+ by dotted arrows. For example, t_6 could possibly occur, say, when the dataset graph G is updated by edge removals, the previous $g' \subseteq G$ may turn to $g' \nsubseteq G$, after being verified by subgraph isomorphism tests if they were to perform. However, as mentioned, GraphCache+ dedicates in utilizing previous queries to expedite future query processing and will not repeat the execution of issued queries, e.g., g'. Therefore, the containment relationship of g' and G is vague, i.e., t_6 is covered by t_4. Similarly, t_7 is included in t_3 and there exist no transitions of t_8 or t_9 in GraphCache+ either.

Think from another perspective. If the said subgraph isomorphism tests were indeed to be performed, there will not exist unknown state x at all. In that case, a single dataset graph mutation would make all the executed queries repeat query processing, which is totally out of the question.

The following contents of this section shall illustrate the specific logic of Candidate Set Pruner in GraphCache+ system, when processing subgraph queries. When a subgraph query g arrives, GraphCache+ discovers whether g is a subgraph or supergraph of cached queries by processors GC+$_{sub}$/GC+$_{super}$ in parallel, i.e., the subgraph case and the supergraph case, respectively.

Subgraph Case

For example, in Figure 5.6, the new query g comes with candidate set $CS_M(g)$(the current dataset) containing four graphs $\{G_1, G_2, G_3, G_4\}$. [1] [2] GC+$_{sub}$ Processor finds that there exists a previous query g', such that $g \subseteq g'$. Then g''s cached answer set $\{G_2, G_3\}$, as

[1]Note that examples in Figure 5.6 and in Figure 5.4 represent different perspectives. The former is from the view of graph dataset mutation categories, while the latter highlights the operations by which GraphCache+ reduces the candidate set and in turn expedites a subgraph query.

[2]Without specific explanations, each notation effects locally by default, e.g., g' in Figure 5.6 does not refer to that in Figure 5.4.

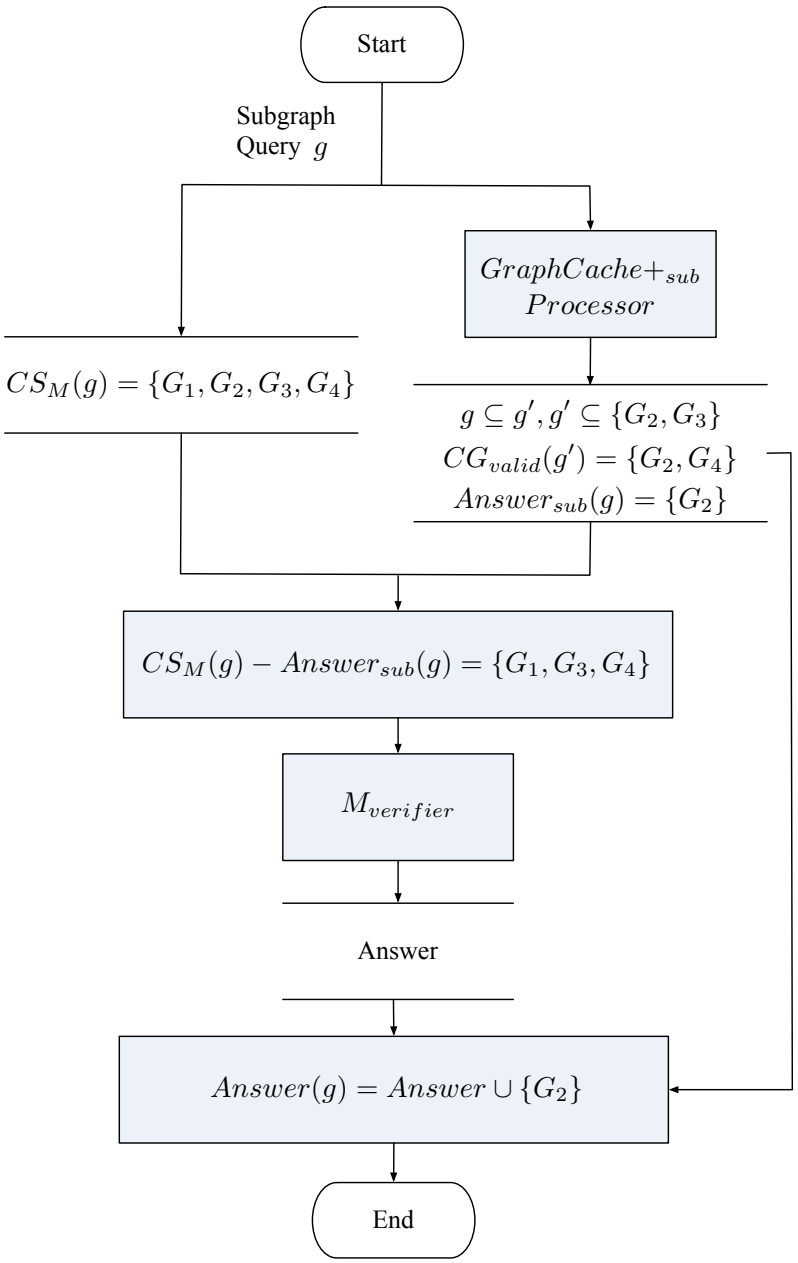


Figure 5.6: GraphCache+ Subgraph Case when g is a Q_{sub}

well as its up-to-time validity indicator $\boldsymbol{CG_{valid} = \{G_2, G_4\}}$, is retrieved. [3] Consider each graph in $\boldsymbol{CS_M(g)}$:

- For graph $\boldsymbol{G_2} \in \boldsymbol{CS_M(g)}$, providing $\boldsymbol{g \subseteq g'}$ and $\boldsymbol{g' \subseteq G_2}$ that still holds for current dataset since $\boldsymbol{G_2}$ exists in $\boldsymbol{CG_{valid}(g')}$, it must follow that $\boldsymbol{g \subseteq G_2}$. Hence, $\boldsymbol{G_2}$ can be safely removed from $\boldsymbol{CS_M(g)}$ and added directly to the final answer set of $\boldsymbol{g}$.
- Whereas $\boldsymbol{G_3}$ is not free of sub-iso testing though it appears in $\boldsymbol{g'}$'s cached answer set, as $\boldsymbol{g' \subseteq G_3}$ fails to hold over state-of-the-art dataset, i.e., $\boldsymbol{G_3}$ does not appear in

[3] As mentioned in Algorithm 7, both $\boldsymbol{Answer(g')}$ and $\boldsymbol{CG_{valid}(g')}$ are BitSet structures; here, we employ a set containing the id of dataset graph to help illustrate.

$CG_{valid}(g')$. That is, $g' \subseteq G_3$ was found when executing previous query g' but has been faded by subsequent dataset changes (e.g., G_3 was updated by removing some edges). Similarly, G_1 is not free of sub-iso testing either, for being absent in the validity indicator $CG_{valid}(g')$.

- Turning attention to G_4. Though $g' \subseteq G_4$ is still valid, there is no clue of the containment relationship between g and G_4, as both $g \subseteq G_4$ and $g \nsubseteq G_4$ could possibly occur.

Interestingly, the aforementioned operations of the four dataset graphs well represent the state transitions that could take place in subgraph query processing (see Figure 5.5). Table 5.2 shows such transitions pertaining to the subgraph status of each graph pair.

Table 5.2: State Transition Analysis: the Subgraph Case when g' is a Q_{sub}

graph pair	transition	start state	end state
(g', G_1)	t_3	0	x
(g', G_2)	t_2	1	1
(g', G_3)	t_4	1	x
(g', G_4)	t_1	0	0

Therefore, the logic of GC+ for subgraph case is that only dataset graphs in $CG_{valid}(g') \cap Answer(g')$ are sub-iso test-free, which can be safely removed from $CS_M(g)$ and directly added to the final answer set of query g. In the general case, g may be a subgraph of multiple previous query graphs g'_i. Then, the said sub-iso test-free graphs $Answer_{sub}(g)$ is given by:

$$Answer_{sub}(g) = \bigcup_{g'_i \in Result_{sub}(g)} CG_{valid}(g'_i) \cap Answer(g'_i) \tag{5.1}$$

where $\text{Result}_{sub}(g)$ contains all the currently cached queries of which g is a subgraph.

Hence, the set of dataset graphs for sub-iso testing is:

$$CS_{GC+_{sub}}(g) = CS_M(g) \setminus Answer_{sub}(g) \tag{5.2}$$

Finally, if $Answer_{GC+_{sub}}(g)$ is the set of graphs verified to be containing g through sub-iso tests on $CS_{GC+_{sub}}(g)$, the final answer set for query g will be:

$$Answer(g) = Answer_{GC+_{sub}}(g) \cup Answer_{sub}(g) \tag{5.3}$$

Lemma 11. *For subgraph queries, the final answer of GraphCache+ in the subgraph case does not contain false positives.*

Proof. Assume false positives are possible and consider the first ever false positive produced by GraphCache+; i.e., for some query g, $\exists G_{FP}$ such that $g \nsubseteq G_{FP}$ and $G_{FP} \in Answer(g)$. Note that G_{FP} cannot be in $Answer_{GC+_{sub}}(g)$ where each graph has passed the sub-iso test, which follows that $G_{FP} \in Answer_{sub}(g)$ by formula (5.3). Furthermore, formula (5.1) implies $\exists g'$ such that $g \subseteq g'$, $G_{FP} \in CG_{valid}(g')$ and $G_{FP} \in Answer(g')$. Hence, $G_{FP} \in Answer(g')$ is valid for up-to-date dataset, i.e., $g' \subseteq G_{FP}$. But $g' \subseteq G_{FP}$ and $g \subseteq g' \Rightarrow g \subseteq G_{FP}$ (a contradiction). □

Lemma 12. *For subgraph queries, the final answer of GraphCache+ in the subgraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by GraphCache+ particularly; i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. As $G_{FN} \in CS_M(g)$ in GraphCache+ by default and sub-iso testing does not introduce any false negative, the only possibility for error is that G_{FN} was removed using formula (5.2); i.e., $G_{FN} \notin CS_{GC+_{sub}}(g)$. That implies that $\exists g'$ such that $g \subseteq g'$ and $G_{FN} \in Answer(g')$. But then, by formula (5.3), G_{FN} will be added to $Answer_{sub}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). □

Theorem 6. *For subgraph queries, the final answer of GraphCache+ in the subgraph case is correct.*

Proof. There are only two possibilities for error; GraphCache+ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 11 and 12. □

Supergraph Case

Figure 5.7 depicts an example of supergraph case in GraphCache+, where GC+$_{super}$ Processor identifies there exists a previous query g'' satisfying $g'' \subseteq g$. For g'', the cached answer set $\{G_2, G_3\}$ and the dataset-graph-validity indicator $CG_{valid}(g'')$ ($\{G_2, G_4\}$) are retrieved. Again, the new query g comes with candidate set $CS_M(g)$ ($\{G_1, G_2, G_3, G_4\}$).

Now, consider each dataset graph in $CS_M(g)$.

- With regard to graph $G_1 \in CS_M(g)$: G_1 does not hold validity of its status over g''. Hence, no previous query result about G_1 could be utilized by g; G_1 will have to undergo sub-iso testing. For the same reason, G_3 is not free of sub-iso test either.
- For graph $G_2 \in CS_M(g)$: $g'' \subseteq G_2$ holds, which does not remove G_2 from sub-iso test despite $g'' \subseteq g$, as the subgraph status of g against G_2 is still obscure and has to be verified by sub-iso test.

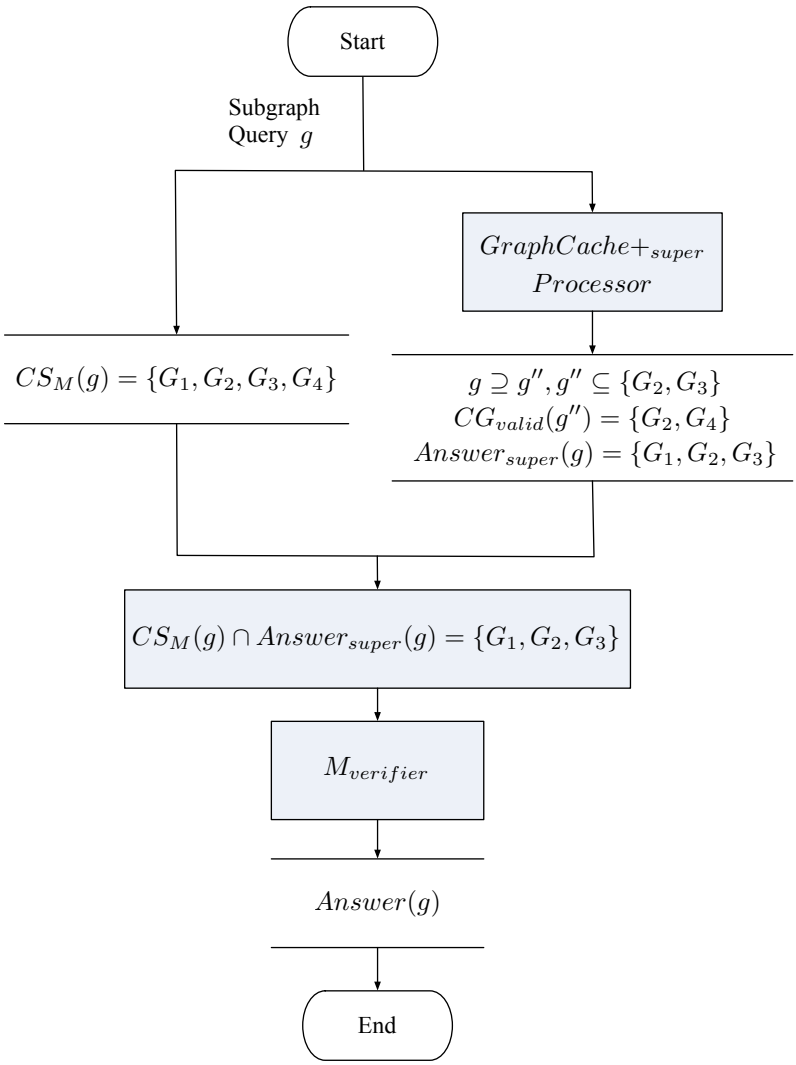


Figure 5.7: GraphCache+ Supergraph Case when g is a Q_{sub}

- As to graph $G_4 \in CS_M(g)$: G_4 holds validity for $g'' \nsubseteq G_4$. Since $g'' \subseteq g$, if $g \subseteq G_4$ were to be true, it should follow $g'' \subseteq G_4$, which is a contradiction. So it is safe to conclude that $g \nsubseteq G_4$ and thus G_4 can be removed from $CS_M(g)$.

Table 5.3 shows the transitions with respect to each dataset graph in Figure 5.7.

Table 5.3: State Transition Analysis: the Supergraph Case when g'' is a Q_{sub}

graph pair	transition	start state	end state
(g'', G_1)	t_3	0	x
(g'', G_2)	t_2	1	1
(g'', G_3)	t_4	1	x
(g'', G_4)	t_1	0	0

In overall, among graphs in $CS_M(g)$, those failing to appear in $(\overline{CG_{valid}(g'')} \cup Answer(g'')$

can never exist in the final answer set of g and thus become sub-iso test free, where $\overline{CG_{valid}(g'')}$ is the complementary set of $CG_{valid}(g'')$ against state-of-the-art dataset. In other words, the set $(\overline{CG_{valid}(g'')} \cup Answer(g'')$ covers all graphs that could possibly exist in the final answer set of g, denoted as $g''.Answer_{super}(g)$, i.e.,

$$g''.Answer_{super}(g) = (\overline{CG_{valid}(g'')} \cup Answer(g'') \tag{5.4}$$

Performing sub-iso tests on $CS_M(g) \cap g''.Answer_{super}(g)$ therefore results the verified query answer $Answer(g)$.

In the general case, g may be a supergraph of multiple previous query graphs g''_j. Then, the set of graphs tested for sub-iso by GC+ is:

$$CS_{GC+_{super}}(g) = CS_M(g) \cap \bigcap_{g''_j \in Result_{super}(g)} g''_j.Answer_{super}(g) \tag{5.5}$$

where Result$_{super}(g)$ contains all the currently cached queries of which g is a supergraph.

The final answer for query g, $Answer(g)$, will be the set of graphs in $CS_{GC+_{super}}(g)$ that pass the sub-iso test.

Lemma 13. *For subgraph queries, the final answer of GraphCache+ in the supergraph case does not contain false positives.*

Proof. This trivially follows by construction as all graphs in $Answer(g)$ have passed through subgraph isomorphism testing at the final stage of query processing. □

Lemma 14. *For subgraph queries, the final answer of GraphCache+ in the supergraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by GraphCache+; i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. Since $G_{FN} \in CS_M(g)$, the only possibility for error is that G_{FN} is removed from $CS_{GC+_{super}}(g)$ by formula (5.5). This implies that $\exists g''$ such that $G_{FN} \notin g''.Answer_{super}(g)$ and $g'' \subseteq g$. By formula (5.4), it turns to $G_{FN} \notin (\overline{CG_{valid}(g'')}$ and $G_{FN} \notin Answer(g'')$, with $G_{FN} \notin (\overline{CG_{valid}(g'')} \Rightarrow G_{FN} \in CG_{valid}(g'')$. Hence, for state-of-the-art dataset, $G_{FN} \notin Answer(g'')$ is valid, i.e., $g'' \nsubseteq G_{FN}$. But $g \subseteq G_{FN}$ and $g'' \subseteq g \Rightarrow g'' \subseteq G_{FN}$ (a contradiction). □

Theorem 7. *For subgraph queries, the final answer of GraphCache+ in the supergraph case is correct.*

Proof. There are only two possibilities for error; GraphCache+ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 13 and 14. □

Putting It All Together and Optimal Cases

The Query Processing Runtime subsystem first applies equation (5.2) on CS_M and then applies (5.5) on the result of the previous operation. The end result is a reduced candidate set, which is then sub-iso tested.

Additionally, there are two optimal cases that warrant further performance gains. First, note that GraphCache+ can easily recognize the case where a new query, g, is isomorphic to a previous cached query g'. For connected query graphs, this holds providing that (i) $g \subseteq g'$ or $g \supseteq g'$; and (ii) g and g' have the same number of nodes and edges; and (iii) g' holds validity on all the up-to-date dataset graphs. Hence, GraphCache+ can return the cached result of g' directly, rendering sub-iso test free.

Second, consider the case: for a new query g, a cached query g'' is discovered by GraphCache+ that $g'' \subseteq g$, $Answer(g'') = \emptyset$ and g'' holds validity on all graphs currently in dataset. Thus, GraphCache+ can directly return an empty result set for g. The idea is that if there were a dataset graph G such that $g \subseteq G$, since $g'' \subseteq g$ we would conclude that $g'' \subseteq G \Rightarrow G \in Answer(g'')$, contradicting the fact that $Answer(g'') = \emptyset$; thus, no such graph G can exist and the final result set of g is necessarily empty.

5.3.4 CON Expediting Supergraph Query Processing

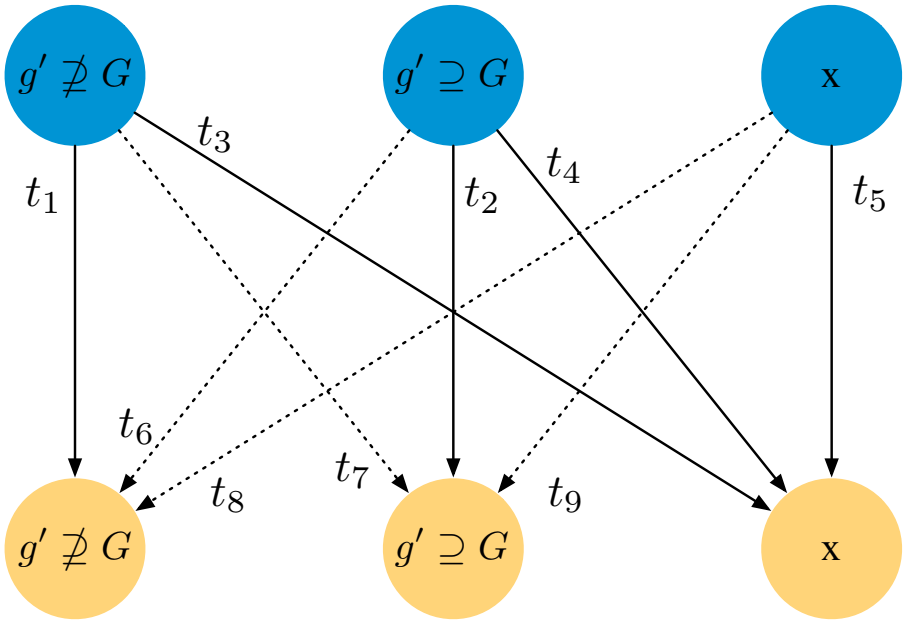


Figure 5.8: State Transitions in GraphCache+ for Supergraph Query Processing

GraphCache+ affords the elegance of double use for subgraph/supergraph query processing. Like in processing subgraph queries, Figure 5.8 enumerates the state transitions that could

occur regarding a graph pair (g', G), where g' is an executed query and G is a dataset graph (using solid arrows). In turn, dotted arrows show transitions that are not of interest in GraphCache+, just to be complete. For more details of such cases, please refer to §5.3.3.

