GC: A Graph Caching System for Subgraph/Supergraph Queries

Jing Wang[1,*], Zichen Liu[2], Shuai Ma[1,*], Nikos Ntarmos[3], and Peter Triantafillou[4]
[1]BDBC, Beihang University [2]ICT, Chinese Academy of Sciences [3] University of Glasgow [4] University of Warwick
[*]{j.wang.7,mashuai}@buaa.edu.cn,liuzichen@ict.ac.cn,nikos.ntarmos@glasgow.ac.uk,p.triantafillou@warwick.ac.uk

ABSTRACT

We demonstrate a graph caching system GC for expediting subgraph/supergraph queries, which are computationally expensive due to the entailed NP-Complete subgraph isomorphism problem. Unlike existing caching systems for fast data access where each cache hit saves one disk I/O, GC reduces the computational costs due to subgraph isomorphism testing. Moreover, GC harnesses both subgraph and supergraph cache hits, extending the traditional exact-match-only hit, thus resulting in significant speedups. Furthermore, GC features dashboards for both skilled developers and general end-users; the former could investigate and experiment with alternative components/mechanisms while the latter could look into the principle of GC through a number of demonstration scenarios.

PVLDB Reference Format:
Jing Wang, Zichen Liu, Shuai Ma, Nikos Ntarmos and Peter Triantafillou. GC: A Graph Caching System for Subgraph/Supergraph Queries. *PVLDB*, 11 (12): 2022-2025, 2018.
DOI: https://doi.org/10.14778/3229863.3236250

1. INTRODUCTION

Modern graph applications in chemistry, bioinformatics, and social networking demand systems with high performance graph analytics, in which a central task is to locate patterns in dataset graphs. For example, there are prevalent needs pertaining to finding similarities for chemical compounds, computer-aided design of electronic circuits, software debugging, etc. These involve performing subgraph/ supergraph queries. Informally, given a query (pattern) graph g and a graph dataset, the system is called to return the set of dataset graphs that contain g (subgraph query) or are contained in g (supergraph query), namely the *answer set* of g. This entails the problem of subgraph isomorphism. As typical in the literature, we focus on non-induced subgraph isomorphism for undirected labelled graphs where only vertices have labels; all our results straightforwardly generalize to directed graphs and/or graphs with edge labels. The challenge lies in the NP-Completeness nature of subgraph isomorphism (abbreviated as *sub-iso* or SI in this work), which makes subgraph/supergraph query processing computationally expensive.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236250

To this end, the community has continuously put forth solutions that usually fall into two categories, i.e., SI algorithms [3] and FTV ("filter-then-verify") methods [1]. The former use heuristics so as to improve the efficiency of sub-iso tests per se. Whereas the latter employ an index of dataset graphs to filter out graphs that are definitely not in the query's *answer set*; the remaining graphs, coined *candidate set* of g then are verified for sub-iso. Indeed, FTV solutions advance by including an extra filtering stage than that of standard SI algorithms, such that fewer graphs are sub-iso tested and the query processing time is reduced. However, FTV still ends up executing unnecessary sub-iso tests; think of the example when a query is resubmitted to the system, it shall be processed from scratch. Hence it gives rise to a significant drawback of approaches in the literature: they fail to exploit the knowledge of executed queries.

Interestingly, many real-world applications indicate that graph queries submitted in the past share subgraph or supergraph relationships with future queries. Biochemical queries could range from simple molecules and aminoacids to complex proteins of multi-cell organism. Also, social networking queries may start off broad (e.g., all the people in a geographic location) and become narrower (e.g., those having specific demographics). Recall the computationally expensive graph queries; a straightforward solution for acceleration is using cache, by which GC [11] is motivated.

Indeed, caching result has been studied for graph structured queries. [8] contributed a cache to optimize SPARQL queries over RDF graphs, in which a canonical labelling scheme is proposed to identify cache hits. However, the way cache contents are exploited in GC differs from that in [8]. GC discovers subgraph, supergraph, and exact-match relationships between a new query and the cached queries [11], which the canonical labelling scheme in [8] fails to achieve.

The community has also looked into subgraph queries against a single large graph with billions of nodes [9]. And there exist distributed systems for general graph computations [6, 5]. GC does not target such use cases for the time being and extending our system to benefit queries against a single massive graph or distributed operation is left for future work. On a related note, [4] presents a solution of optimizing graph queries through "merging" materialized views (subgraphs and their results). Such optimization is similar as one of the several cases in GC; however, [4] does not deal with central issues of a caching system (cache replacement, admission control, overall architecture/design, etc.).

