Ensuring Consistency in Graph Cache for Graph-Pattern Queries

Jing Wang
School of Computing Science
University of Glasgow, UK
j.wang.3@research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Graph queries are costly, as they entail the NP-Complete subgraph isomorphism problem. Graph caching had been recently suggested, showing the potential to significantly accelerate subgraph/supergraph queries. Subsequently, GraphCache, the first full-fledged graph caching system was put forth. However, when the underlying dataset changes concurrently with the query workload proceeding, how to ensure the graph cache consistency becomes an issue. The current work provides a systematic solution to address this problem, by presenting an upgraded GraphCache system coined GraphCache+ (abbreviated as GC+). We develop two GC+ exclusive models that employ different approaches to deal with the consistency issue. Moreover, we present the logic of GC+ in expediting queries, bundled with the formally proved correctness. We evaluate the performance of GC+ by a real-world graph dataset and a number of query workloads with different characteristics, highlighting the considerable speedup in term of quantified benefit and overhead.

CCS Concepts

•Information systems → Database query processing;

Keywords

Graph queries; Caching system; Cache consistency

1. INTRODUCTION

Graph structured data is increasingly prevalent in modern big data applications. Central to high performance graph analytics is to locate patterns in dataset graphs. Informally, given a query graph g, the system is called to return the set of dataset graphs that contain g (subgraph query) or are contained in g (resp. supergraph query), named the *answer set* of g. These operations can be time consuming for the NP-Complete problem of subgraph isomorphism [4]. Hence, the community has contributed a number of innovative solutions in recent years. One research thread follows the "filter-then-verify" (FTV) paradigm: first, dataset graphs are decomposed to substructures and indexed; then, substructures of coming query graph g are looked up in the dataset index, producing the set of dataset graphs that contain all substructures of g (resp. dataset graphs whose substructures are all contained in g), coined the *candidate set* of g; finally, graphs in the candidate set are verified for subgraph isomorphism (abbreviated as *sub-iso* or *SI*), returning the answer set of g. Similarly, [23] provides a solution for subgraph queries against historical (i.e., snapshotted) graphs – a variation of typical graph queries where snapshots can be viewed as different graphs. However, extensive evaluations of FTV methods [7, 8] show significant performance limitations. Another research thread is concerned with heuristic SI algorithms, which can operate either against a single very large graph or a dataset containing a large number of graphs. These algorithms are capable of pruning away (parts of) dataset graphs that cannot contain the query, expediting sub-iso testing without specialized graph indexes. [14, 9] provide insightful evaluations for such SI algorithms.

Unfortunately, both FTV and SI methods suffer from executing unnecessary sub-iso tests. For example, if a previously issued query is submitted again to the system, it still has to undergo sub-iso tests, as all laboriously derived knowledge (i.e., answer set of previous queries) has been thrown away. Moreover, in many applications, it is natural that submitted graph queries share subgraph or supergraph relations with future queries – in protein datasets, there is a hierarchy of queries for aminoacids, proteins, protein mixtures, unicell bacteria, all the way to multi-cell organisms; in social network exploratory, queries could start off broad (e.g., all people in a geographic location) and become gradually narrower (e.g., by homing in on specific demographics). Based on these observations, we proposed in [25] a fresh graph query processing method, where queries (and their answers) are indexed to expedite future query processing with FTV methods. Subsequently, we presented GraphCache [26], the first full-fledged caching system for general subgraph/supergraph query processing, applicable for all current FTV and SI methods. Following established research, GraphCache handles graph queries against a static dataset throughout the continuous query streaming.

In real-world applications, however, the graph dataset naturally evolves/changes over time. For example, in social networking, newly added groups (graph modeled relations/interactions among people), break-up of existed groups, and the changed relations/interactions among group members are frequently happening. Also, biochemical datasets keep

©2017, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2017 Joint Conference (March 21, 2017, Venice, Italy) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

refreshing by newly-translated, disregarded or transformed proteins, for application/research reason. 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.