With respect to the logics of reducing candidate set, again, processors GC+$_{sub}$/GC+$_{super}$ are responsible to identify whether the new supergraph query g is a subgraph or supergraph of previously executed queries in cache, namely subgraph/supergraph case as follows.

Subgraph Case

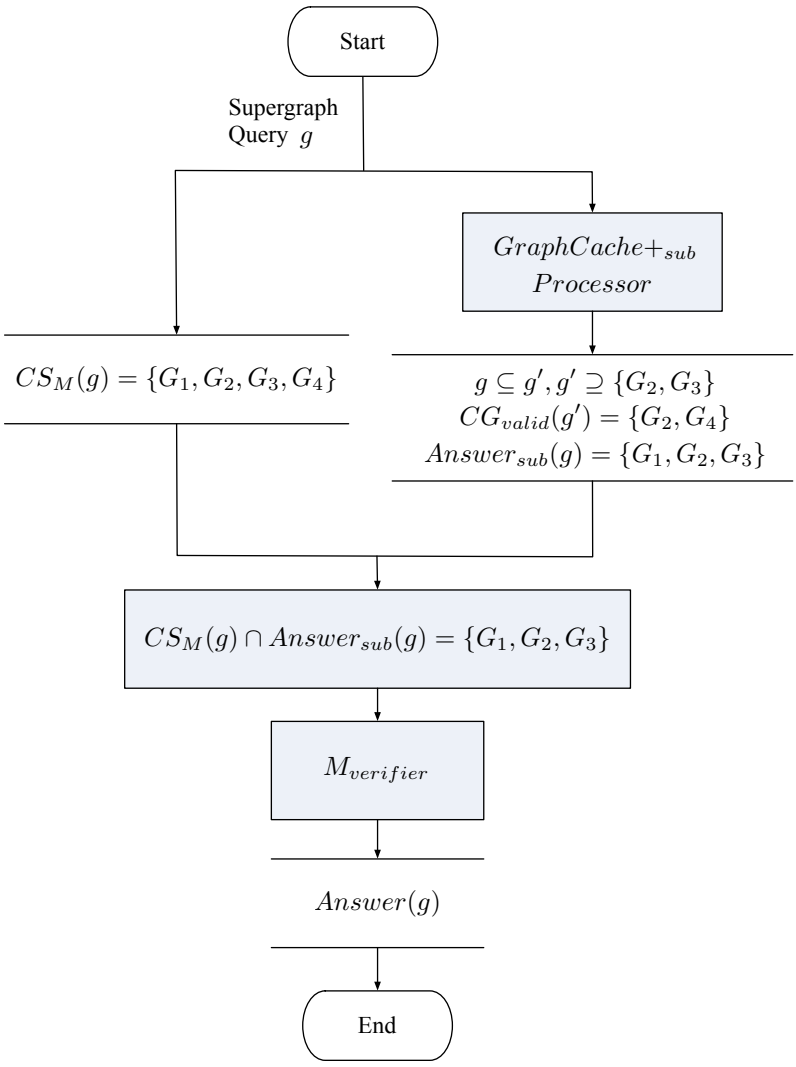


Figure 5.9: GraphCache+ Subgraph Case when g is a Q$_{super}$

Figure 5.9 shows an example of subgraph case when GraphCache+ handles a supergraph query g, during which GC+$_{sub}$ Processor identifies there exists a previous query g' such that $g \subseteq g'$. Pertaining to g', the cached answer set is $\{G_2, G_3\}$ and the corresponding validity of dataset graphs is $CG_{valid}(g')$, i.e., $\{G_2, G_4\}$. Again, the new query g comes with the candidate set $CS_M(g)$, i.e., $\{G_1, G_2, G_3, G_4\}$.

As per usual, we shall first look into each graph in $CS_M(g)$.

- Regarding graph G_1, its status with g' is not valid. Leveraging invalid previous query result to facilitate query processing is not permitted. Hence, G_1 has to undergo subgraph isomorphism test, so as to determine $g \supseteq G_1$ or $g \not\supseteq G_1$. Similarly, G_3 is not free of sub-iso test either.
- Consider graph G_2 that holds validity of g'. However, combining $g' \supseteq G_2$ and $g \subseteq g'$ does not deliver clear conclusion of $g \supseteq G_2$ or $g \not\supseteq G_2$, as both are possible. For this reason, G_2 shall be tested for subgraph isomorphism.
- As to graph G_4, it is valid that $g' \not\supseteq G_4$. Since $g \subseteq g'$, if $g \supseteq G_4$ were to be true, then it should follow $g' \supseteq G_4$, which is a contradiction. Hence, it is safe to result $g \not\supseteq G_4$ and G_4 can be safely removed from $CS_M(g)$.

Table 5.4 shows the state transitions regarding a previous query g' and each concerned dataset graph.

Table 5.4: State Transition Analysis: the Subgraph Case when g' is a Q_{super}

graph pair	transition	start state	end state
(g', G_1)	t_3	0	x
(g', G_2)	t_2	1	1
(g', G_3)	t_4	1	x
(g', G_4)	t_1	0	0

Therefore, for graphs in $CS_M(g)$, if they do not exist in $(\overline{CG_{valid}(g')} \cup Answer(g')$, these graphs shall not appear in the final answer set of query g for sure; hence no need to be tested for subgraph isomorphism. In turn, the set $(\overline{CG_{valid}(g')} \cup Answer(g')$ covers all the dataset graphs that could possibly exist in the final answer set of g, denoted as $g'.Answer_{sub}(g)$, i.e.,

$$g'.Answer_{sub}(g) = (\overline{CG_{valid}(g')} \cup Answer(g') \tag{5.6}$$

Performing sub-iso tests on $CS_M(g) \cap g'.Answer_{sub}(g)$ hence results the final query answer $Answer(g)$.

In the general case, g may be a subgraph of multiple previous supergraph queries g'_i. Then, the set of graphs tested for sub-iso by GraphCache+ is given by:

$$CS_{GC+_{sub}}(g) = CS_M(g) \cap \bigcap_{g'_i \in Result_{sub}(g)} g'_i.Answer_{sub}(g) \tag{5.7}$$

where Result$_{sub}(g)$ contains all the currently cached queries such that g is a subgraph.

The final answer for query g, $\boldsymbol{Answer}(g)$, will be the set of graphs in $\boldsymbol{CS}_{GC+_{sub}}(g)$ that pass the sub-iso test.

Lemma 15. *For supergraph queries, the final answer of GraphCache+ in the subgraph case does not contain false positives.*

Proof. This trivially follows by construction as all graphs in $\boldsymbol{Answer}(g)$ have passed through subgraph isomorphism testing at the final stage of query processing. □

Lemma 16. *For supergraph queries, the final answer of GraphCache+ in the subgraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by GraphCache+; i.e., for some query g, $\exists \boldsymbol{G_{FN}}$ such that $g \supseteq \boldsymbol{G_{FN}}$ and $\boldsymbol{G_{FN}} \notin \boldsymbol{Answer}(g)$. Since $\boldsymbol{G_{FN}} \in \boldsymbol{CS}_M(g)$, the only possibility for error is that $\boldsymbol{G_{FN}}$ is removed from $\boldsymbol{CS}_{GC+_{sub}}(g)$ by formula (5.7). This implies that $\exists g'$ such that $\boldsymbol{G_{FN}} \notin g'.\boldsymbol{Answer}_{sub}(g)$ and $g \subseteq g'$. By formula (5.6), it follows that $\boldsymbol{G_{FN}} \notin (\overline{\boldsymbol{CG}_{valid}(g')}$ and $\boldsymbol{G_{FN}} \notin \boldsymbol{Answer}(g')$, with $\boldsymbol{G_{FN}} \notin (\overline{\boldsymbol{CG}_{valid}(g')} \Rightarrow \boldsymbol{G_{FN}} \in \boldsymbol{CG}_{valid}(g')$. Hence, for the current graph dataset, $\boldsymbol{G_{FN}} \notin \boldsymbol{Answer}(g')$ is valid, i.e., $g' \not\supseteq \boldsymbol{G_{FN}}$. But $g \supseteq \boldsymbol{G_{FN}}$ and $g \subseteq g' \Rightarrow g' \supseteq \boldsymbol{G_{FN}}$ (a contradiction). □

Theorem 8. *For supergraph queries, the final answer of GraphCache+ in the subgraph case is correct.*

Proof. There are only two possibilities for error; GraphCache+ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 15 and 16. □

Supergraph Case

In turn, Figure 5.10 shows the supergraph query g coming with candidate set $\boldsymbol{CS}_M(g)$, i.e., $\{\boldsymbol{G_1}, \boldsymbol{G_2}, \boldsymbol{G_3}, \boldsymbol{G_4}\}$. GC+$_{super}$ Processor discovers that there exists a previously executed query g'', such that $g \supseteq g''$. Also, the cached answer of g'', i.e., $\{\boldsymbol{G_2}, \boldsymbol{G_3}\}$, and the state-of-the-art validity indicator $\boldsymbol{CG}_{valid} = \{\boldsymbol{G_2}, \boldsymbol{G_4}\}$ are available.

Turning attention to each dataset graph that appears in $\boldsymbol{CS}_M(g)$.

- First, consider graph $\boldsymbol{G_2}$. As $\boldsymbol{G_2}$ exists in $\boldsymbol{CG}_{valid}(g'')$, $g' \supseteq \boldsymbol{G_2}$ is valid for the current dataset. Together with $g \supseteq g''$, it must follow that $g \supseteq \boldsymbol{G_2}$. As a result, $\boldsymbol{G_2}$ can be safely removed from $\boldsymbol{CS}_M(g)$ and added directly to the final answer set of g.

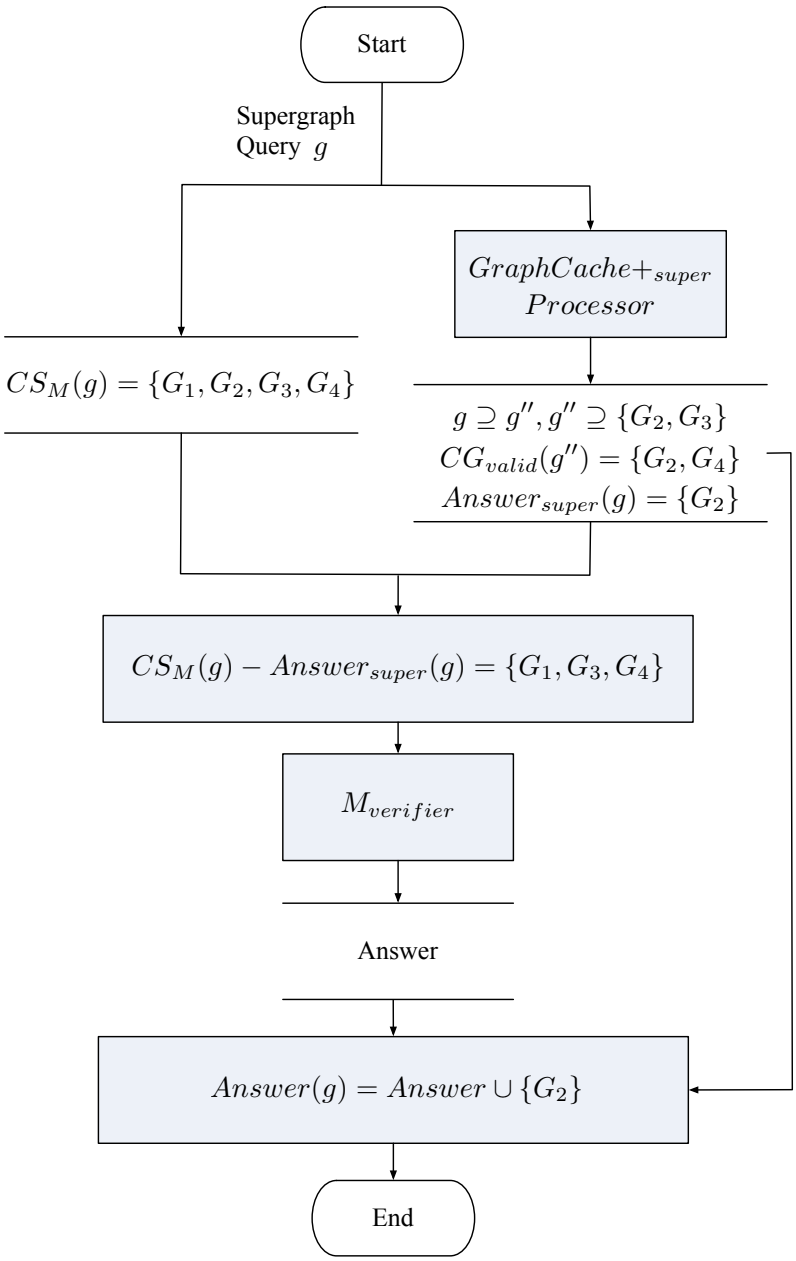


Figure 5.10: GraphCache+ Supergraph Case when g is a Q_{super}

- Then, it is the graph G_3 that also appears in the answer set of g''. Since G_3 does not appear in $CG_{valid}(g'')$, $g'' \supseteq G_3$ is invalid for state-of-the-art dataset. In other words, $g'' \supseteq G_3$ was found when executing previous query g'' but has been faded by subsequent dataset changes (e.g., G_3 was updated by adding some edges). As a result, G_3 will have to be verified by sub-iso test. Similarly, G_1 is not free of sub-iso testing either, for failing to hold validity in $CG_{valid}(g'')$.
- Consider graph G_4. As the valid $g'' \not\supseteq G_4$ and $g \supseteq g''$ does not lead to the supergraph status of g over G_4, G_4 will have to undergo subgraph isomorphism test.

As to the state transitions, Table 5.5 shows the details pertaining to a previous supergraph

Table 5.5: State Transition Analysis: the Supergraph Case when g'' is a Q_{super}

graph pair	transition	start state	end state
(g'', G_1)	t_3	0	x
(g'', G_2)	t_2	1	1
(g'', G_3)	t_4	1	x
(g'', G_4)	t_1	0	0

query g'' and every dataset graph.

In overall, the logic of GraphCache+ handling supergraph queries in the supergraph case is that only dataset graphs in $CG_{valid}(g'') \cap Answer(g'')$ are free of sub-iso tests. The said dataset graphs can be safely removed from $CS_M(g)$ and directly added to the final answer set of query g. In the general case, g may be a supergraph of multiple previous query graphs g''_j. Then, the aforementioned graphs that are free of sub-iso tests, $Answer_{super}(g)$, is given by:

$$Answer_{super}(g) = \bigcup_{g''_j \in Result_{super}(g)} CG_{valid}(g''_j) \cap Answer(g''_j) \tag{5.8}$$

where $\text{Result}_{super}(g)$ contains all the currently cached queries of which g is a supergraph.

In turn, the set of dataset graphs for sub-iso testing is given by:

$$CS_{GC+_{super}}(g) = CS_M(g) \setminus Answer_{super}(g) \tag{5.9}$$

Finally, if $Answer_{GC+_{super}}(g)$ is the set of graphs verified to be contained in g through sub-iso tests among $CS_{GC+_{super}}(g)$, the final answer set for supergraph query g will be:

$$Answer(g) = Answer_{GC+_{super}}(g) \cup Answer_{super}(g) \tag{5.10}$$

Lemma 17. *For supergraph queries, the final answer of GraphCache+ in the supergraph case does not contain false positives.*

Proof. Assume false positives are possible and consider the first ever false positive produced by GraphCache+; i.e., for some query g, $\exists G_{FP}$ such that $g \not\supseteq G_{FP}$ and $G_{FP} \in Answer(g)$. Note that G_{FP} cannot be in $Answer_{GC+_{super}}(g)$ where each graph has passed the sub-iso test, which follows that $G_{FP} \in Answer_{super}(g)$ by formula (5.10). Furthermore, formula (5.8) implies $\exists g''$ such that $g \supseteq g''$, $G_{FP} \in CG_{valid}(g'')$ and $G_{FP} \in Answer(g'')$. Hence, $G_{FP} \in Answer(g'')$ is valid for the current dataset, i.e., $g'' \supseteq G_{FP}$. But $g'' \supseteq G_{FP}$ and $g \supseteq g'' \Rightarrow g \supseteq G_{FP}$ (a contradiction). □

Lemma 18. *For supergraph queries, the final answer of GraphCache+ in the supergraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by GraphCache+ particularly; i.e., for some query g, $\exists G_{FN}$ such that $g \supseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. As $G_{FN} \in CS_M(g)$ in GraphCache+ by default and sub-iso testing does not introduce any false negative, the only possibility for error is that G_{FN} was removed using formula (5.9); i.e., $G_{FN} \notin CS_{GC+_{super}}(g)$. That implies that $\exists g''$ such that $g \supseteq g''$, $G_{FN} \in Answer(g')$ and $G_{FN} \in CG_{valid}(g'')$. But then, by formula (5.10), G_{FN} will be added to $Answer_{super}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). □

Theorem 9. *For supergraph queries, the final answer of GraphCache+ in the supergraph case is correct.*

Proof. There are only two possibilities for error; GraphCache+ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 17 and 18. □

Putting It All Together and Optimal Cases

In handling supergraph queries, the Query Processing Runtime subsystem of GraphCache+ shall first apply equation (5.9) on CS_M and then apply (5.7) on the result of the previous operation. In the end, it results a reduced candidate set, which is then sub-iso tested. For supergraph query processing, there are two optimal cases that bear further performance gains as well.

First, the optimal case of isomorphic query still exists. That is, GraphCache+ can easily detect the case where a new query, g, is isomorphic to a previous cached query g'. For connected query graphs, this holds providing that (i) $g \subseteq g'$ or $g \supseteq g'$; and (ii) g and g' have the same number of nodes and edges; and (iii) g' holds validity on all the up-to-date dataset graphs. Hence, GraphCache+ can return the cached result of g' directly, without performing any sub-iso test.

Second, consider the case: for a new query g, a cached query g'' is discovered by GraphCache+ that $g \subseteq g''$, $Answer(g'') = \emptyset$ and g'' holds validity on all graphs currently in dataset. Thus, GraphCache+ can directly return an empty answer set for g. The reasoning behind is that if there were a dataset graph G such that $g \supseteq G$, since $g \subseteq g''$ one can conclude that $g'' \supseteq G \Rightarrow G \in Answer(g'')$, contradicting the fact that $Answer(g'') = \emptyset$; therefore, no such graph G can exist and the final result set of g is necessarily empty.

5.4 Summary

Underpinned by the work in §4, this chapter has presented an upgraded system GraphCache+. To the best of our knowledge, GraphCache+ [100] is the first study pertaining to ensuring graph cache consistency for graph structured queries. GraphCache+ is featured by two exclusive cache models, namely EVI and CON, which reflect different designs in dealing with the consistency of graph cache. EVI could swiftly adapt from the existing GraphCache system whereas CON claims advanced requirements. Regarding the CON cache, GraphCache+ affords the logics of reducing the candidate set for subgraph/supergraph queries (with formally proved correctness), the central algorithms and structures.

Furthermore, to address the problem stemming from the dynamic graph dataset, GraphCache+ takes a swift manner by plugging in new subsystems and components to GraphCache, instead of starting from scratch. Such good practice of well utilizing the existing resources to tackle new challenges is noteworthy.

Chapter 6

Performance Evaluation

As designed at startup, GraphCache/GraphCache+ can be used as a front end, complementing state of the art graph query processing method as a pluggable component. Currently, GraphCache comes bundled with three top-performing filter-then-verify (FTV) subgraph query methods and three well-established direct subgraph-isomorphism (SI) algorithms – representing different categories of graph query processing research.