Underpinned by [10], GC[11] is applicable for both SI and FTV approaches, offering detailed discussions of design and implementation, resource management (memory and threads) and dynamic management of the cache index. GC rests on two processing components to discover whether the coming query is a subgraph/supergraph of cached queries (i.e., sub/super case); these constitute graph cache hits, allowing for expedited query processing by leveraging the knowledge from cached results.

Contributions. To the best of our knowledge, this work is the first demonstration in literature for showcasing key techniques inside graph cache pertaining to expediting subgraph/supergraph queries. Via a number of scenarios, audiences could look into GC characteristics that depart from existing caching systems. **Problem:** 1) GC targets at expediting subgraph/supergraph queries against a set of data graphs and is capable of dealing with the NP-Complete problem of subgraph isomorphism. 2) GC does not produce any false negative or false positive (see formal proof of correctness in [10]). 3) By using over 6 million queries, GC is attested performing impressively with speedups in query time up to 40× [11]. **Techniques:** 1) GC is designed as a pluggable cache, allowing any future component to be incorporated (SI/ FTV methods, replacement policies, scenarios, etc.). 2) Existing caching systems (e.g. caches in Neo4j [2]) usually focus on fast data access such that each cache hit saves one disk I/O, whereas GC aims to reduce costs associated with executing sub-iso tests; cases in GC are more complicated - each cache hit shall evoke various numbers of savings in sub-iso testing, which could in turn render quite different query times; GC is thus bundled with a number of exclusive replacement strategies that are graph query aware and workload adaptive. 3) Central to GC is a semantic graph cache [11] that could harness both subgraph and supergraph cache hits, extending the traditional exact-match-only hit and hence leading to impressive speedups. **Demonstration:** 1) As a non-trivial extension of the GC kernel in [11], this work is characterized by embedding dashboards for audiences with varying programming skills, ranging from professional developers to general end-users. 2) For developers, GC provides programming interfaces such that alternative components/mechanisms could be investigated. 3) For general end-users, GC employs approachable demonstrations to enhance the audience experience, including two scenarios - *The Query Journey* interprets the computations inside GC and its principle of accelerating queries; *The Workload Run* presents the query processing of a workload, cache hits and cache replacement with various policies; users are closely involved and properly guided in both scenarios; furthermore, users could even create and run their own workloads. 4) GC offers automatic visualization for graphs, delivering useful functionality for applications in chemistry, bioinformatics, and social networking.

2. SYSTEM OVERVIEW

GC offers a graph caching system, the kernel of which is a scalable semantic cache [11] for expediting subgraph/supergraph queries. Figure 1 presents the system architecture of GC, including three subsystems: Kernel, Dashboard Manager and Demonstrator. The last two are specific to the current work, responsible for showcasing the principle and techniques of GC.

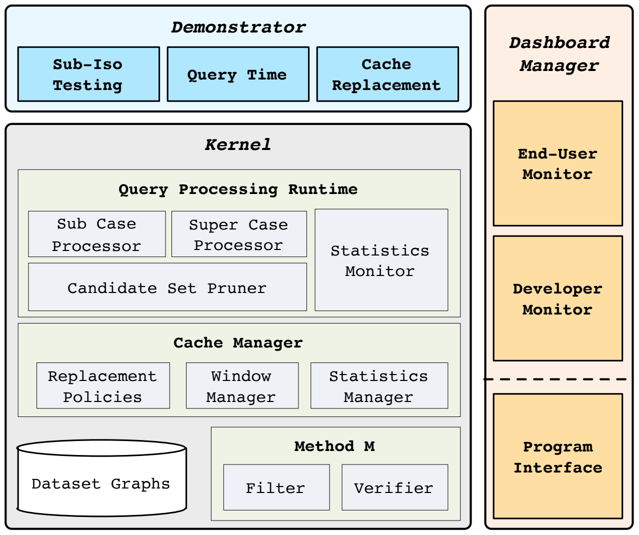


Figure 1: GC System Architecture

The GC Kernel [11] consists of several components: 1) Dataset Graphs cover those underlying graphs over which queries are executed; 2) Method M could incorporate any FTV or SI method for acceleration and hence is embedded with a filtering component Filter in addition to a sub-iso test implementation Verifier; 3) Query Processing Runtime deals with the execution of queries, including the Sub/Super Case Processor for detecting cache hits, the Candidate Set Pruner to reduce the number of sub-iso tests and a Statistics Monitor for the monitoring of key operational metrics; 4) Cache Manager is responsible for the management of data and metadata stored in the cache, comprising Replacement Policies dealing with graph cache replacement, Window Manager for cache admission control and Statistics Manager.