Recall the particular circumstance of graph queries processed over a dynamic dataset, in which FTV methods require extra mechanisms to deal with an updatable index. To the best of our knowledge, however, none of the FTV algorithms in literature is equipped with updatable index or similar solutions to tackle dataset changes. Whereas SI algorithms could accommodate such dataset changes on the fly, as each graph in the dataset shall undergo the subgraph isomorphism verification. Therefore, GraphCache+ is intended to complement SI algorithms for the time being and the current GraphCache+ is accompanied with three well-established SI methods.

This chapter shall contribute a comprehensive performance evaluation of GraphCache/GraphCache+. As to the concern of any caching system that a large number of queries are required so as to obtain reliable results, more than 6 million queries shall be generated using different workload generators, and executed against both real-world and synthetic graph datasets of different characteristics. Extensive experiments shall be carried out in a number of dimensions, such as different graph datasets, query workloads, algorithm contexts, replacement policies and etc, quantifying the benefits and overheads, emphasizing the non-trivial lessons learned.

6.1 Experimental Setup

All aforementioned components and subsystems of GraphCache(+) have been implemented in Java over ≈6,000 lines of code. Experiments were performed on a Dell R920 host (4 Intel Xeon E7-4870 CPUs, 15 cores each), with 320GB of RAM and 4×1TB disks, running Ubuntu Linux 14.04.4LTS.

6.1.1 Graph Datasets

Fortunately there exist a number of real-world graph datasets that are commonly used in related research. These of course help concretize the effects of any solution on real-world data and allow direct comparison of methods and result repeatability. For this reason, this work will report evaluations conducted over three popular such graph datasets, namely, AIDS[86], PDBS[87], and PCM[91].

More specifically, AIDS[86], the Antiviral Screen Dataset of the National Cancer Institute, contains topological structures of 40,000 molecules. Graphs in AIDS contain on average ≈45 vertices (std.dev.: 22, max: 245) and ≈47 edges (std.dev.: 23, max: 250) each, whereby the few largest graphs have an order of magnitude more vertices and edges. PDBS[87] is a dataset of graphs representing DNA, RNA and proteins, consisting of fewer (600) but larger graphs compared to AIDS, with on average ≈2,939 vertices (std.dev.: 3,215, max: 16,341) and ≈3,064 edges (std.dev.: 3,261, max: 16,781) per graph. PCM[91] consists of 200 graphs representing protein interaction maps, with on average ≈377 nodes (std.dev.: 187, max: 883) and ≈4,340 edges (std.dev.: 1,912, max: 9,416) per graph.

However, it is worth using additional synthetic datasets so as to perform evaluations under characteristics unseen in the real-world datasets. Specifically, this work shall use a synthetic dataset [14] as a larger counterpart to the PCM dataset, consisting of 5× more graphs, each being 2-3× larger on average than the average PCM graph. Interestingly, the Synthetic dataset positions the scalability limit in [14] where experiments verified that none of its studied methods [40, 49, 54, 18, 16, 42] can scale beyond the said limit - graph datasets containing 1000 graphs, with on average ≈892 nodes and ≈7,991 edges per graph, of high average node degree ≈19.52.

Characteristics of these datasets are detailed in Table 6.1. Graphs in AIDS and PDBS have low average node degree (AIDS ≈2.09, PDBS ≈2.13), whereas graphs of PCM and Synthetic have much higher average node degrees (PCM ≈22.39, Synthetic ≈19.52). With respect to dataset with graphs having a high average node degree, it is discovered that GraphCache(+) needs special mechanisms, without which its performance benefits degrade.

Table 6.1: Characteristics of Multiple Datasets

	AIDS	PDBS	PCM	Synthetic
unique vertex labels	62	10	21	20
graphs in dataset	40,000	600	200	1,000
average node degree	2.09	2.13	22.39	19.52
avg (node # per graph)	45	2,939	337	892
std.dev (node # per graph)	22	3,217	187	417
max (node # per graph)	245	16,341	883	7,135
avg (edge # per graph)	47	3,064	4,340	7,991
std.dev (edge # per graph)	23	3,264	1,912	5
max (edge # per graph)	250	16,781	9,416	8,007

6.1.2 Query Workloads

Recall the identified design goals in §4.1.2, the query workload for GraphCache(+) system is required to satisfy a number of criteria.

- It should consider queries that are not guaranteed to have any answer, in spite of merely dealing with queries that are directly generated from dataset graphs as most works [16, 42, 13, 18, 40] did.
- It should be capable of handling various skewness levels regarding the probability distribution of possible queries (in term of the popularity of query graphs or of regions in the dataset graphs), from uniform to highly skewed Zipf distributions.
- It should consist of a large number of queries so as to obtain the reliable experiment results pertaining to the performance.

In spite of the availability of several graph datasets, unfortunately there is a lack of well established benchmarks and/or real-world query logs for these datasets. Therefore, all known works derive queries from the dataset graphs. The current work shall follow the established

principle and the design goals for the generation of workloads, using two different algorithms to synthesize queries from the dataset graphs, to be outlined shortly. Though NP-Complete subgraph isomorphism leads to queries with possibly very long execution times, it shall utilize well over 6 million queries for the performance evaluation.

Type A Workloads

Queries in these workloads are generated in the following manner: first, a source graph is selected randomly from the dataset graphs; then, a node is selected randomly in said graph; finally, a query size is selected uniformly at randomly from those mentioned above and a BFS is performed starting from the selected node. For each new node, all its edges connecting it to already visited nodes are added to the generated query, until the desired query size is reached. For the first two random selections above, two different distributions have been used; namely, Uniform (U) and Zipf (Z), with the probability density function of the latter given by:

$$p(x) = x^{-\alpha}/\zeta(\alpha) \tag{6.1}$$

where ζ is the Riemann Zeta function[93].

Ultimately, there will be three categories of Type A workloads: "UU", "ZU" and "ZZ", where the first letter in each pair denotes the distribution used for selecting the starting graph, and the second for the starting node.

Type B Workloads (with no-answer queries)

These workloads are generated as follows. For each of the query sizes mentioned above, two query pools are first created: a 10,000-query pool with queries with non-empty answer sets against the dataset, and a second 3,000-query pool with no match in any dataset graph (i.e., empty result set).

Queries for the first pool are extracted from dataset graphs by uniformly selecting a start node across all nodes in all dataset graphs, and then performing a random walk till the required query graph size is reached. Generation of no-answer queries has one extra step: it continuously relabels the nodes in the query with randomly selected labels from the dataset, until the resulting query has a non-empty candidate set but an empty answer set against the dataset graphs.

Once the query pools are filled up, workloads are generated by first flipping a biased coin to choose between the two pools (with the "no-answer" pool selected with probability 0%, 20% or 50%), then randomly (Zipf) selecting a query from the chosen pool. Hence produces three categories of Type B workloads: "0%", "20%" and "50%", denoting the above probability used.

6.1.3 Algorithmic Context

GraphCache(+) is intended to be a general-purpose front-end for graph query processing, which entails the indexing strategy of iGQ as explained in §3. GraphCache(+) is designed to assure double use in two dimensions: (i) applicable for both subgraph and supergraph queries; (ii) capable of accommodating both FTV methods and SI algorithms. In fact, any such algorithm is viewed as a pluggable component into the architecture, allowing any future algorithm to be incorporated.

Specifically, GraphCache shall be used on top of three subgraph FTV and three SI methods [1]. For the FTV methods GraphGrepSX [16] (GGSX), Grapes [42], and CT-Index [18] shall be used, particularly because they are proven to be top performers in their class [14]. GGSX indexes paths in a trie and employs VF2 [10] algorithm for the verification stage. Like GGSX, Grapes also indexes paths, but utilizes location information to pursue the better filtering efficiency. For the verification, Grapes performs on (typically) small connected components of dataset graphs, with several threads running in parallel. On the other hand, CT-Index derives canonical representations for (tree, cycle) features of dataset graphs, to the fact that finding string-based canonical forms for trees and cycles can be done in linear time (unlike general graphs). These representations are then hashed into a bitmap structure per dataset graph. Filtering is thus performed by checking bitmaps between query and dataset graph, with simple bitwise operations.

With respect to the SI methods, GraphCache(+) shall use GraphQL [17] as provided by [13] and a modified version of VF2 [10] (denoted VF2+) provided by [18], again for being well-established and good performers [13, 14]; vanilla VF2 [10] is also used since it has been used by several FTV implementations [16, 42, 13].

GraphCache(+) shall use the Java Native Interface to directly execute the native C++ implementations of Grapes, GGSX, GraphQL and VF2, while CT-Index and VF2+ are implemented in Java and thus invoked directly from GraphCache(+). This diversity in the implementation languages of the incorporated methods attests to the flexibility of GraphCache(+).

6.1.4 Dataset Change Plan

As GraphCache+ is the first study to explore the topic of handling graph queries against dynamic underlying dataset, the corresponding dataset plan shall be created so as to test the performance of GraphCache+. Dataset change operations are performed in batches, with

[1] We would like to thank the authors of [16, 42, 18, 13] for sharing their source code of subgraph queries, so that GraphCache could incorporate their native implementations for the performance evaluations. For supergraph queries, we have tried the utmost efforts in communicating related authors. Though some of them kindly replied, we have not been granted any access of their source code yet.

occurrence time indicated by the IDs of queries in workload (see Figure 5.4). Each plan consists of 2,000 operations (in 100 batches, 20 operations per batch), during the processing of 10,000 queries. A batch of operations are generated as following:

- First, an occurrence time for the batch is selected uniformly at randomly from the id of queries;
- Next, a type uniformly selected from {ADD, DEL, UA, UR}, a graph uniformly selected from dataset (ADD using the initial dataset instead of synthesizing additional graphs so as to maximumly keep the original dataset characteristics; DEL, UA and UR using the up-to-date dataset at running time) and a uniformly selected edge within the graph providing UA or UR being the selected type (UA would add an edge that has not been in graph yet; UR would remove an existed edge of graph) codetermine a specific operation – such process is repeated until the batch contain the required number of operations.

6.1.5 Parameters and Metrics

The default value for the upper limit on the sizes C of the Cache and W of the Window stores were C = 100 and W = 20 respectively; experiments were also performed with other values for both C (200, 300) and W (50, 100, 200) to test their impact on performance.

Grapes and GGSX were configured to index paths up to length 4, and CT-Index to index trees up to size 6 and cycles up to size 8 using 4,096-bit-wide bitmaps. For Grapes, two alternatives are examined, Grapes1 and Grapes6, with 1 and 6 threads respectively. To be fair, the code of Grapes is altered so to stop query processing after the first match in each dataset graph (for the decision problem of interest). Please note that all mentioned values match their default configurations in [16, 42, 18].

As to the skewed distribution of queries, Zipf $\alpha = 1.4$ is by default; it also uses $\alpha = 1.1$ representing a smaller skewness and $\alpha = 1.7$ for a higher skewness. As a reference point, web page popularities follow a Zipf distribution with $\alpha = 2.4$ [93].

Query graphs are generated in different sizes: 4, 8, 12, 16 and 20-edge graphs for the smaller AIDS and PDBS datasets; 20, 25, 30, 35 and 40-edge queries for the larger PCM and Synthetic datasets (as almost half of the dataset graphs in AIDS contain no more than 40 edges, larger queries are not usable). Such sizes are typical in the literature [42, 18, 40].

Workloads for AIDS and PDBS consist of 10,000 queries, while workloads for PCM and Synthetic contain 5,000 queries for practical reasons, as PCM/Synthetic queries take much longer to execute. GraphCache(+) only allows one for one Window (i.e., 20 queries) before starting measuring the performance.

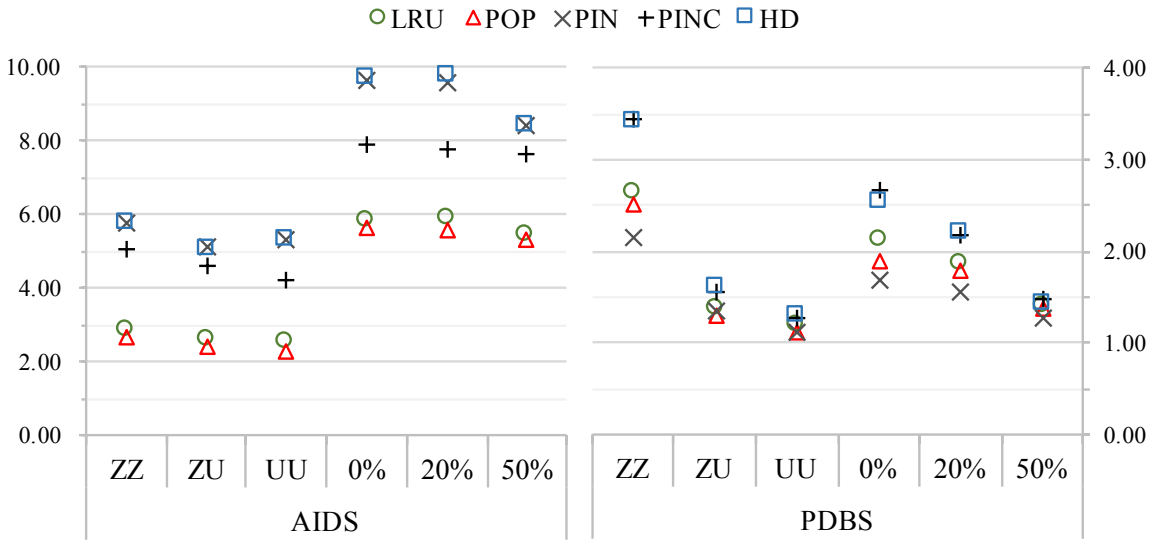


Figure 6.1: GC Speedup in Query Time over CT-Index across Replacement Policies

Both the benefits and the overheads of GraphCache(+) will be investigated. Reported metrics include query time and number of sub-iso tests per query, along with the speedups introduced by GraphCache(+). Speedup is defined as the ratio of the average performance (query time or number of sub-iso tests) of the base Method M over the average performance of GraphCache(+) when deployed over Method M (i.e., speedups >1 indicate improvements). The results were produced over more than 6 million queries! As a yardstick, [74] (also a cache but for XML databases) report a query time speedup of 2.6× with 10,000-query workloads generated using Zipf $\alpha = 1.5$, and a 1,500-query warm-up.

6.2 Results and Insights

6.2.1 HD Wins

Figure 6.1/6.2/6.3 depicts the speedups attained by GraphCache when CT-Index, Grapes1 or GQL was used as Method M respectively. Results for other FTV and SI methods showed similar trends and are thus omitted. One can see that GraphCache attains significant speedups (up to 42× lower query processing times in this case), and that it is always one of the GC exclusive policies that produces the best results. A more subtle observation, though, is that there are cases where PIN wins over PINC and vice-versa; for example, in Figure 6.1, PIN dominates the scene for queries against the AIDS dataset but it is PINC that takes the lead when querying the PDBS dataset.

Ultimately, different cache replacement policies exhibit different performance depending on the workload and dataset characteristics. The question then is how to choose a replacement

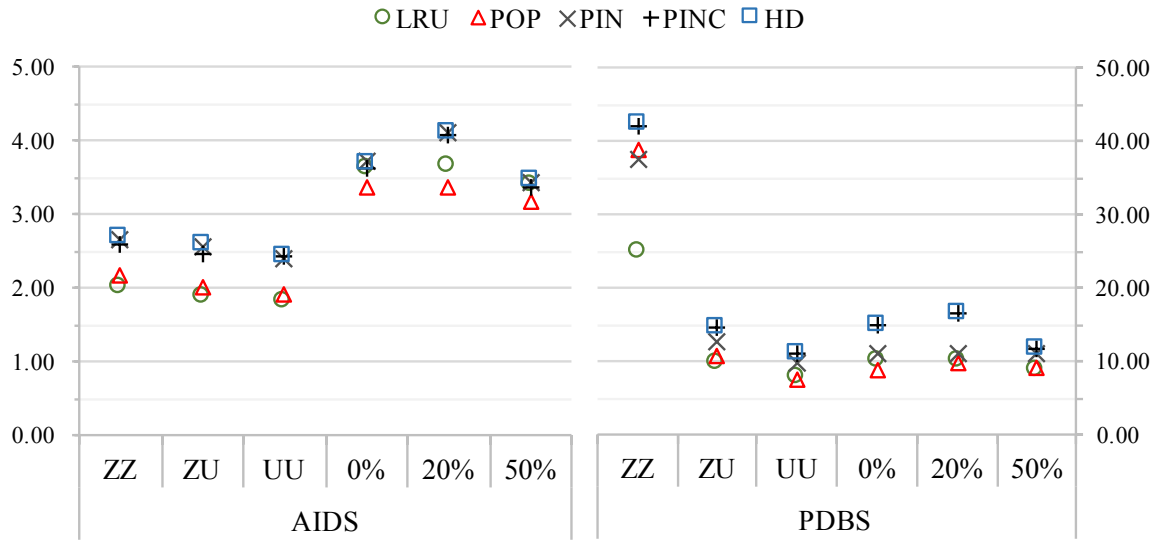


Figure 6.2: GC Speedup in Query Time over Grapes1 across Replacement Policies

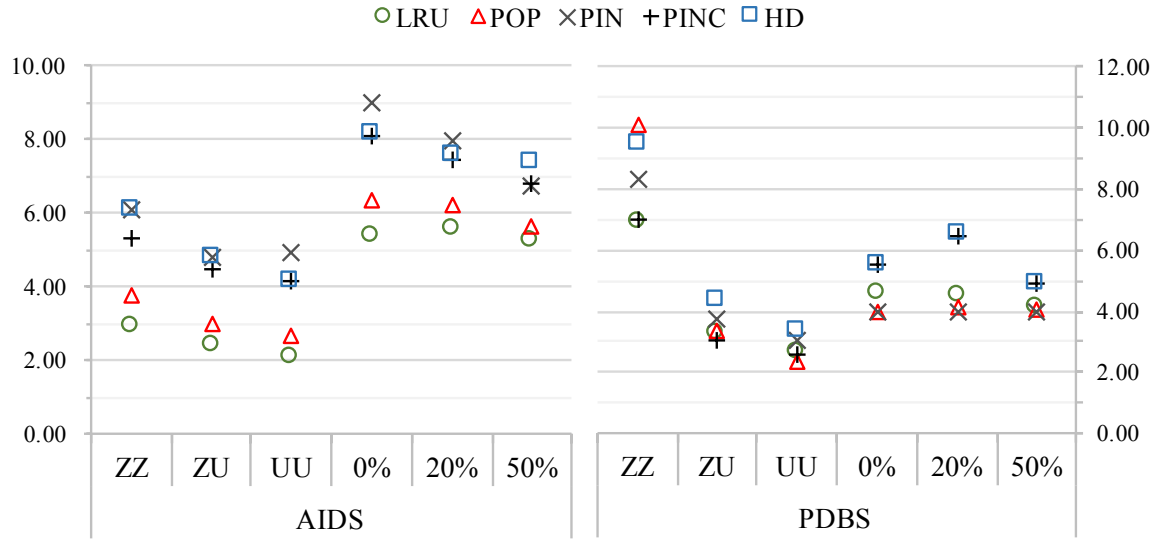


Figure 6.3: GC Speedup in Query Time over GQL across Replacement Policies

policy when said characteristics are unknown a priori. Our answer to this question then, and the first takeaway message, is: **When in doubt, use the HD replacement policy, as it always manages to do better or on par with the best of the alternatives.** The remaining of this section will be using HD as the replacement policy; results for other caching policies show similar trends and are thus omitted.

6.2.2 GC/FTV versus FTV

Figure 6.4/6.6 depicts speedups in query processing time against all FTV methods for queries on the PDBS/AIDS dataset (results for other datasets are similar). For example, in Figure 6.4, query processing time speedups range from 1.60× (i.e., 37.5% lower processing time)

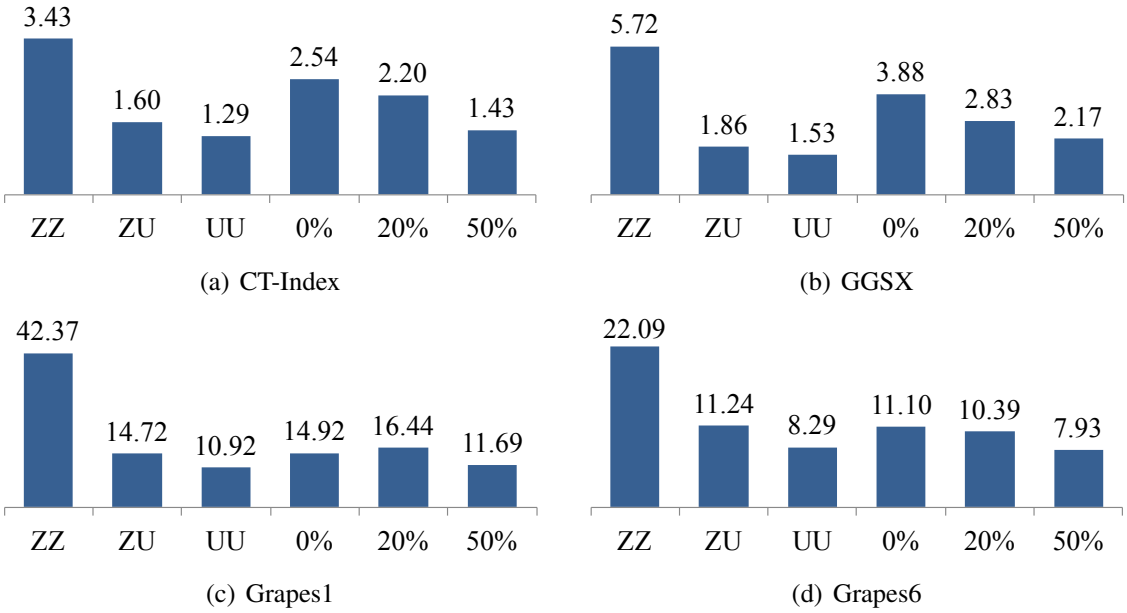


Figure 6.4: GC Speedup in Query Time for PDBS across FTV Methods M

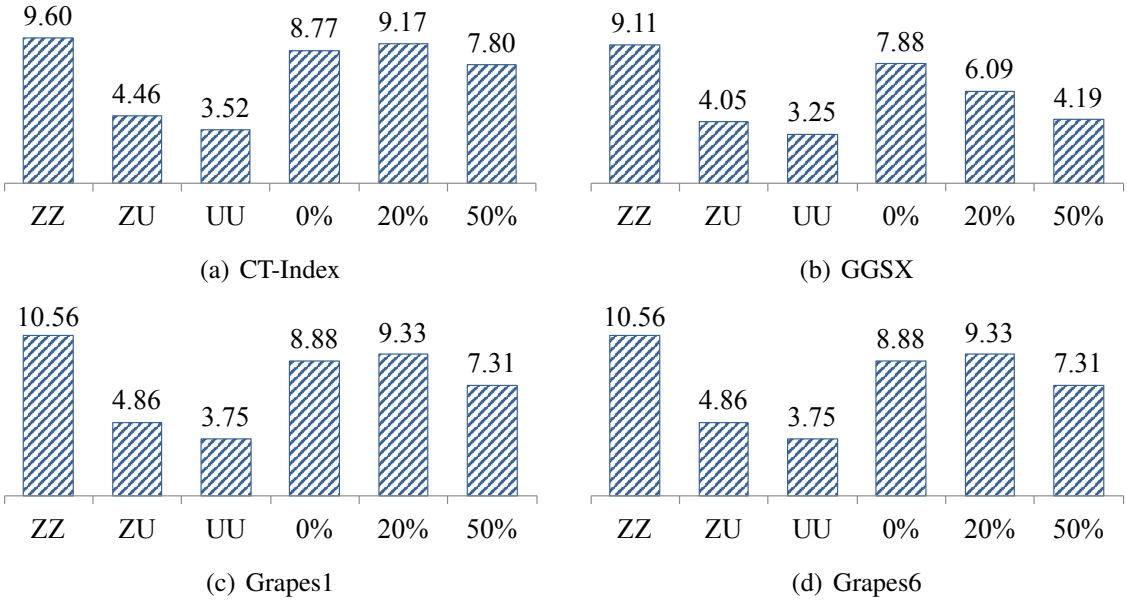


Figure 6.5: GC Reduction (Speedup) in Number of Sub-Iso Tests for PDBS across FTV Methods M

to more than 42×. A similar picture is drawn in Figure 6.5/6.7 for speedups in the number of subgraph isomorphism tests performed. Juxtaposing Figure 6.4 and 6.5 (ditto for Figure 6.6 and 6.7) leads to the following interesting insight: **Reductions in the number of subgraph isomorphism tests do not translate directly into reductions in query time**; this validates our claim that cache hits in GraphCache render different benefits. In all cases, though, **GraphCache achieves significant improvements in both query processing time and number of subgraph isomorphism tests performed**.

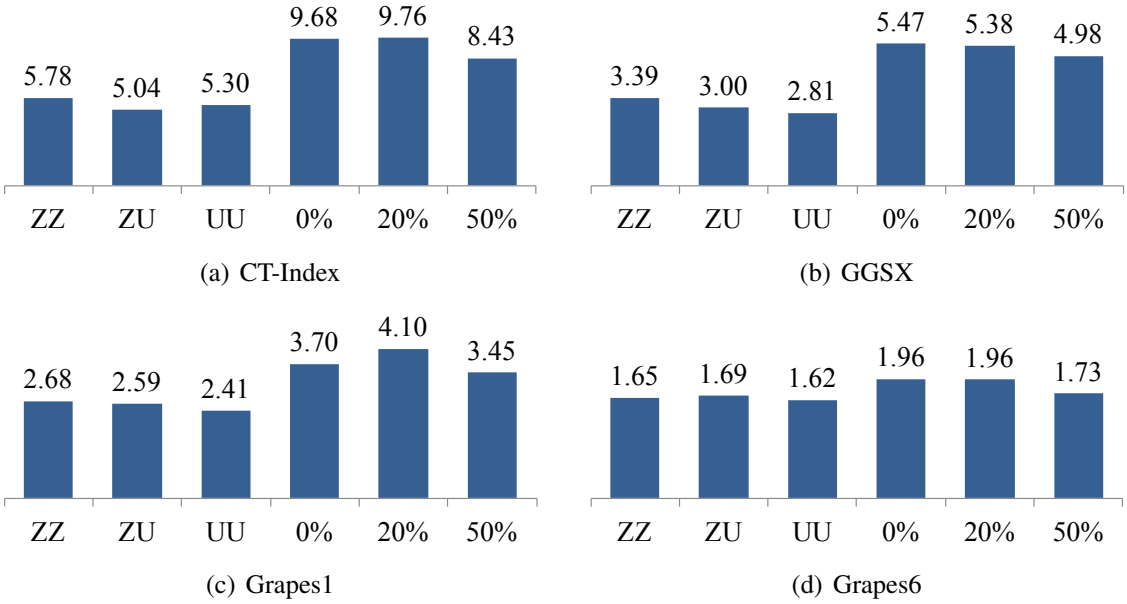


Figure 6.6: GC Speedup in Query Time for AIDS across FTV Methods M

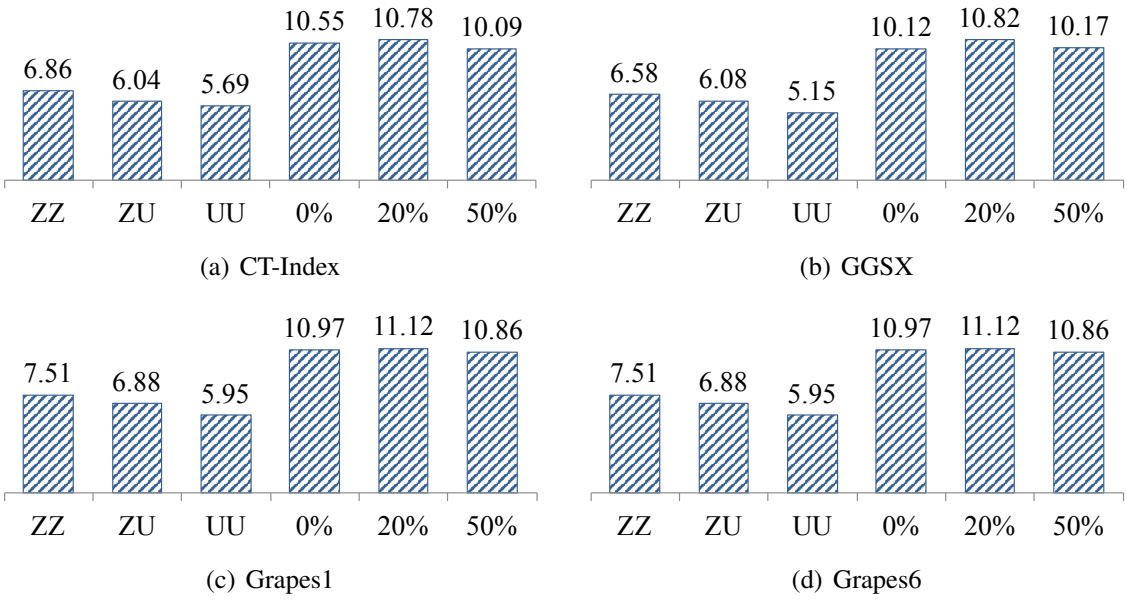
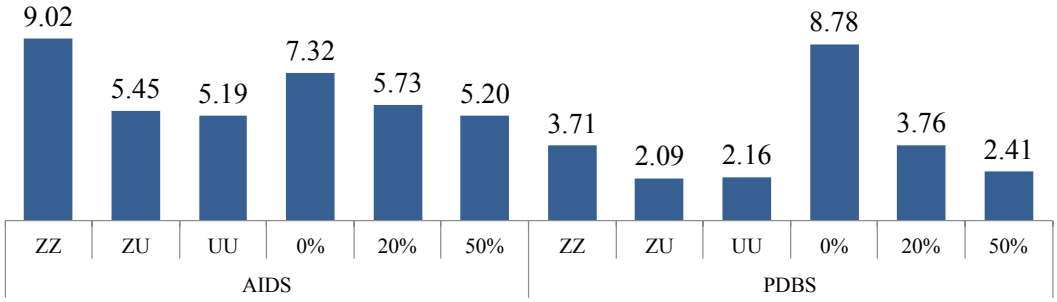


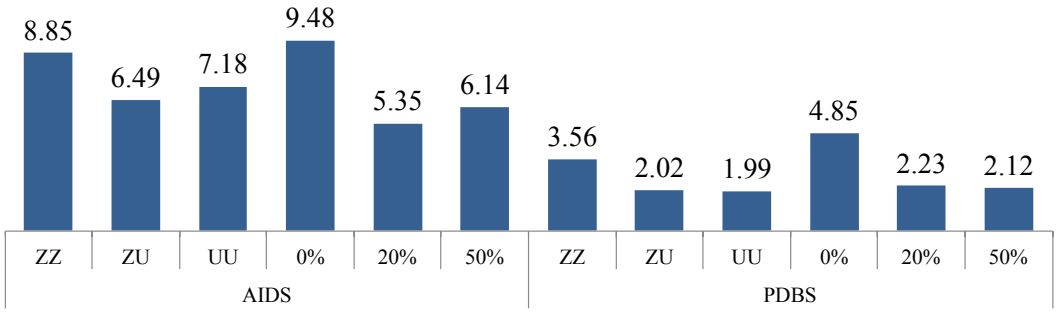
Figure 6.7: GC Reduction (Speedup) in Number of Sub-Iso Tests for AIDS across FTV Methods M

6.2.3 GC/SI versus SI

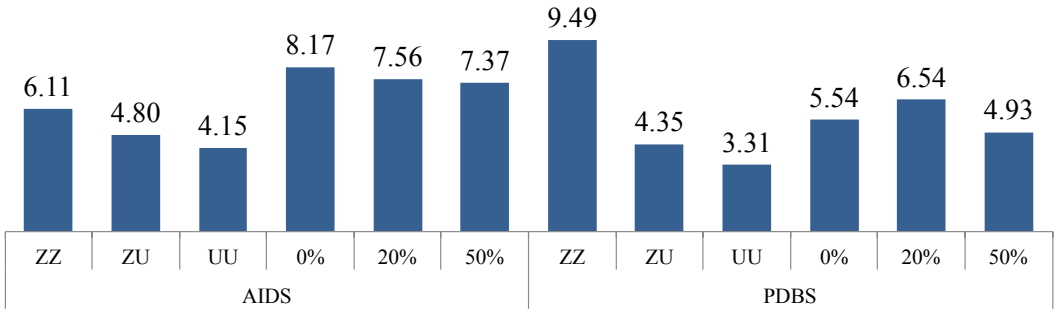
Figure 6.8 depicts the query processing speedups of GC over SI methods considered in this work. One can see that **GraphCache improves the performance of well-established SI methods**, with the same meagre 100-query cache configuration as above. **This is significant in that GraphCache provides a new way to expedite sub-iso tests (as opposed to developing yet another SI heuristic) which is usable with any mainstream SI method.**



(a) VF2



(b) VF2+



(c) GQL

Figure 6.8: GC Speedup in Query Time for AIDS/PDBS across SI Methods M

Note the interesting finding in Figure 6.8(b) that VF2+ speedup for AIDS UU workload is close to that of AIDS ZU (7.18 vs 6.49), whereas one might have expected a different outcome. Intuitively, the ZU workload bears more exact-match hits than UU, due to the skewness of selecting source graphs during query generation (see §6.1.2). And it does: we measured circa 2.5X the number of exact-match cache hits in ZU vs UU. However, recall that GraphCache exploits also subgraph/supergraph hits. When exact-matches are not frequent, GraphCache loads graphs in the cache that can help with their subgraph/supergraph relationships. Indeed, around 2X such matches are discovered for the UU workload vs ZU. Of course, the overall performance result is a very complex picture and depends on how big benefit is each saved exact-match versus each saved subgraph/supergraph match. Ditto for the similar phenomenons of FTV methods in Figure 6.6. But the key insight here is that **by**

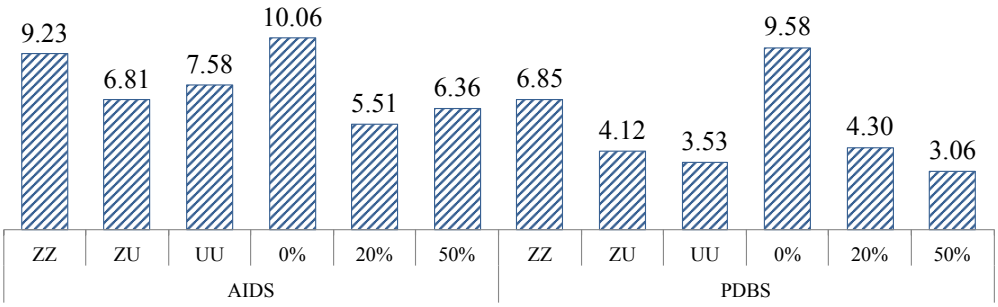


Figure 6.9: GC Reduction (Speedup) in Number of Sub-Iso Tests for AIDS across SI Methods M

utilizing exact-matches and subgraph/supergraph matches, GraphCache can introduce significant benefits in both skewed and non-skewed workloads.

Furthermore, Figure 6.9 shows the speedups in the number of subgraph isomorphism tests performed. Please note that **under a given GraphCache configuration (specified by dataset, workload, replacement policy, the upper limit on the sizes of Cache and the Window stores), whatever SI solution being the Method M, GraphCache results exactly the same pruned candidate set for each query.** Therefore, Figure 6.9 is independent of the three methods (i.e., VF2, VF2+, GQL) considered in this work, as well as any other SI method that the community shall contribute in the future.

6.2.4 GC/SI versus FTV

Let us now take a step back and look at how FTV methods and GraphCache operate: they both expedite queries by filtering out dataset graphs, thus producing a reduced candidate set. The logical question then is: what happens if we pitch a full-blown FTV method against GraphCache operating on top of a simple SI method?

Figure 6.10 shows the results when comparing GraphCache on top of SI methods VF2 against GGSX (using VF2 for verification chores). Correspondingly, Figure 6.11 is for GraphCache on top of VF2+ versus CT-Index. And Figures 6.12 and 6.13 depict their speedups in the number of subgraph isomorphism tests. Take the Figure 6.11 for example. For the small 100-query cache, GraphCache performs on par or better than CT-Index in eight out of twelve cases, slightly worse in two other cases, and takes up to double the time of CT-Index in the remaining worst case. Note, though, that GraphCache's space requirements are under $\approx$15% of the space requirements of CT-Index's index for PDBS and under 0.2% for AIDS, and that CT-Index has the fastest verification algorithm and by far the smallest index among all FTV methods considered in this work. The situation is more impressive when using the larger (500-query) cache, where GraphCache matches or outperforms CT-Index across the board (by a factor of 2.6$\times$ on average). Note that even for this "larger" cache, GraphCache's

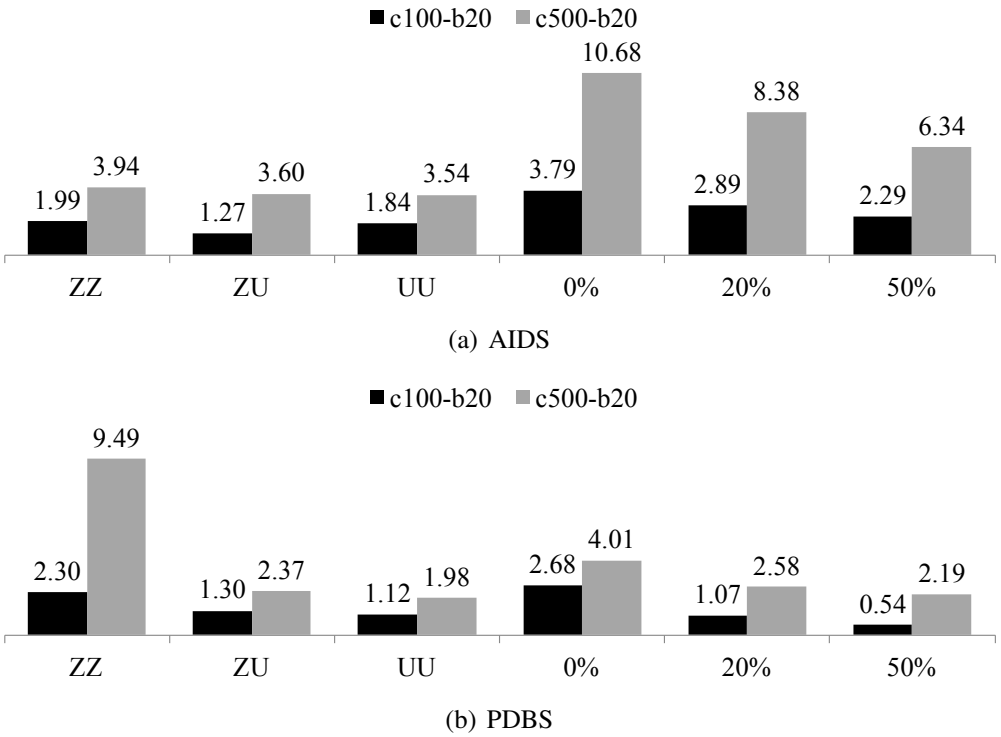


Figure 6.10: Speedup in Query Time of GC/SI vs FTV across Datasets and Workloads: GC/VF2 vs GGSX

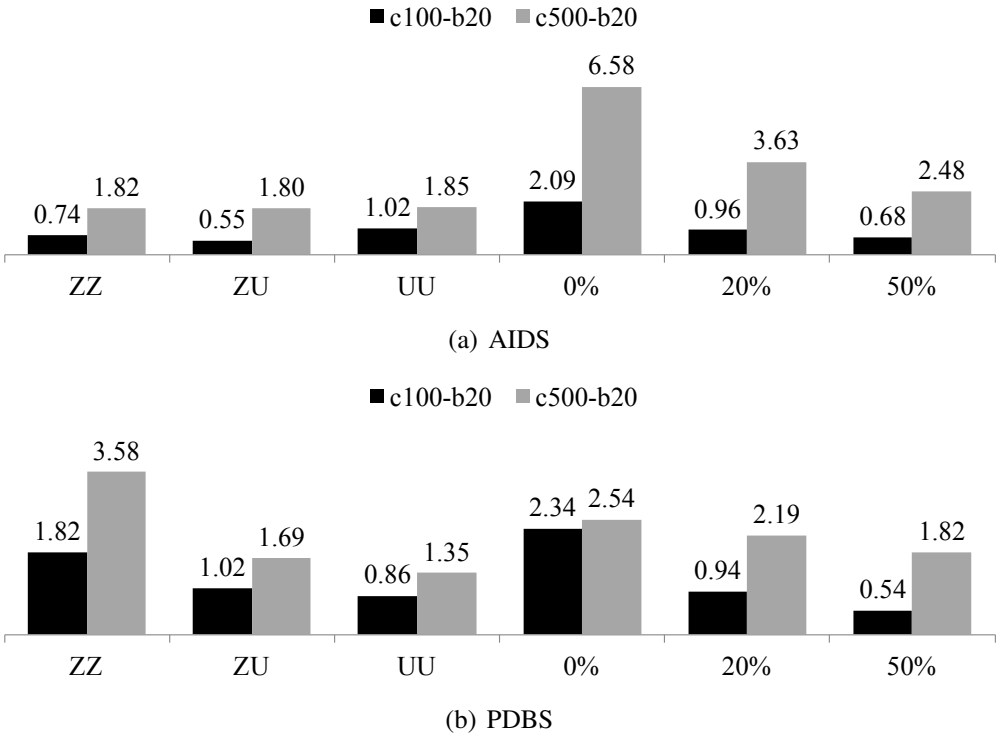


Figure 6.11: Speedup in Query Time of GC/SI vs FTV across Datasets and Workloads: GC/VF2+ vs CT-Index