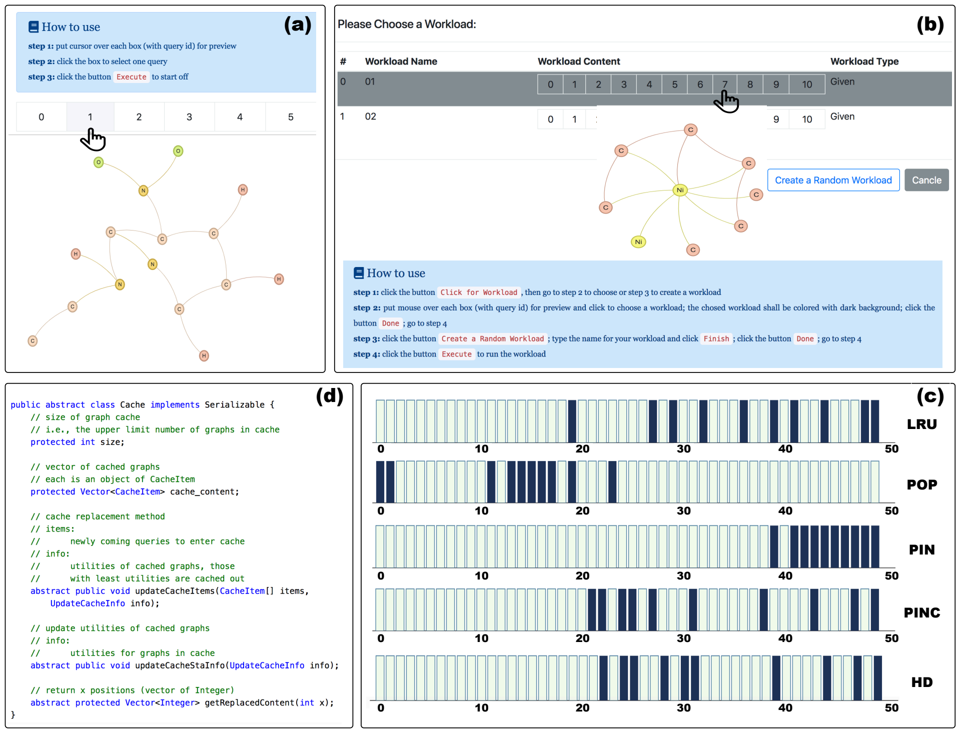
The Dashboard Manager handles the interaction with GC audiences. 1) Three major components include the End-User Monitor, the Developer Monitor and the Program Interface, catering to the needs of audiences with varying programming skills. 2) To enhance the audience experience for general end-users, GC offers a number of demonstration scenarios. 3) Through Program Interface, developers could enjoy extending/optimizing GC.

The Demonstrator presents GC performance in terms of Sub-Iso Testing, Query Time and Cache Replacement. We report query time and number of sub-iso tests, along with the speedups introduced by GC. 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 GC when deployed over Method M (i.e., speedups >1 indicate improvements).

Remark. GC per se could be plugged into general graph systems as a library, allowing for future extensions.

3. DEMONSTRATION

For ease of demonstration, GC is deployed over Linode Cloud with 2GB memory and bundled (but not limited) with 100 real-world graphs from AIDS dataset [7] (the Antiviral Screen Dataset for topological structures of molecules), workloads each with 10 queries (generated from graphs in dataset following established principles), graph cache with



How to use

step 1: put cursor over each box (with query id) for preview

step 2: click the box to select one query

step 3: click the button `Execute` to start off

0	1	2	3	4	5

Please Choose a Workload:

#	Workload Name	Workload Content	Workload Type
0	01	0 1 2 3 4 5 6 7 8 9 10	Given
1	02	0 1 : 9 10	Given

How to use

step 1: click the button `Click for Workload`, then go to step 2 to choose or step 3 to create a workload

step 2: put mouse over each box (with query id) for preview and click to choose a workload; the chosed workload shall be colored with dark background; click the button `Done`; go to step 4

step 3: click the button `Create a Random Workload`; type the name for your workload and click `Finish`; click the button `Done`; go to step 4

step 4: click the button `Execute` to run the workload

```
public abstract class Cache implements Serializable {
    // size of graph cache
    // i.e., the upper limit number of graphs in cache
    protected int size;

    // vector of cached graphs
    // each is an object of CacheItem
    protected Vector<CacheItem> cache_content;

    // cache replacement method
    // items:
    //      newly coming queries to enter cache
    // info:
    //      utilities of cached graphs, those
    //      with least utilities are cached out
    abstract public void updateCacheItems(CacheItem[] items,
        UpdateCacheInfo info);

    // update utilities of cached graphs
    // info:
    //      utilities for graphs in cache
    abstract public void updateCacheStaInfo(UpdateCacheInfo info);

    // return x positions (vector of Integer)
    abstract protected Vector<Integer> getReplacedContent(int x);
}
```

Figure 2: GC Interfaces for The Query Journey, The Workload Run, Cache Replacement and API

50 executed queries and Method M in [1]. We have implemented all aforementioned GC components in Java, using HTML and JavaScript (with WebSocket) for front-end. Online demonstration of GC is available [1].