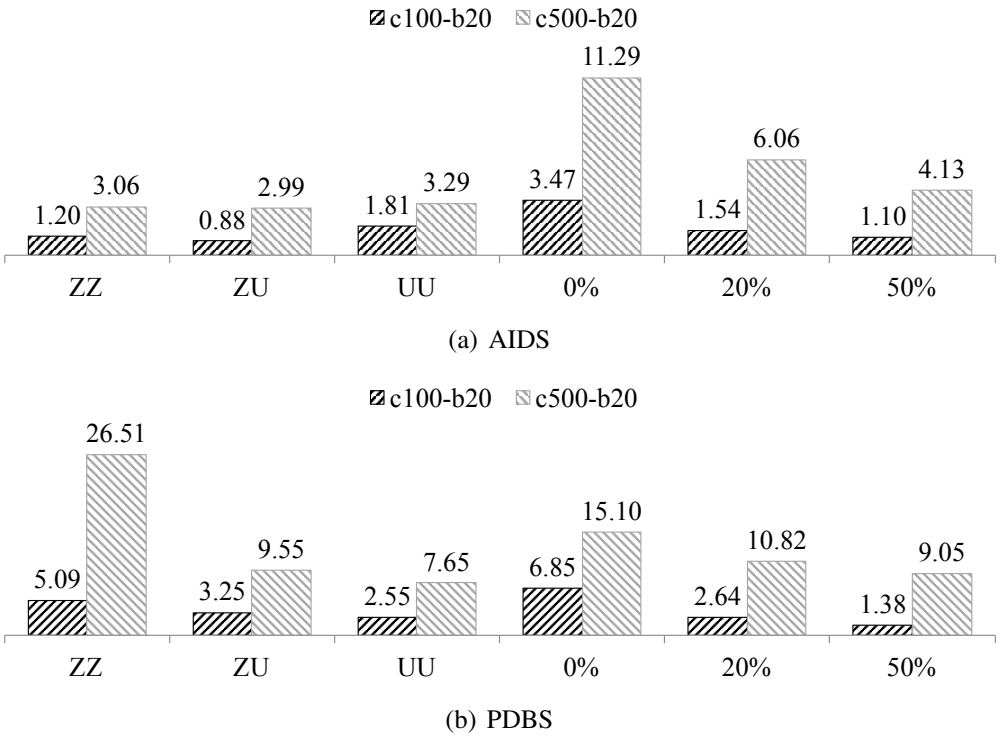


Figure 6.12: Reduction (Speedup) in Number of Sub-Iso Tests for GC/SI vs FTV across Datasets and Workloads: GC/VF2 vs GGSX

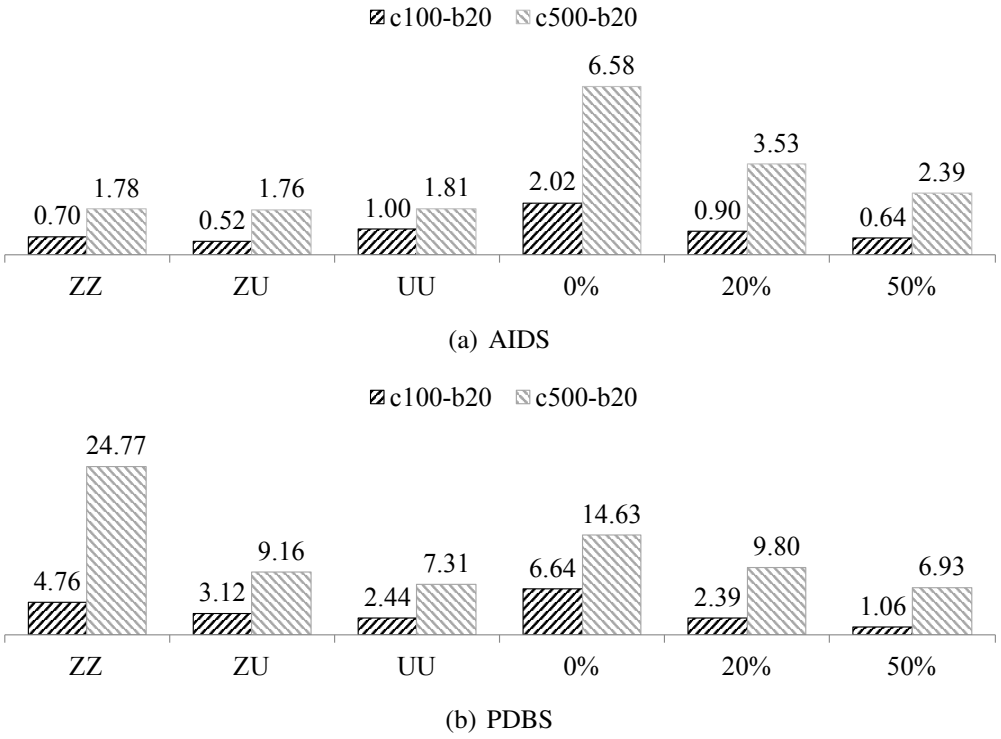


Figure 6.13: Reduction (Speedup) in Number of Sub-Iso Tests for GC/SI vs FTV across Datasets and Workloads: GC/VF2+ vs CT-Index

space requirements are less than ≈70% of CT-Index's index size for PDBS and less than 1% for AIDS (results over GGSX and Grapes show similar trends). The conclusion is then that **GraphCache can replace the best-performing FTV methods, achieving comparable or better performance for a fraction of the space and no pre-processing cost** as no indexing is needed.

6.2.5 GC+/SI versus SI

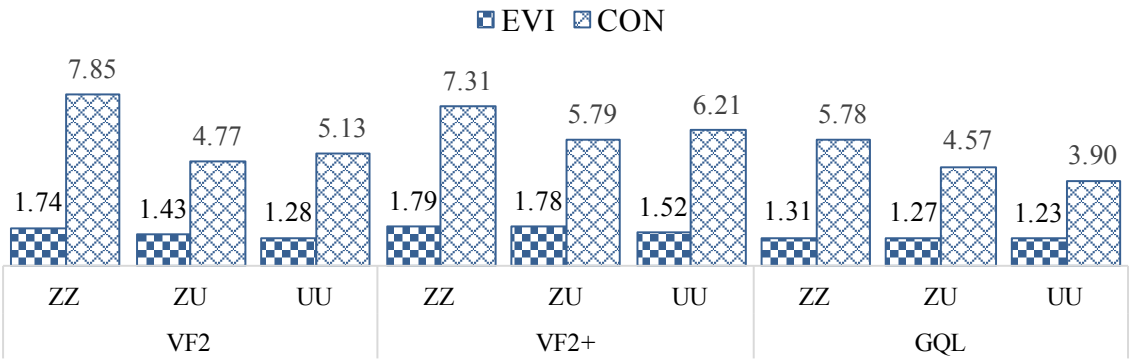


Figure 6.14: GC+ Speedup in Query Time for Type A Workloads

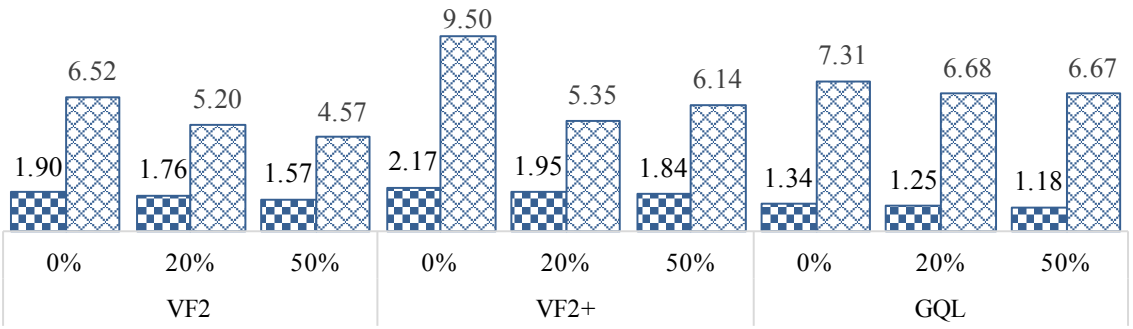


Figure 6.15: GC+ Speedup in Query Time for Type B Workloads

Now, turning attention to the performance of GraphCache+, where graph query processing is against a dynamic dataset based on AIDS. Figure 6.14 and Figure 6.15 depict the query time speedups of GC+ across all method M and workloads. We can see that **CON achieves considerable speedup with the meagre 100-query cache configuration** whereas gains of EVI are limited. Similar to that of GraphCache in Figure 6.8(b); regarding the interesting finding that speedups of GraphCache+ for UU workload are close to those of ZU (e.g., 4.77 vs 5.13 against VF2 base method), it is due to the fact that the overall performance result is a very complex picture and depends on how big benefit is each saved exact-match versus each saved subgraph/supergraph match. This echoes the claim that **by utilizing exact-matches and subgraph/supergraph matches, GraphCache+ can benefit both skewed and non-skewed workloads**.

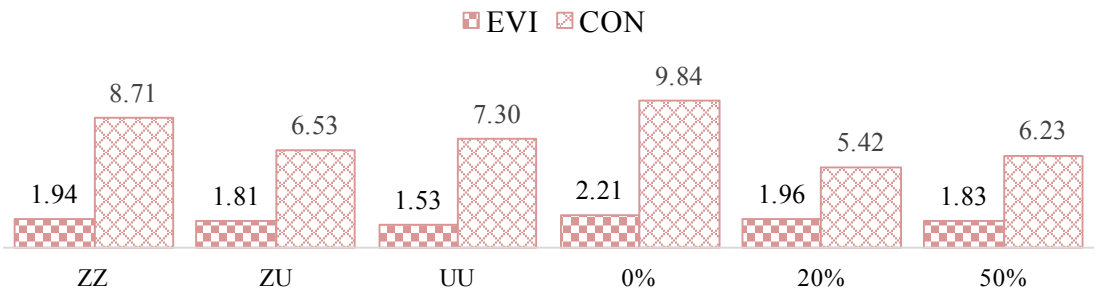
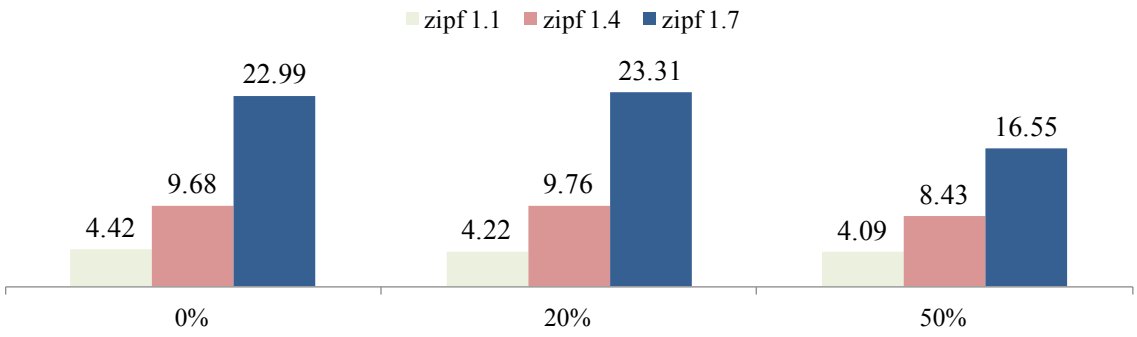


Figure 6.16: GC+ Reduction (Speedup) in Number of Sub-Iso Tests

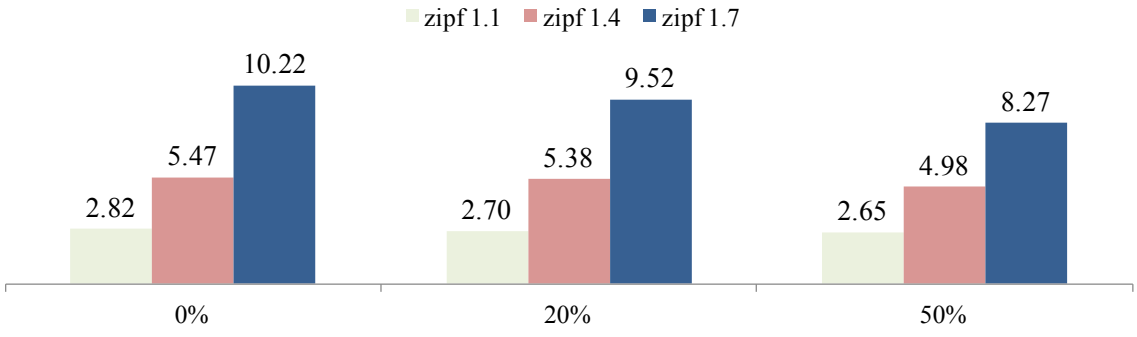
Correspondingly, Figure 6.16 shows the speedups in the number of subgraph isomorphism tests performed in GraphCache+. Along with GraphCache, **regarding a configuration specified by dataset, dataset change plan, workload, replacement policy, cache model EVI/CON, the upper limit on the sizes of Cache and the Window stores, GraphCache+ generates exactly the same final candidate set for each query, for each SI method**. Again, Figure 6.16 is independent of the SI methods considered. Juxtaposing Figures 6.14, 6.15 and 6.16 delivers the similar insight of GraphCache+ as that of GraphCache, i.e., reductions in the number of subgraph isomorphism tests do not translate directly into reductions in query time. In all cases, though, **GraphCache+ achieves significant improvements in both query processing time and number of subgraph isomorphism tests**.

6.2.6 Varying the Skewness of Query Distribution

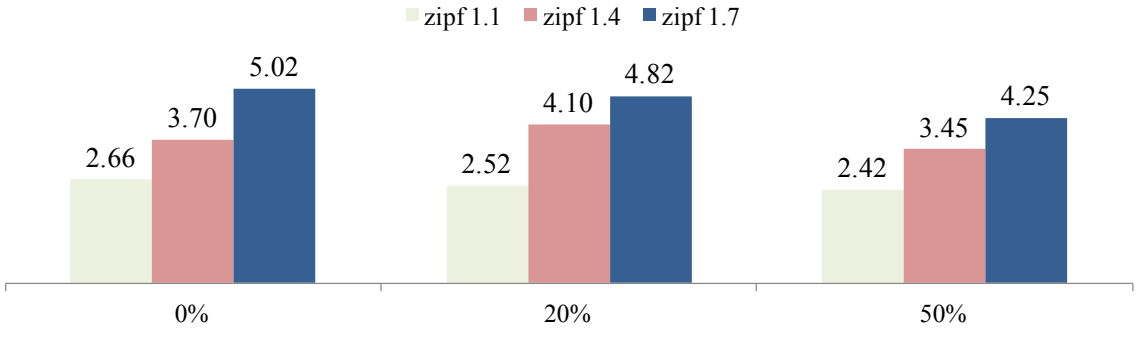
Figure 6.17 shows the speedups achieved by GraphCache for Type B workloads against the AIDS dataset, for various values of the Zipf α skewness parameter (results for number of sub-iso tests and other workloads show similar trends and are thus omitted). We can see that **the more skewed the query distribution, the higher the gains from caching**. This is, of course, expected and has been shown times and again in related work on traditional caches, as caches are built on the premise of (temporal) locality of reference and thus more skewed query distributions have the potential to translate to higher hit ratios. A subtler, but equally important observation here, reached by examining Figure 6.4 in the light of the above result, is that **GraphCache leads to significant performance gains even for query workloads with uniform query popularity distributions**. These distributions represent worst-case scenarios for caching schemes, but we can see speedups from 1.29× (≈20% lower times) up to ≈11× for the UU workloads, emphasizing a significant characteristic of GraphCache where the realm of "locality" is extended by subgraph/supergraph matches among queries, in addition to the traditional exact-match of isomorphic queries.



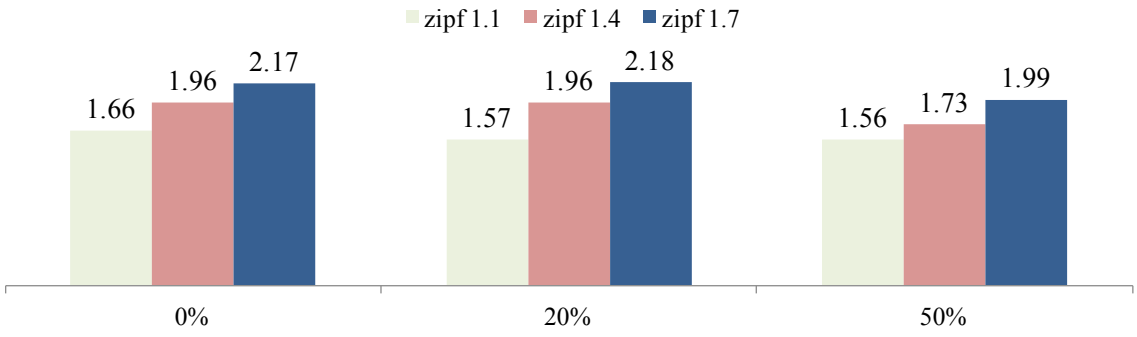
(a) CT-Index



(b) GGSX



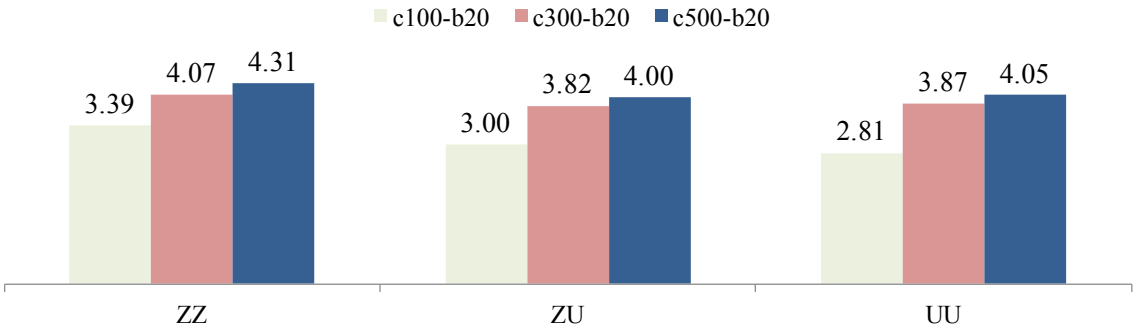
(c) Grapes1



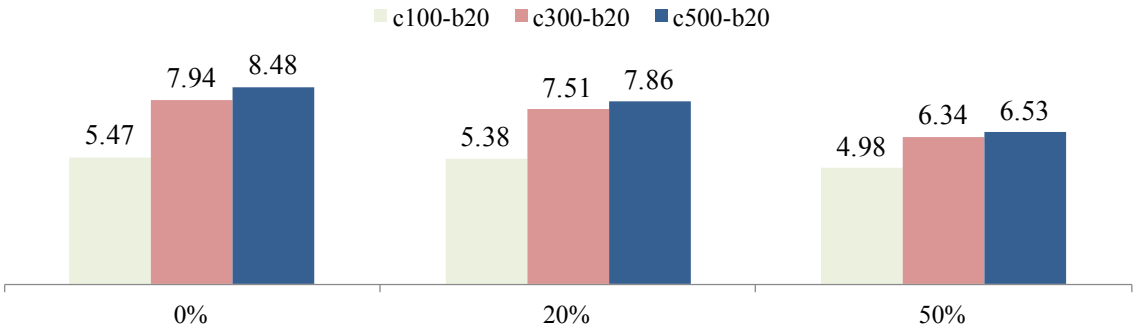
(d) Grapes6

Figure 6.17: GC Speedup in Query Time for Type B Workloads on the AIDS Dataset, for Various Values of Zipf α

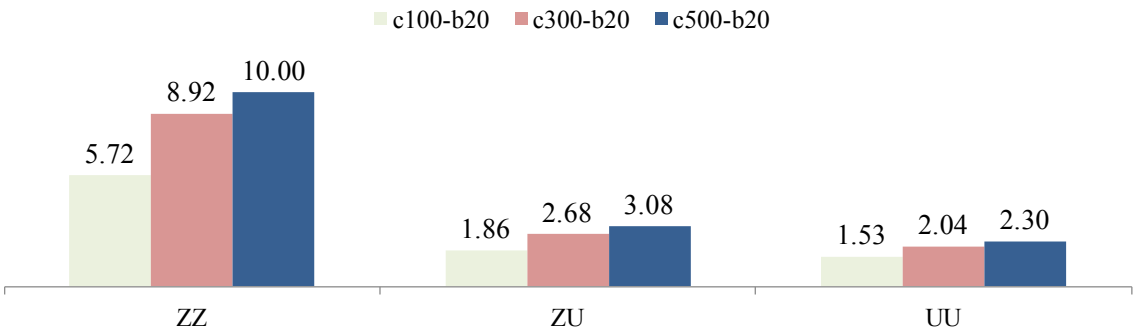
6.2.7 Various Cache Sizes



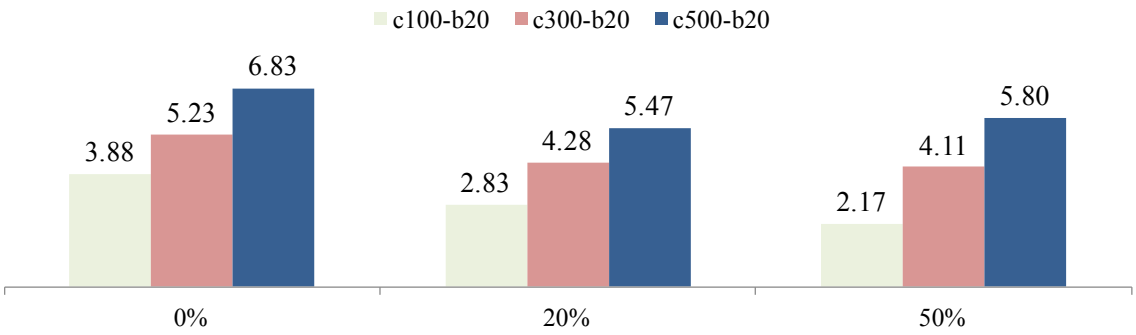
(a) AIDS/Type A Workloads



(b) AIDS/Type B Workloads



(c) PDBS/Type A Workloads

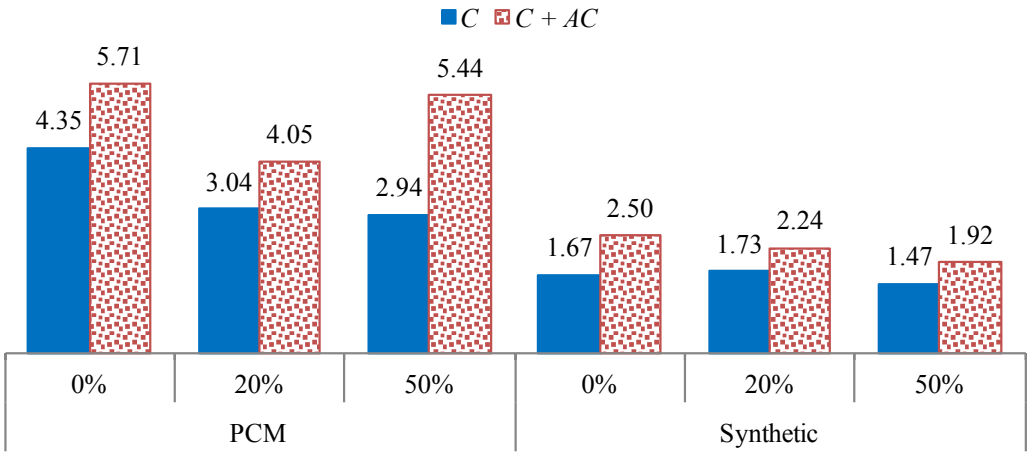


(d) PDBS/Type B Workloads

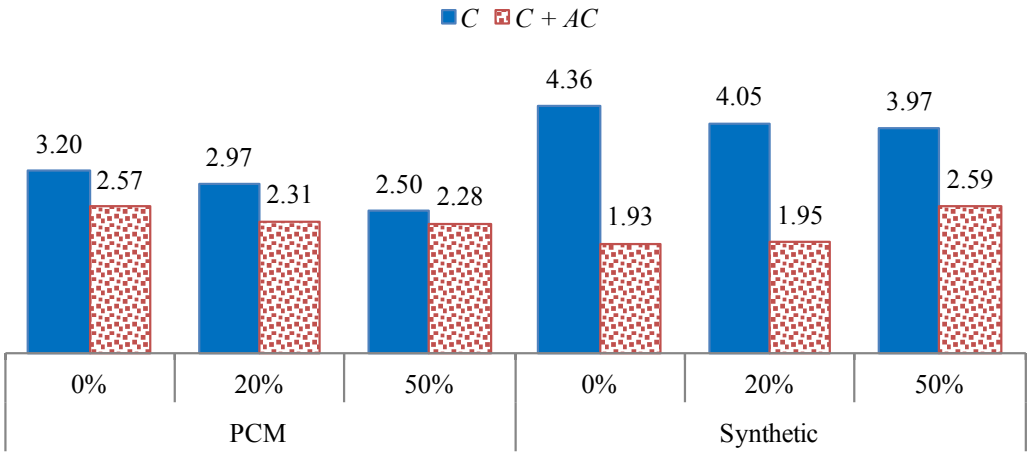
Figure 6.18: GC Speedup in Query Time against GGSX with Various Cache Sizes

Figure 6.18 shows the performance of GraphCache against GGSX for queries on AIDS and PDBS, for varying cache sizes (results for other methods and datasets show similar trends). We can see that **increasing the cache size improves the performance of the cache**. However, this does not mean that one can increase the size of the cache indefinitely; the size of the cache is first limited by the amount of main memory available for GraphCache, then by the overhead associated with updating the cache contents (more on this shortly).

6.2.8 Higher Gains with Cache Admission Control



(a) Query Time Speedups



(b) Reduction (Speedup) in Number of Sub-Iso Tests

Figure 6.19: GC Performance vs Grapes6 for Type B Workloads on PCM/Synthetic Datasets

Figure 6.19 shows the speedups in query time (6.19(a)) and number of sub-iso tests (6.19(b)) against Grapes6 for the PCM and Synthetic datasets, attained when the cache admission control is disabled (C) and enabled ($C + AC$). For clarity, performance without specific notes refer to turning off the cache admission control (C) by default. We can see that **cache**

admission control leads to even higher speedups, thus validating our observation regarding cache pollution and the appropriateness of our "expensiveness"-based mechanism.

A subtler observation is that the corresponding speedup in the number of sub-iso tests is reduced when cache admission control is enabled, as shown in Figure 6.19(b). For better understanding of this trend, let us concentrate on the Synthetic-50% workload: GC without admission control yields a speedup as high as $\approx 4\times$ in the number of sub-iso tests, but the resulting query time speedup is only $\approx 1.5\times$. The reason is that top expensive queries do not benefit as much when the cache is **polluted**: more specifically, the average time for the top-1% most time-consuming queries is $\approx$16.5 seconds with Grapes6, going down to $\approx$15 seconds for GraphCache without admission control – a $1.1\times$ speedup; the remaining 99% "inexpensive" queries enjoy speedups of $2\times$, going from $\approx$0.200 seconds down to $\approx$0.100 seconds, but they account for a much smaller percentage of the overall query processing time compared to the top-1% ones. When the admission control mechanism is enabled, these top-1% expensive queries are prioritized, with their average query processing time going down considerably to $\approx$10 seconds – a much improved $1.65\times$ speedup. Hence, **despite the lower reduction (speedup) in number of subgraph isomorphism tests, the overall query processing time benefits greatly**.

6.2.9 Negligible Space Overhead

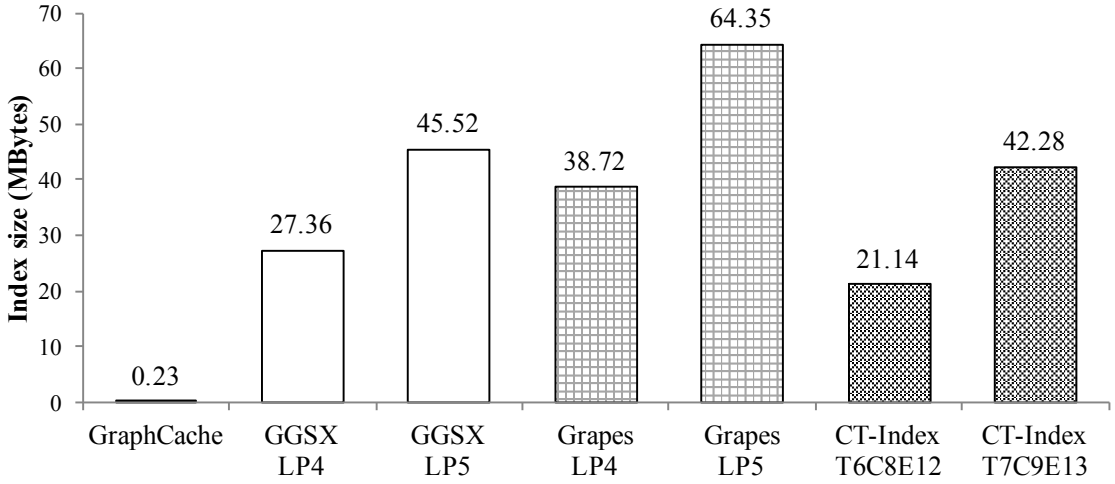


Figure 6.20: Absolute Index Sizes (in MByte) for AIDS with GC Cache Size of 500

We have shown so far that GraphCache leads to significant decreases in the query processing time and number of subgraph isomorphism tests of both FTV and SI methods. Recall that subgraph isomorphism tests take up the majority of the query processing time for FTV methods. A logical consideration, then, would be to try and increase the filtering power of

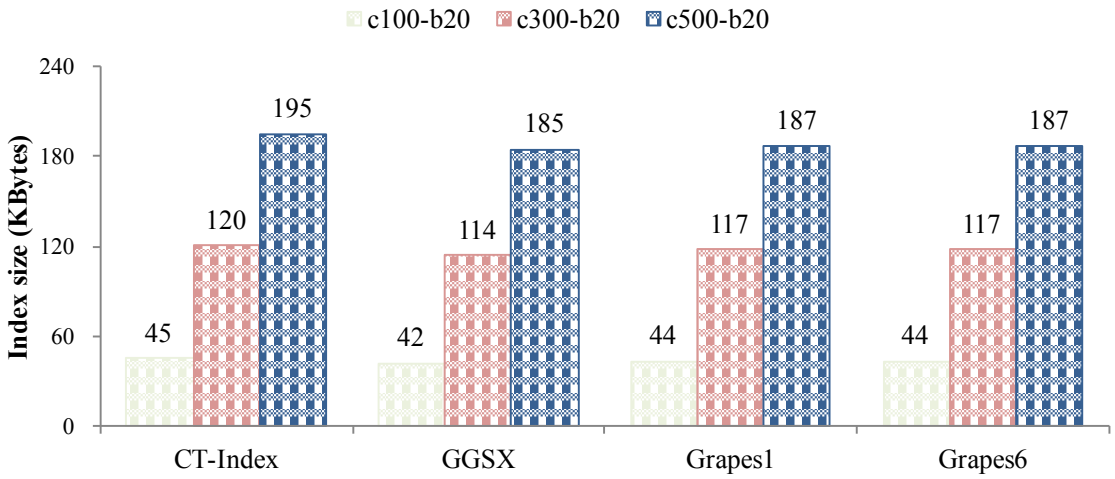


Figure 6.21: GC Space Overhead (in KByte) with Various Cache Sizes

these methods so as to further decrease the size of the resulting candidate set. This can be accomplished by increasing the size of the features recorded by FTV methods; larger features bear higher discriminative power as, obviously, the larger a feature the less its occurrences in dataset graphs.

To this end, we reconfigured all FTV methods increasing their feature sizes by just one (i.e., max path length of 5 for Grapes and GGSX; trees of size 7, cycles of size 9, and 8192 bits per bitmap for CT-Index). As shown in Figure 6.20, this minimal increase in feature size indeed led to better performance, with the average query processing time going down by approximately 10%; however, it also led to an almost doubling of the space required for the FTV indexes across all methods. At the same time, **GraphCache accomplishes its speedup for a negligible space overhead**; for the AIDS dataset, the memory and disk space required by GraphCache was just over 1% (0.23 versus sizes of Method M; the shown 0.23 MB being the maximum query index across all datasets/workloads/replacement policies when the cache size is 500) of the space required for the indexes of the various FTV methods, but leading to time speedups of up to 42× (see Figure 6.4).

Along these lines, Figure 6.21 depicts the size of GraphCache data stores (in KByte) for different cache sizes of AIDS 20% workload. We can see that, as expected, the space overhead of GraphCache increases almost linearly with the size of the cache. Now, contrast the numbers in the figure with the more than 20 MBytes required for the index of CT-Index (by far the smallest of the indexes across FTV methods), keeping in mind that for this meagre space overhead GraphCache achieves query time speedups of 10×, 19× and 21× against CT-Index for the cache sizes shown in the figure (i.e., C =100, C =300 and C =500 respectively).

Now, contrast the numbers in the figure with the more than 20 MBytes required for the index

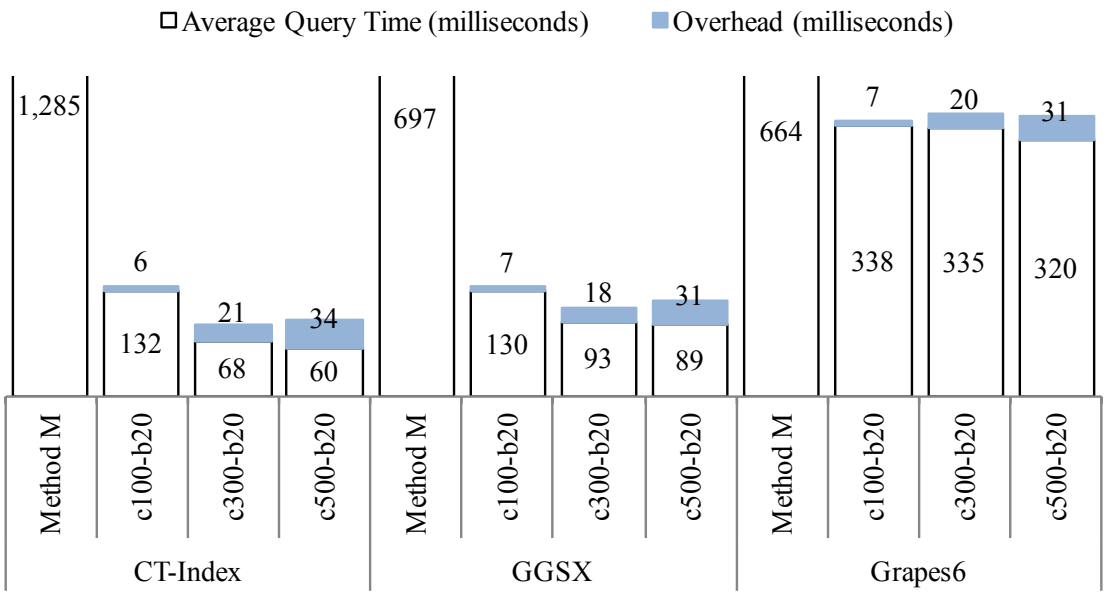


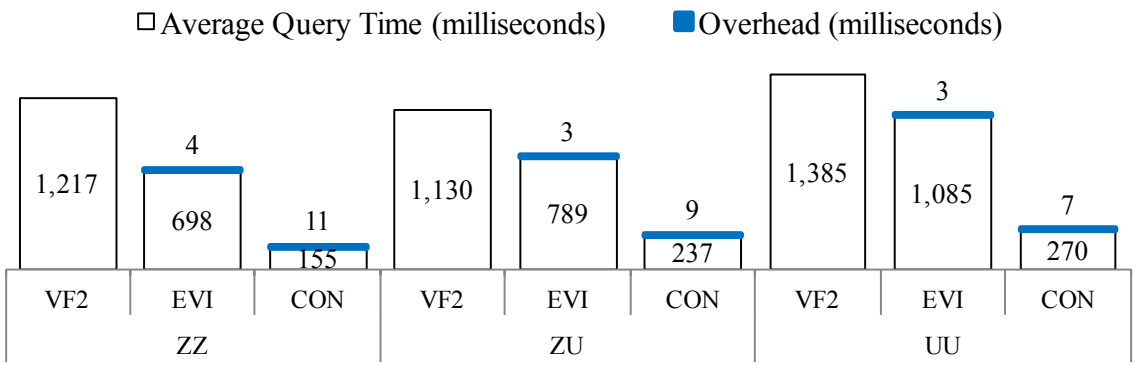
Figure 6.22: GraphCache: Average Execution Time and Overhead (in Millisecond) Per Query for the 20% Workload on AIDS Dataset

of CT-Index (by far the smallest of the indexes across FTV methods), keeping in mind that for this meagre space overhead GraphCache achieves query time speedups of 10×, 20× and 24× against CT-Index for the cache sizes shown in the figure (i.e., C =100, C =300 and C =500 respectively).

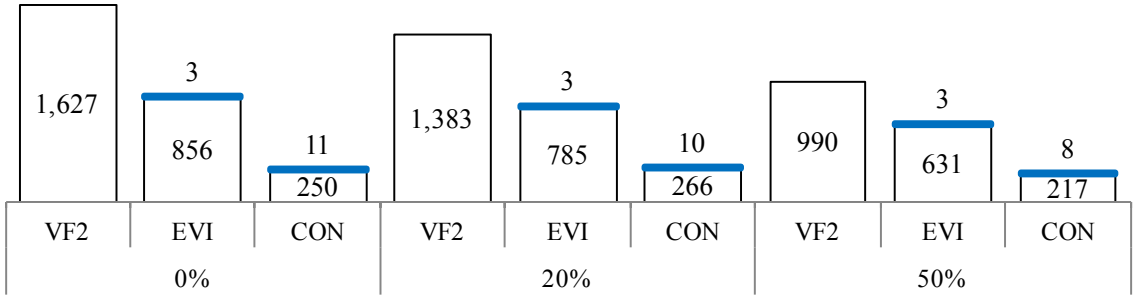
6.2.10 Query Time Break-down Analysis

Figure 6.22 depicts a break-down of query processing time for FTV methods and GraphCache, showing how much of GraphCache time is spent (on average) to update the Window and Cache data stores (including executing the cache replacement algorithms and re-indexing the cached query graphs), for various cache sizes. As we can see, **the time overhead for cache maintenance chores is trivial**. Another interesting observation is that, although increasing the size of the cache improves query processing time (as also shown in Figure 6.18), it also leads to an increase in the overhead associated with the maintenance of the cache contents. For the cache sizes considered in this work, one can see that GraphCache loses in maintenance overhead and gains in query time. The upside is, though, that **even with the meagre cache sizes used in this work, the performance gains are enough to not warrant a much larger cache**.

In turn, Figure 6.23 depicts a break-down of query processing time for method M, EVI and CON in GraphCache+ against a dynamic dataset based on AIDS. EVI pays overhead on updating the Window and Cache data stores, including executing the cache replacement algorithms and re-indexing the cached graphs. Whereas the overhead of CON also covers



(a) Type A Workloads



(b) Type B Workloads

Figure 6.23: GraphCache+: Average Execution Time and Overhead (in Millisecond) Per Query Graph

the time of analyzing dataset log and validating cache (see Algorithm 6 and 7) – such CON specific cost is trial, taking less than 1% in CON overhead, across all the aforementioned workloads and methods M. This confirms a significant conclusion – **the CON exclusive algorithms 6 and 7 are efficient.** The dominant part of CON overhead (for updating the Window and Cache data stores) is higher than that of EVI, as the latter is frequently purged hence bearing less to be updated. Putting the Figures of 6.14, 6.15 and 6.16 aside Figure 6.23, it is obvious that **CON sweeps EVI in speedup regarding both query time and number of subgraph isomorphism tests, with a negligible additional overhead**.