Next, we shall first overview GC performance in [11] and then take a walk through GC as end-users and developers.

3.1 Performance Comparison

The performance evaluation of GC employs over 6 million queries, both real-world and synthetic graph datasets with various characteristics, and a number of Methods M.

I: Competition Among Various Policies. GC offers a number of graph cache replacement policies with different trade-offs, including: 1) the well established policy LRU; 2) the popularity based strategy POP; 3) PIN and PINC where graph utilities go down to the level of sub-iso test numbers and sub-iso testing costs, respectively; 4) HD that coalesces both PIN and PINC. We have observed through a large number of experiments that different cache replacement policies take the lead depending on the workload and dataset characteristics. Here comes the question: how to choose a replacement policy when said characteristics are unknown a-priori? Our answer to this question and the first takeaway message is: ***When in doubt, use the HD replacement policy***, as it is attested performing better or on par with the best alternative.

II: Speedup versus Overhead. Recall that FTV algorithm rests on a dataset index that is built upon graph features (feature is the sub-structure of graph, e.g., a path, tree or subgraph) and sub-iso tests take up the majority of the query processing time for FTV methods. Then, why not improve FTV performance by increasing the filtering power and hence reducing *candidate set*? Indeed, this can be accomplished by indexing larger features that usually bear higher discriminative power. To this end, we reconfigured all FTV methods through increasing their feature sizes by just one. This minimal increase in feature size did offer better performance, with the average query processing time going down by approximately 10%; however, the space requirement was almost doubled. Whereas ***GC could lead to significant speedups with a negligible space overhead***; e.g., for AIDS dataset [7], the memory required by GC was just over 1% of the space required for the various FTV indices, but resulting query time speedups up to 40×.

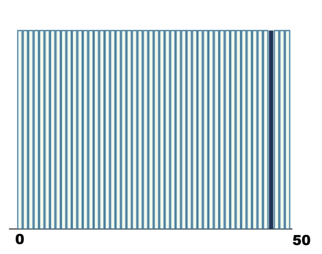
3.2 For General End-Users

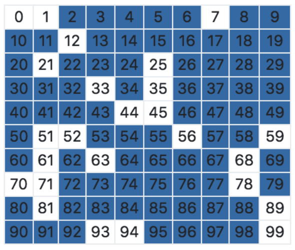
Currently, GC provides two scenarios for general end-users, including *The Query Journey* and *The Workload Run*. And new scenarios could be swiftly plugged in.

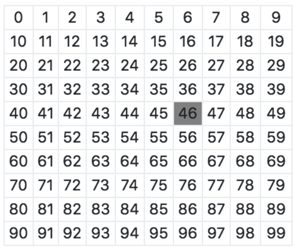
Scenario I: The Query Journey. Through the execution of one query, users are guided to discover the computations inside GC and its principles for acceleration. Figure 2(a) shows the interface where a number of patterns (molecules) are given such that the user could preview (by putting cursor over graph id) and select one (by clicking graph id) to start off. Figure 3 illustrates GC computations, with each subfigure pertaining to a key operation.

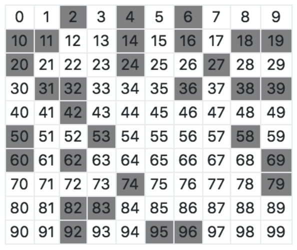
- 3(a) and 3(e): the coming query renders four cache hits (by bars filled with dark blue), among which one is of sub case (H) and three are of super case (H').
- 3(b): Method M evicts C_M with 75 data graphs (by colored boxes having id 2, 3, ..., 98) for sub-iso tests.
- 3(c) and 3(d): cache hits leverage the savings in the number of sub-iso tests; H delivers S, i.e., data graph with id 46 is in the *answer set* for sure (not necessary go for sub-iso verification); similarly, H' brings S' of data graphs that are definitely not in the *answer set* (no need to be verified either).