6.3 Summary

This chapter has presented extensive performance evaluations for the GraphCache/GraphCache+ system. It first described the configuration of experiments, regarding a number of settings such as graph datasets, query workloads, algorithm context, dataset change plan, the upper limit of cache/window size, workload distributions and etc. Performance metrics are highlighted by the speedup, which is defined as the performance gains of GraphCache/GraphCache+ against each underlying algorithm considered in this work. Such speedup

is expressed in the number of subgraph isomorphism tests and in the query processing time, which are the two major measurements in literature.

Comprehensive experiments are performed with over 6 million queries, against a number of graph datasets with different characteristics, and in the context of various algorithms, which has proven the applicability and appropriateness of our approaches. Analyzing these experiments has revealed a number of key lessons, pertaining to graph caching and query processing. GraphCache/GraphCache+ achieves considerable performance gains in the number of subgraph isomorphism tests and the query processing time, with meagre space overheads. A number of GraphCache(+) exclusive replacement strategies are tested for performance, among which the novel hybrid dynamic (HD) policy wins incontrovertibly, and the cache admission control affords further gains. Moreover, regarding the two cache models specific to GraphCache+, CON sweeps EVI in both query time and the number of subgraph isomorphism tests, with negligible additional overheads. Furthermore, performing GraphCache on top of SI is proved competitive even against state-of-the-art FTV methods, which in turn offers food for thought.

Chapter 7

Conclusions

Modern big data applications demand high performing graph query processing. Whereas recent studies [13, 14] have proved the significant performance limitation of the current research. Therefore, this work has presented a suite of algorithms and systems for ensuring high-performance graph query processing, complementing existing state of the art approaches and performing well across the board. Central to the idea is to utilize the knowledge derived from previously executed queries, rather than throwing them away.

Following the novel approach, this thesis proceeds to afford a number of contributions to the community, including the iGQ query method [92], the first full-fledge graph caching system GraphCache [95], and the first study of exploring graph cache consistency GraphCache+ [100]. This chapter shall summarize these contributions and discuss the future work, concluding the whole thesis.

7.1 Summary of Contributions

7.1.1 Contributions: Algorithms

Figure 7.1 depicts the big picture of contributions pertaining to algorithms and techniques. Each module itself consists of several microcosm components that closely collaborate, which are to be illustrated module-wise as follows.

Indexing Queries to Expedite Graph Query Processing

Recall the fresh principle regarding graph query processing, in which knowledge of previous queries is leveraged to facilitate future queries. This is the soul of iGQ query method. As to the contribution to the literature, iGQ departs from related work in three ways.

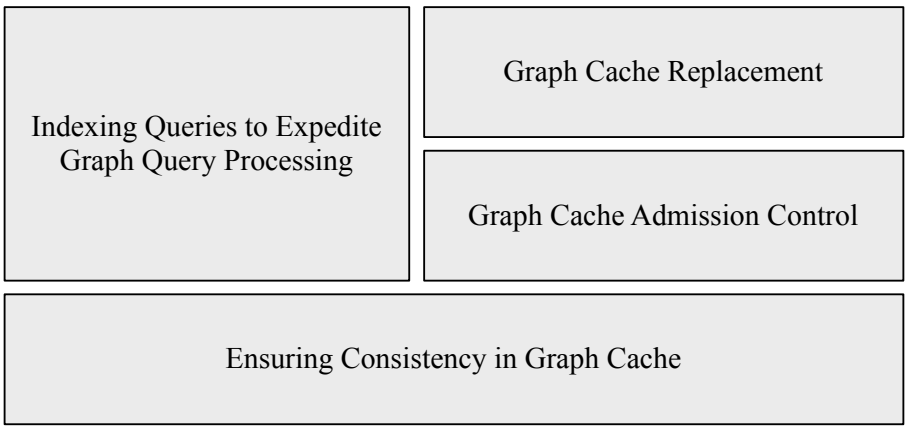


Figure 7.1: Contributions in Algorithm/Technique

- As its name, iGQ builds index for graph queries, as opposed to merely resting on the index of dataset graphs. This substantially benefit query workloads that share little characteristics of the graph dataset.
- iGQ maintains the knowledge that is laboriously and painstakingly obtained when executing previous queries, instead of throwing away blindly.
- iGQ can be utilized to expedite both subgraph queries and supergraph queries, bridging the two separate research threads and affording the elegance of killing two birds using one stone.

Moreover, iGQ offers food for thought. First, it endeavours to "mine" the containment status among queries. Besides the queries that are frequently submitted to system, iGQ manages to expedite queries that share subgraph/supergraph relationships with those executed. Second, as to the component to detect supergraph status between new and previous queries, iGQ provides a novel approach that could swiftly avoid the heavy overhead, practicing the principle of being simple yet efficient.

Graph Cache Replacement

This thesis presents GraphCache, to the best of our knowledge the first full-fledged caching system for the general subgraph/supergraph query processing. Like any caching system, GraphCache is required to properly address the problem of cache replacement. To this end, this work contributes as follows.

- It has materialized the classical caching replacement policies such as LRU and popularity based strategy in GraphCache system, exploring their applicabilities in the new setting of replacing graph query.

- It has proposed a number of GC exclusive replacement policies with different trade-offs, by well considering the particularity of graph cache. These graph query aware strategies are highlighted by a novel hybrid policy with competitive performance.

Graph Cache Admission Control

Furthermore, among the various experiments, we have discovered the phenomenon of cache pollution, in which the overall query time speedup gained by graph caching was very low despite the fact that the majority of queries were benefited. To address this problem, this work has proposed an admission control mechanism that rewards expensive queries time-wise, which manages to obtain further performance gains than that of vanilla graph cache.

Ensuring Consistency in Graph Cache

Following the established research of the community, GraphCache system deals with graph queries against a static dataset. Whereas real-world applications indicate that the underlying graph dataset could change over time. Therefore, this thesis looks into the topic of graph cache consistency.

- It presents a full-fledged system GraphCache+, which is the first study that explores the topic of graph cache consistency and manages to expedite the general subgraph/supergraph query processing against dynamic graph datasets.
- GraphCache+ is characterized by two GC+ exclusive cache models, reflecting the different designs in dealing with the consistency issues of graph cache.
- To accommodate the new setting with dynamic dataset, GraphCache+ extends the iGQ paradigm for subgraph and supergraph query processing, offering the formally proved correctness.

7.1.2 Contributions: Systems

In turn, Figure 7.2 shows the contribution of this thesis in system. GraphCache and GraphCache+, the systems afforded by this work, share the characteristic of being full-fledged. Take GraphCache for example. The following shows the thinkings that direct the design and implementation.

- Considering the well-established caching principles and the particularity of graph queries, a number of design goals are first identified.

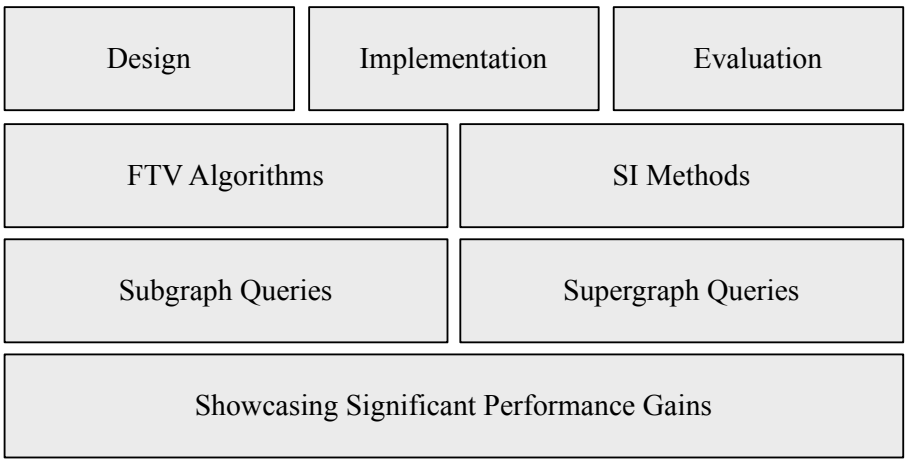


Figure 7.2: Contributions Pertaining to System

- It then follows the system architecture of GraphCache, featured by well defined subsystems and interfaces, allowing for the flexible plug-in of new components.
- The performance evaluation of GraphCache system uses millions of queries against both real-world and synthetic datasets of different characteristics, quantifying the benefits and overheads, putting forth a number of non-trivial lessons.
- GraphCache is capable of accelerating all the current FTV algorithms and SI methods.
- GraphCache offers the elegance of double use in expediting both subgraph and supergraph queries.
- Extensive experiments showcase the significant performance gains achieved by GraphCache system.

GraphCache (GraphCache+) is implemented in Java over ≈6,000 lines of code, pulling everything together to make a concrete system. GraphCache+ system is further highlighted by ensuring the consistency of graph cache, for which GraphCache+ plugs in new subsystems and components to GraphCache system, instead of starting from scratch. Such good practice of well utilizing the existing resources to address new challenges is also remarkable.

7.2 Future Work

Future work currently focuses on three big ticket items. First, it is to extend GraphCache (GraphCache+) so as to benefit subgraph queries when finding all occurrences of a query graph against a single massive stored graph. Recall the two versions of subgraph isomorphism problem. The decision problem answers Y/N as to whether the query is contained in each graph in the dataset. The matching problem locates all occurrences of the query graph

within a large graph (or a dataset of graphs). For both the decision and matching problems, the brute-force approach is to execute sub-iso tests of the query against all dataset graphs, i.e., the aforementioned SI methods. Currently, this thesis deals with the decision problem. The community has also looked into subgraph queries against a single, very large graph (consisting of possibly billions of nodes) [38, 39]. GraphCache (GraphCache+) system does not target such use cases for the time being and extending the system to benefit queries against a single massive graph is left for the future work.

Moreover, it is to develop a distributed/decentralized version of GraphCache (GraphCache+). Fundamentally, there are two central issues to figure out. On the one hand, regarding every query graph, a machine among the cluster is properly selected for execution such that the query processing could benefit from GraphCache (GraphCache+) to the utmost. On the other hand, each executed query shall be collected by suitable machines (i.e., a subset of the cluster) so as to best expedite future queries. Scaling out GraphCache (GraphCache+) could consider using the Pythia framework [101] that is originally designed to predict and engage the appropriate subset of cluster when dealing with big data missing value imputations; with respect to the particular task of graph caching, it requires relevant mechanisms to derive knowledge from the local cache stores of each machine. These should be further investigated.

Furthermore, it is worthwhile exploring the applicability and appropriateness of GraphCache (GraphCache+) for alternative data types. In principle, GraphCache (GraphCache+) is a caching system for improving query performance. Such queries could cover various data types. For data types that can be transformed into or represented as graphs and for which queries can also be posed in the form of subgraph or supergraph queries, GraphCache (GraphCache+) can obviously be used to improve query performance. Additionally, GraphCache (GraphCache+) can be used in general for data types that define commutative and transitive containment operators (e.g., “A contains B” or “A is contained in B”), and for which queries are also of the form “is Q contained in the items of the dataset D?” or “are items of the dataset D contained in Q?”. Two prominent examples are image containment queries (as found in image detection use cases) and string containment queries (often arising in biological/genome datasets). Future work should be conducted over materializing GraphCache (GraphCache+) components upon use case specialties (e.g., mechanisms for cache replacement and cache admission control so as to further optimize image queries).

Appendix A

Approximation of the Time Element for PINC and HD Strategies

As illustrated in §4.3.3, GraphCache replacement policies of PINC and HD involve an approximation for the time cost of those reduced subgraph isomorphism tests when dealing with the utility of cached graphs. Such "query time" is estimated as the corresponding sub-iso tests are not performed at all. Fortunately, regarding the backtracking subgraph isomorphism algorithms, [102] provides a framework of time complexity analysis, which is applicable for all the SI methods considered in this work. The following shall first review the perspective of [102] and then extend it to accommodate the cases of GraphCache.

Subgraph isomorphism testing occurs between two graphs, i.e., a query $\boldsymbol{g} = (\boldsymbol{N_g}, \boldsymbol{B_g})$ and a dataset graph $\boldsymbol{G} = (\boldsymbol{N_G}, \boldsymbol{B_G})$, where $\boldsymbol{N_{(.)}}$ is the node set and $\boldsymbol{B_{(.)}}$ is the edge (bonding between nodes) set. SI methods aim to discover whether there exists a mapping $\boldsymbol{M}$ that covers all nodes of query $\boldsymbol{g}$, i.e., a mapping $\boldsymbol{M} = \{(\boldsymbol{n}, \boldsymbol{m})\}$ pertaining to a set of matched node pairs $(\boldsymbol{n}, \boldsymbol{m})$ such that $\boldsymbol{n} \in \boldsymbol{N_g}$ and $\boldsymbol{m} \in \boldsymbol{N_G}$. Hence, the backtracking paradigm could be viewed as constantly adding pairs of matched nodes to partial mappings until the end result is reached (i.e., a mapping $\boldsymbol{M}$ is found or there exists no such $\boldsymbol{M}$), during which further exploration of a branch can be avoided if the addition of a node pair violates the matching criteria [102].

According to the analysis in [102], the time complexity of backtracking sub-iso algorithms are given by:

$$\mathcal{O}(\boldsymbol{N} \times \boldsymbol{N!}) \tag{A.1}$$

where N is the node number of query graph $\boldsymbol{g}$ (or dataset graph $\boldsymbol{G}$), with two assumptions as follows:

- The query $\boldsymbol{g}$ and dataset graph $\boldsymbol{G}$ have the same number of nodes.

- There is no label restriction on nodes, i.e., label information does not involve in differentiating nodes.

More specifically, [102] uses a state to represent a partial mapping solution. Therefore, analyzing the time cost of a sub-iso test turns to finding out how many states are involved and the cost pertaining to each state.

Regarding the time cost per state, [102] further decomposes the matching process into two stages. First, given a selected successor node $\boldsymbol{x}$ to be matched in query $\boldsymbol{g}$, produce the candidate node set $\{\boldsymbol{c}\}$, from those unmatched nodes in the dataset graph $\boldsymbol{G}$. Second, decide whether each candidate $\boldsymbol{c}$ satisfy the feasibility rules of subgraph isomorphism, such that if there is an edge $\boldsymbol{e}$ bridging node $\boldsymbol{x}$ and node $\boldsymbol{y}$ in query $\boldsymbol{g}$, there must exist a corresponding edge $\boldsymbol{e'}$ in dataset graph $\boldsymbol{G}$, such that $\boldsymbol{e'}$ connects node $\boldsymbol{x'}$ and node $\boldsymbol{y'}$ (in $\boldsymbol{G}$), $\boldsymbol{x'}$ matches $\boldsymbol{x}$ and $\boldsymbol{y'}$ matches $\boldsymbol{y}$. As demonstrated in [102], the first stage dominates the time cost pertaining to each state, which is given by $\mathcal{O}(\boldsymbol{N})$ where $\boldsymbol{N}$ is node number of query graph $\boldsymbol{g}$ (or dataset graph $\boldsymbol{G}$), again with the two aforementioned assumptions.

As to the number of states to compute, [102] had analyzed both the best and worst cases. The best case results the complexity of $\mathcal{O}(\boldsymbol{N})$ such that each node in query $\boldsymbol{g}$ has only to go through one state so as to determine that the query $\boldsymbol{g}$ is or is not a subgraph of the dataset graph $\boldsymbol{G}$. Whereas the worst case renders $\mathcal{O}(\boldsymbol{N!})$, in which nodes of query $\boldsymbol{g}$ have to undergo all possible states to finish the matching process in the end. Still, $\boldsymbol{N}$ is node number of query graph $\boldsymbol{g}$ (or dataset graph $\boldsymbol{G}$), resting on the said assumptions.

Table A.1: Number of States on Each Level in the Worst Case

level	**number of states to explore**
0	N
1	N × (N-1)
2	N × (N-1) × (N-2)
3	N × (N-1) × (N-2) × (N-3)
4	N × (N-1) × (N-2) × (N-3) × (N-4)
5	N × (N-1) × (N-2) × (N-3) × (N-4) × (N-5)
...	...
...	...
N-1	N × (N-1) × (N-2) × (N-3) × (N-4) × (N-5) × ... × 2 × 1

[102] emphasizes the worst case for generality and concludes the formula as (A.1). Table A.1 details the number of states to go through of the worst case. On each level, a node $\boldsymbol{x}$ in query graph $\boldsymbol{g}$ will undergo states in maximum. For example, $\boldsymbol{N}$ states on **level 0** indicates that there are $\boldsymbol{N}$ candidate nodes in the dataset graph to match the first node of query graph $\boldsymbol{g}$. Then, $\boldsymbol{N-1}$ nodes are remained maximumly for the second node of query graph $\boldsymbol{q}$ on the

level 1. Such process continues until all the nodes in query graph are covered. It is apparent that **level N-1** dominates, which in turn provides the complexity for the number of states to compute:

$$\boldsymbol{N \times (N-1) \times (N-2) \times ... \times 2 \times 1 = \mathcal{O}(N!)} \tag{A.2}$$

Turning attention to GraphCache. As is common in literature [16, 42, 13], the settings of sub-iso tests in GraphCache are featured by two characteristics.

- The dataset graph $\boldsymbol{G}$ is typically much larger than the query $\boldsymbol{g}$, containing more nodes.
- A graph dataset usually consists of several unique labels. Each graph node possesses a label, such that two nodes can never be matched unless sharing the same label.

Interestingly, such settings for sub-iso testing turn off the assumptions as stated in [102]. Therefore, to better estimate the time element used in PINC and HD strategies, it requires a more general approach. Based on the formula (A.1) provided by [102], this work shall proceed to fit the use case of GraphCache from two aspects, i.e., "enlarging" the dataset graph and "labeling" each graph node.

A.1 Enlarging the Dataset Graph

Following the analysis in [102], the extension of enlarging the dataset graph considers both the time cost per state and the total number of states. Say, the dataset graph $\boldsymbol{G}$ has $\boldsymbol{N}$ nodes whereas the query $\boldsymbol{g}$ has $\boldsymbol{n}$ nodes, such that $\boldsymbol{N > n}$. Regarding the time cost per state, the conclusion of $\mathcal{O}(N)$ is still applicable. However, pertaining to the total number of states, things are different, as illustrated by Table A.2.

Table A.2: Number of States Per Level when Dataset Graph is Larger than Query

level	number of states to explore
0	N
1	N × (N-1)
2	N × (N-1) × (N-2)
3	N × (N-1) × (N-2) × (N-3)
4	N × (N-1) × (N-2) × (N-3) × (N-4)
5	N × (N-1) × (N-2) × (N-3) × (N-4) × (N-5)
...	...
...	...
n-1	N × (N-1) × (N-2) × (N-3) × (N-4) × (N-5) × ... × (N-n+1)

Among all the levels, again, **level n-1** dominates. Hence returns the complexity for the total number of states to compute during a subgraph isomorphism testing, which is:

$$N \times (N-1) \times (N-2) \times ... \times (N-n+1) = \mathcal{O}(\frac{N!}{(N-n)!}) \tag{A.3}$$

As a result, when dealing with the case of larger dataset graph than query, the time complexity of subgraph isomorphism testing is given by:

$$\mathcal{O}(\frac{N \times N!}{(N-n)!}) \tag{A.4}$$

where N and n are the node number of dataset graph G and query g, respectively.

A.2 Labeling Each Graph Node

Based on formula (A.4), a further extension is to take the label information of graph node into consideration. Given a query graph g with n nodes and a dataset graph G with N nodes, say, there are L unique labels (label varieties) across the dataset. For simplicity, it assumes that labels distribute uniformly among graph nodes – for a specific node x in the query g, a node x' in the dataset graph G possesses the following probability of sharing the same label as x.

$$\frac{1}{L} * 100\% \tag{A.5}$$

Acting as a grouping criteria, node label impacts the complexity of both the time cost per state and the total number of states. The former complexity is adapted as follows:

$$\mathcal{O}(\frac{N}{L}) \tag{A.6}$$

This lies in the fact that while creating the candidate node set $\{c\}$ in dataset graph G regarding a given query node x, only those bearing the same label as that of x will be considered for further exploration.