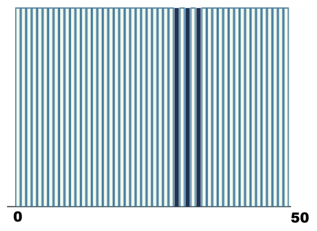
[1]http://74.207.247.121:8081

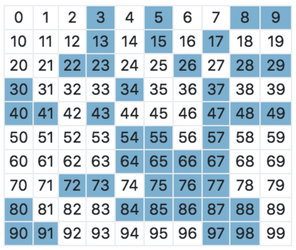

(a) H: Cache Hits (Sub Case)

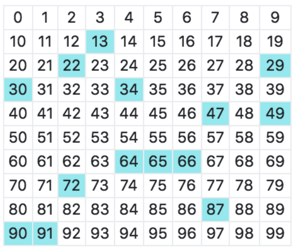

(b) C_M: Candidate Set of M


(c) S: Savings by Sub Case


(d) S': Savings by Super Case


(e) H': Cache Hits (Super Case)


(f) C: Candidate Set of GC


(g) R: Sub-Iso Result over C

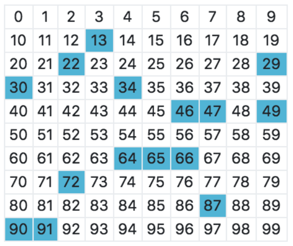

(h) A: Answer Set

Figure 3: GC Interface for Showing Computations during *The Query Journey*

- 3(f): due to S and S', GC reduces the number of sub-iso tests from 75 to 43 (cardinality of C).
- 3(g): 14 graphs in C survive sub-iso tests, resulting R.
- 3(h): the final *answer set* A consists of R and S.

In this example, GC offers speedup 1.74 (75/43) in number of sub-iso testing and in turn expedites query processing.

Scenario II: The Workload Run. It is designed to show users the various cache replacement over a number of policies bundled with GC. User interface is shown as Figure 2(b) and audience experiences are mainly as follows.

- For the workload to run, users could either choose one (from a number of given options after preview) or create a new workload (in which queries are uniformly selected from a pattern pool).
- Upon each executed query, users can view sub/super case cache hit (in percentage calculated as the number of cache-hits over the number of cached graphs).
- After the workload run, users could observe the graph cache replacement due to the limited space in memory. Figure 2(c) shows an example of cache replacement using different policies, where bars filled in dark blue represent graphs with least utilities to be evicted out (meanings are different from those in Figure 3): each graph cache is full of 50 previously executed queries, 10 of which are replaced by the newly coming queries in the workload; obviously, different graphs are cached out in different caches (e.g., PIN cache evicted graphs with id of 39, 41, ..., 49), echoing that various graph replacement policies bear various trade-offs.

3.3 For Developers

GC offers ease of programming for developers, allowing for extension/optimization. For example, alternative graph cache replacement strategies could be swiftly incorporated through extending the Cache class, as defined in Figure 2(d). The developer has to override three abstract methods:

- *updateCacheItems* method deals with the replacement in graph cache, i.e., those with least utilities are evicted such that newly executed queries are accommodated;
- *updateCacheStaInfo* method updates graph utilities, upon the contribution in accelerating other queries;
- *getReplacedContent* method returns the positions of top x cached graphs to be replaced.

4. CONCLUSIONS

We have demonstrated GC, a graph caching system for subgraph/supergraph queries. GC has shown its applicability for audiences with varying programming skills, its "plug-and-play" mechanism for incorporating alternative components, its exclusive graph cache replacement policies with competitive performance and its significant speedup in query time with meagre space overheads when comparing against state-of-the-art approaches.

Acknowledgments. This work is supported by NSFC U1636210, 973 Program 2014CB340300, and NSFC 61421003.

5. REFERENCES

[1] V. Bonnici et al. Enhancing graph database indexing by suffix tree structure. In *PRIB*, pages 195–203, 2010.
[2] R. V. Bruggen. *Learning Neo4j*. O'Reilly Media, 2013.
[3] L. P. Cordella et al. A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.
[4] W. Fan et al. Answering graph pattern queries using views. In *ICDE*, 2014.
[5] W. Fan et al. Parallelizing sequential graph computations. In *SIGMOD*, 2017.
[6] G. Malewicz et al. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.
[7] National Cancer Institute. `http://www.nci.nih.gov/`, March 10 2013.
[8] N. Papailiou et al. Graph-aware, workload-adaptive SPARQL query caching. In *SIGMOD*, 2015.
[9] Z. Sun et al. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.
[10] J. Wang et al. Indexing query graphs to speedup graph query processing. In *EDBT*, 2016.
[11] J. Wang et al. GraphCache: a caching system for graph queries. In *EDBT*, 2017.