As to the latter element, i.e., the total number of states, Table A.3 shows the details. Again, **level n-1** dominates; hence the complexity regarding the total number of states is given by:

$$\frac{N}{L} \times \frac{(N-1)}{L} \times \frac{(N-2)}{L} \times \frac{(N-3)}{L} \times ... \times \frac{(N-n+1)}{L} = \mathcal{O}(\frac{N!}{L^n \times (N-n)!}) \tag{A.7}$$

In the end, compared with [102], a more general time complexity for backtracking sub-iso

test algorithms is resulted:

$$\mathcal{O}(\frac{\boldsymbol{N} \times \boldsymbol{N!}}{\boldsymbol{L^{n+1}} \times \boldsymbol{(N-n)!}}) \tag{A.8}$$

where $\boldsymbol{n}$ is the node number of query $\boldsymbol{g}$, $\boldsymbol{N}$ is the node number of dataset graph $\boldsymbol{G}$, and $\boldsymbol{L}$ is the number of unique labels in the dataset.

Table A.3: Number of States Per Level with Node Label and Larger Dataset Graph

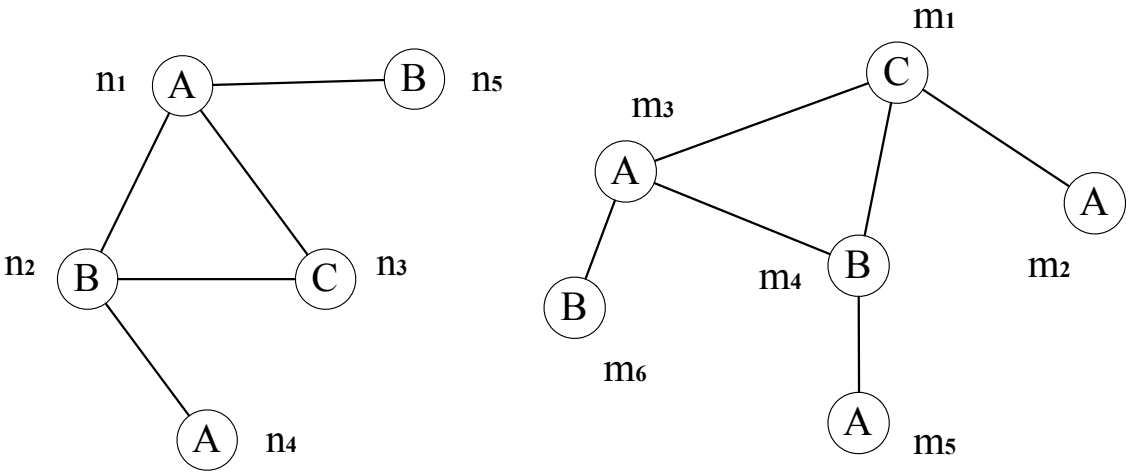
level	number of states to explore
0	$\frac{\boldsymbol{N}}{\boldsymbol{L}}$
1	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}}$
2	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-2)}}{\boldsymbol{L}}$
3	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-2)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-3)}}{\boldsymbol{L}}$
4	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-2)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-3)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-4)}}{\boldsymbol{L}}$
5	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-2)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-3)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-4)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-5)}}{\boldsymbol{L}}$
...	...
...	...
n-1	$\frac{\boldsymbol{N}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-1)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-2)}}{\boldsymbol{L}} \times \frac{\boldsymbol{(N-3)}}{\boldsymbol{L}} \times \boldsymbol{...} \times \frac{\boldsymbol{(N-n+1)}}{\boldsymbol{L}}$

A.3 An Example of the Use Case in GraphCache

This section shall present an example of sub-iso testing in GraphCache, showcasing the matching process with the general settings such that the dataset graph is larger than the query and each graph node is associated with a label.

Figure A.1 depicts a pair of graphs to undergo the subgraph isomorphism verification, i.e., testing whether the query $\boldsymbol{g}$ is a subgraph of the dataset graph $\boldsymbol{G}$. For simplicity, the dataset

graph G is allocated only one more node than the query g in Figure A.1, where the node label is described by a capital letter and the tag beside each node represents the node ID.



(a) Query g (b) Dataset Graph G

Figure A.1: An Example of Query g and Dataset Graph G.

Table A.4 shows the overall matching process, where **level (step)** pertains to a transition, i.e., transferring from state s to state s' by adding a pair of matched nodes. The **mapping** M represents the partial mapping at the moment and n is a node in query g, which is going to be matched on the current level.

As to the dataset graph G, **candidate of** m covers the possible matches, i.e., those sharing the same label as node n (semantic comparison). And m are the matched nodes of n, i.e., verified results of **candidate of** m, by passing the structural feasibility: on level k, if the node n has an edge with node n_i in query g, where n_i in $M_{k.()}$, node m must have a corresponding edge with a node m_j in the dataset graph G such that (n_i, m_j) in $M_{k.()}$ (syntactic comparison).

Finally, s refers to the start state of the current transition and s' in turn is for the end state. As stated by equation (A.9), the mapping M_{51} covers all nodes of the query g, i.e., $\{n_1, n_2, n_3, n_4, n_5\}$. Hence, the matching process is terminated, returning one match of query g and a subgraph of the dataset graph G (with nodes $\{m_3, m_4, m_1, m_5, m_6\}$ and the bonding edges in accordance with those in the query g). As a result, in this example, the query g is a subgraph of the dataset graph G.

$$M_{51} = \{(n_1, m_3), (n_2, m_4), (n_3, m_1), (n_4, m_5), (n_5, m_6)\} \tag{A.9}$$

Table A.4: Matching Process of Subgraph Isomorphism Test

level (step)	**mapping M**	n	**candidate of m**	m	s	s'
0	$M_0 = \emptyset$	n_1	$\{m_2, m_3, m_5\}$	$\{m_2, m_3, m_5\}$	M_0	$\{M_{11}, M_{12}, M_{13}\}$
	$M_{11} = \{(n_1, m_2)\}$	n_2	$\{m_4, m_6\}$	$\emptyset$	M_{11}	$\emptyset$
1	$M_{12} = \{(n_1, m_3)\}$	n_2	$\{m_4, m_6\}$	$\{m_4, m_6\}$	M_{12}	$\{M_{21}, M_{22}\}$
	$M_{13} = \{(n_1, m_5)\}$	n_2	$\{m_4, m_6\}$	$\{m_4\}$	M_{13}	$\{M_{23}\}$
	$M_{21} = \{(n_1, m_3), (n_2, m_4)\}$	n_3	$\{m_1\}$	$\{m_1\}$	M_{21}	$\{M_{31}\}$
2	$M_{22} = \{(n_1, m_3), (n_2, m_6)\}$	n_3	$\{m_1\}$	$\emptyset$	M_{22}	$\emptyset$
	$M_{23} = \{(n_1, m_5), (n2, m4)\}$	n_3	$\{m_1\}$	$\emptyset$	M_{23}	$\emptyset$
3	$M_{31} = \{(n_1, m_3), (n_2, m_4), (n_3, m_1)\}$	n_4	$\{m_2, m_5\}$	$\{m_5\}$	M_{31}	$\{M_{41}\}$
4	$M_{41} = \{(n_1, m_3), (n_2, m_4), (n_3, m_1), (n_4, m_5)\}$	n_5	$\{m_6\}$	$\{m_6\}$	M_{41}	$\{M_{51}\}$

A.4 Summary

By considering the common settings of subgraph isomorphism testing, this appendix affords a more general estimation of the time cost for backtracking sub-iso algorithms than that of [102]. Indeed, it is a straightforward extension and is not that comprehensive to cover sophisticated elements such as the variance of graph structures. But it does provide a practical solution in estimating the time cost of those "unperformed" sub-iso tests for GraphCache, where the cache replacement rests on the relative rankings instead of the absolute values, like any caching system.

Bibliography

[1] PubChem, "https://pubchem.ncbi.nlm.nih.gov/."

[2] K. Degtyarenko, J. Hastings, P. de Matos, and M. Ennis, "ChEBI: An open bioinformatics and cheminformatics resource," *Curr. Protoc. Bioinformatics*, vol. 14, no. 26, pp. 1–20, 2009.

[3] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, 2005, pp. 286–295.

[4] E. Petras and C. Faloutsos, "Similarity Searching in Medical Image Databases," *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, vol. 9, no. 3, pp. 435–447, 1997.

[5] R. V. Bruggen, *Learning Neo4j*. O'Reilly Media, 2013.

[6] InfiniteGraph, "http://www.objectivity.com/infinitegraph."

[7] Twitter FlockDB, "https://github.com/twitter/flockdb."

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10)*, 2010, pp. 135–146.

[9] J. L. Gross and J. Yellen, *Graph Theory and Its Applications*, 2nd ed. Chapman and Hall/CRC, 2006.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[11] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 2, pp. 133–144, 2012.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[13] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "iGraph: A framework for comparisons of disk-based graph indexing techniques," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 449–459, 2010.

[14] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Performance and scalability of indexed subgraph query processing methods," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1566–1577, 2015.

[15] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[16] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha, "Enhancing graph database indexing by suffix tree structure," in *Proceedings of the 5th IAPR international conference on Pattern recognition in bioinformatics (PRIB'10)*, 2010, pp. 195–203.

[17] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, 2008, pp. 405–418.

[18] K. Klein, N. Kriege, and P. Mutzel, "CT-index: Fingerprint-based graph indexing combining cycles and trees," in *IEEE 27th International Conference on Data Engineering (ICDE)*, 2011, pp. 1115–1126.

[19] L. W. Beineke and R. J. Wilson, *Topics in Structural Graph Theory*. Cambridge University Press, 2013.

[20] B. D. McKay, "Practical Graph Isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[21] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, no. 0, pp. 94–112, 2014.

[22] P. T. Darga, K. A. Sakallah, and I. L. Markov, "Faster Symmetry Discovery using Sparsity of Symmetries," in *Proceedings of the 45th Design Automation Conference (DAC)*, 2008, pp. 149–154.

[23] H. Katebi, K. A. Sakallah, and I. L. Markov, "Symmetry and Satisfiability: An Update," in *Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing (SAT'10)*, 2010, pp. 113–127.

[24] T. Junttila and P. Kaski, "Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs," in *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007, pp. 135–149.

[25] J. L. Lopez-Presa and A. F. Anta, "Fast algorithm for graph isomorphism testing," in *The 8th International Symposium on Experimental Algorithms (SEA)*, 2009.

[26] D. C. Porumbel, "Isomorphism Testing via Polynomial-Time Graph Extensions," *Journal of Mathematical Modelling and Algorithms*, vol. 10, pp. 119–143, 2011.

[27] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry," *IEEE Transactions on Computer Aided Design*, vol. 22, no. 9, pp. 1117–1137, 2003.

[28] F. A. Aloul, K. A. Sakallah, and I. L. Markov, "Efficient Symmetry Breaking for Boolean Satisfiability," in *Proceedings of the 18th international joint conference on Artificial intelligence (IJCAI'03)*, 2003, pp. 271–276.

[29] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Symmetry Breaking for Pseudo-Boolean Formulas," *Journal of Experimental Algorithmics (JEA)*, vol. 12, no. 1.3, 2008.

[30] Graph Isomorphism, "http://people.cs.uchicago.edu/~laci/update.html."

[31] N. J. Nilsson, *Principles of Artificial Intelligence*. Morgan Kaufmann, 1982.

[32] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism," *Proceedings of the VLDB Endowment (PVLDB)*, pp. 364–375, 2008.

[33] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo$_{ISO}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, 2013, pp. 337–348.

[34] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient Subgraph Matching by Postponing Cartesian Products," *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, pp. 1199–1214, 2016.

[35] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, pp. 340–351, 2010.

[36] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 5, pp. 617–628, 2015.

[37] S. Zhang, S. Li, and J. Yang, "GADDI: Distance Index based Subgraph Matching in Biological Networks," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, 2009, pp. 192–203.

[38] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in MapReduce," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 10, pp. 974–985, 2015.

[39] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 9, pp. 788–799, 2012.

[40] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*, 2004, pp. 335–346.

[41] S. Zhang, M. Hu, and J. Yang, "TreePi: A Novel Graph Indexing Method," in *IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007.

[42] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha, "GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures," *PloS One*, vol. 8, no. 10, p. e76911, 2013.

[43] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu, "Towards graph containment search and indexing," in *Proceedings of the 33rd international conference on Very large data bases (Proc. VLDB)*, 2007, pp. 926–937.

[44] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang, "Prefindex : An efficient supergraph containment search technique," in *Proceedings of the 22nd international conference on Scientific and statistical database management (SSDBM'10)*, 2010, pp. 360–378.

[45] S. Zhang, J. Li, H. Gao, and Z. Zou, "A novel approach for efficient supergraph query processing on graph databases," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, 2009, pp. 204–215.

[46] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu, "Fast graph query processing with a low-cost index," *The International Journal on Very Large Data Bases (VLDBJ)*, vol. 20, no. 4, pp. 521–539, 2010.

[47] Y. Zhu, J. X. Yu, and L. Qin, "Leveraging graph dimensions in online graph search," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 1, pp. 85–96, 2014.

[48] J. Cheng, Y. Ke, W. Ng, and A. Lu, "FG-index: Towards verification-free query processing on graph databases," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, 2007, pp. 857–872.

[49] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree + delta >= graph," in *Proceedings of the 33rd international conference on Very large data bases (Proc. VLDB)*, 2007, pp. 938–949.

[50] D. Yuan and P. Mitra, "Lindex: a lattice-based index for graph databases," *The International Journal on Very Large Data Bases (VLDBJ)*, vol. 22, no. 2, pp. 229–252, 2013.

[51] D. Yuan, P. Mitra, and C. Giles, "Mining and indexing graphs for supergraph search," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 10, pp. 829–840, 2013.

[52] R. Di Natale *et al.*, "Sing: Subgraph search in non-homogeneous graphs," *BMC Bioinformatics*, vol. 11, no. 96, 2010.

[53] R. Giugno and D. Shasha, "GraphGrep: A fast and universal method for querying graphs," in *16th International Conference on Pattern Recognition (ICPR 2002)*, 2002.

[54] L. Zou, L. Chen, J. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology (EDBT '08)*, 2008, pp. 181–192.

[55] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 541–552.

[56] H. He and A. K. Singh, "Closure-Tree: An Index Structure for Graph Queries," in *22nd International Conference on Data Engineering (ICDE)*, 2006.

[57] Y. Tian and J. M. Patel, "TALE: A Tool for Approximate Large Graph Matching," in *IEEE 24th International Conference on Data Engineering (ICDE)*, 2008.

[58] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007.

[59] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Feature-based similarity search in graph structures," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 4, pp. 1418–1453, 2006.

[60] S. Zhang, J. Yang, and W. Jin, " SAPPER: subgraph indexing and approximate matching in large graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 1185–1194, 2010.

[61] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, "Efficient processing of graph similarity queries with edit distance constraints," *The International Journal on Very Large Data Bases (VLDBJ)*, vol. 22, no. 6, pp. 727–752, 2013.

[62] T. Plantenga, "Inexact subgraph isomorphism in MapReduce," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 164–175, 2013.

[63] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: Frequent subgraph and pattern mining in a single large graph," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 7, pp. 517–528, 2014.

[64] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of Computer-Aided Molecular Design*, vol. 16, pp. 521–533, 2002.

[65] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, pp. 2539–2561, 2011.

[66] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph Kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.

[67] T. Gartner, *Kernels for Structured Data*. World Scientific, 2008.

[68] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Proceedings of the 20th International Conference on Very Large Data Bases (Proc. VLDB)*, pp. 487–499, 1994.

[69] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," *Proc. of the 2001 IEEE International Conference on Data Mining, San Jose, California*, pp. 313–320, 2001.

[70] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *IEEE International Conference on Data Mining (ICDM)*, 2002, pp. 721–724.

[71] S. Nijssen and N. J. Kok, "A quickstart in frequent structure mining can make a difference," *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '04)*, pp. 647–652, 2004.

[72] A. Balmin, F. Ozcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh, "A framework for using materialized XPath views in XML query processing," in *Proceedings of the Thirtieth international conference on Very large data bases (Proc. VLDB)*, 2004, pp. 60–71.

[73] K. Lillis and E. Pitoura, "Cooperative XPath caching," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, 2008, pp. 327–338.

[74] B. Mandhani and D. Suciu, "Query caching and view selection for XML databases," in *Proceedings of the 31st international conference on Very large data bases (Proc. VLDB)*, 2005, pp. 469–480.

[75] L. V. S. Lakshmanan, H. Wang, and Z. Zhao, "Answering tree pattern queries using views," in *Proceedings of the 32nd international conference on Very large data bases (Proc. VLDB)*, 2006, pp. 571–582.

[76] J. Wang, J. Li, and J. X. Yu, "Answering tree pattern queries using views: a revisit," in *Proceedings of the 14th International Conference on Extending Database Technology (EDBT '11)*, 2011, pp. 153–164.

[77] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with SPARQL query caching," in *Proceedings of the 7th international conference on The Semantic Web: research and Applications (ESWC'10)*, 2010, pp. 304–318.

[78] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware , workload-adaptive SPARQL query caching," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015, pp. 1777–1792.

[79] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, 2008, pp. 539–552.

[80] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming subgraph isomorphism for RDF query processing," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 11, pp. 1238–1249, 2015.

[81] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries," in *Proceedings of the 17th International Conference on Extending Database Technology (EDBT '14)*, 2014, pp. 439–450.

[82] G. Koloniari and E. Pitoura, "Partial view selection for evolving social graphs," in *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*, 2013.

[83] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: How to say "good enough" in SQL," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*, 2004, pp. 815–826.

[84] S. Bottcher, "Cache consistency in mobile XML databases," in *Proceedings of the 7th international conference on Advances in Web-Age Information Management (WAIM '06)*, 2006, pp. 300–312.

[85] J. Lorey and F. Naumann, "Caching and prefetching strategies for SPARQL queries," in *The Semantic Web: ESWC 2013 Satellite Events.*, P. Cimiano, M. Fernández, V. Lopez, S. Schlobach, and J. Völker, Eds. Springer, Berlin, Heidelberg, 2013, vol. 7955, pp. 46–65.

[86] NCI - DTP AIDS antiviral screen dataset, "http://dtp.nci.nih.gov/docs/aids/aids_data.html."

[87] Y. He, F. Lin, P. R. Chipman, C. M. Bator, T. S. Baker, M. Shoham, R. J. Kuhn, M. E. Medof, and M. G. Rossmann, "Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 99, pp. 10 325–10 329, 2002.

[88] Memcached, "https://memcached.org/."

[89] LinkedIn: World's Largest Professional Network, "https://www.linkedin.com/."

[90] Facebook, "https://www.facebook.com/."

[91] C. Vehlow, H. Stehr, M. Winkelmann, J. M. Duarte, L. Petzold, J. Dinse, and M. Lappe, "CMView: Interactive contact map visualization and analysis," *Bioinformatics*, vol. 27, no. 11, pp. 1573–1577, 2011.

[92] J. Wang, N. Ntarmos, and P. Triantafillou, "Indexing query graphs to speedup graph query processing," in *Proceedings of the 19th International Conference on Extending Database Technology (EDBT '16)*, 2016, pp. 41–52.

[93] M. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, 2005.

[94] R. Nelson, *Probability, Stochastic Processes, and Queueing Theory*. Springer Verlag, 1995.

[95] J. Wang, N. Ntarmos, and P. Triantafillou, "GraphCache: a caching system for graph queries," in *Proceedings of the 20th International Conference on Extending Database Technology (EDBT '17)*, 2017.

[96] J. Handy, *The Cache Memory Book*, 2nd ed. Academic Press, 1998.

[97] National Rail Enquiries, "http://www.nationalrail.co.uk/."

[98] The Big Bang Theory, "https://en.wikipedia.org/wiki/The_Big_Bang_Theory."

[99] WeChat, "https://weixin.qq.com/."

[100] J. Wang, N. Ntarmos, and P. Triantafillou, "Ensuring consistency in graph cache for graph-pattern queries," in *Sixth International Workshop on Querying Graph Structured Data (GraphQ 2017), with EDBT '17*, 2017.

[101] C. Anagnostopoulos and P. Triantafillou, "Scaling out big data missing value imputations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '14)*, 2014, pp. 651–660.

[102] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "Subgraph transformation for the inexact matching of attributed relational graphs," *Graph Based Representations in Pattern Recognition Computing Supplement*, vol. 12, pp. 43–52, 1998.