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, none of the proposed FTV algorithms so far has updatable index or similar solutions to tackle dataset changes. As a result, the way turns to SI methods, where each dataset graph is painstakingly verified. On the other hand, caching is proved efficient in accelerating graph queries [26]. To this end, it naturally follows an approach using graph cache to alleviate the costly SI methods in processing subgraph/supergraph queries with dataset changes – a topic that has not been discussed yet. Therefore, underpinned by our previous work in [25, 26], we contribute in this paper:

- A systematic solution to expedite subgraph/supergraph queries with dataset changing throughout the query streaming, by presenting an upgraded graph caching system GC+ featured by newly plugged-in subsystems to deal with the evoked cache consistency issue.
- Two cache models named EVI and CON that are proposed exclusively for GC+, representing different designs of ensuring graph cache consistency.
- The logic of GC+ in pruning candidate set for general subgraph/supergraph query processing, together with the formally proved correctness.
- Performance evaluations using well-established SI methods, a real-world dataset and a number of workloads, emphasizing the significant speedup of CON cache.

2. RELATED WORK

Subgraph isomorphism problem has two versions: (i) the decision problem answers Y/N to whether the query is contained in each graph in the dataset; (ii) the matching problem locates all occurrences of the query graph within a large graph (or a dataset of graphs). To address the decision and matching problems, the brute-force approach is to execute sub-iso test of query against each dataset graph (SI method). SI algorithms deteriorate when the dataset contains a large number of graphs, which prompted the prevalence of FTV methods. GC+ could benefit both SI and FTV solutions for general subgraph/supergraph queries.

The community has also looked into subgraph queries against a single massive graph via scale-out architectures [12] or large memory clusters with massive parallelism [24]. For the time being, GC+ does not target such use cases, which are left for future work.

Using cache to expedite queries is prevalent in relational databases, whereas little work had been done for graph-structured query processing. XML datasets have used views to expedite path/tree queries [15]; [13] first proposed the MCR (maximally contained rewriting) approach for tree-pattern queries, but introduced false negatives. GC+ does not produce false negatives or false positives (see §6). Also, GC+ deals with much more complex graph queries, which entail the NP-Complete subgraph isomorphism problem.

Caching has also been leveraged to expedite SPARQL query processing in RDF graphs. [18] proposed the first cache for SPARQL queries. [22] contributed a cache for SPARQL queries with a novel canonical labelling scheme to identify cache hits and a popular dynamic programming planner. Similar to GC+, query processing optimization in [22] 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 [10, 24]: 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. Furthermore, GC+ discovers subgraph, supergraph, and exact-match relationships between the coming query and cached queries, which the canonical labelling scheme in [22] fails to achieve. SPARQL query processing also targets at optimizing join execution plans [5] (based on join selectivity estimator statistics and related cost functions), and the cache in [22] is focusing on this goal, whereas GC+ aims to avoid/reduce costs of executing SI heuristics whose execution time can be highly unpredictable and much higher. Therefore, the overall rationale of GC+ and its way to utilize cache contents differ from that in [22] and in related caching solutions for SPARQL queries.

Pertaining to cache consistency, [6] first explicitly specified the consistency constraint in a query-centric approach and presented how it could be expressed succinctly in SQL. Also, there are studies of cache consistency regarding XML datasets [2] and SPARQL query processing [16]. However, the topic of ensuring graph cache consistency for general subgraph/supergraph queries has not been discussed yet.

3. DEFINITIONS

GC+ is implemented for undirected labelled graphs, following established studies in literature [1, 11]. We assume that only vertices have labels; all our results straightforwardly generalize to directed graphs and/or graphs with edge labels.

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 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 $\forall u \in V_i, l_i(u) = l_j(\phi(u))$.

As is common in literature, we focus on non-induced subgraph isomorphism. Informally, there is a subgraph isomorphism $G_i \subseteq G_j$ if G_j contains a subgraph that is isomorphic to G_i. In this case, we say that G_i is a subgraph of (or contained in) G_j, or inversely that G_j is a supergraph of (contains) G_i (denoted by $G_j \supseteq G_i$). The subgraph querying 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 $g \subseteq G_i$. The supergraph querying 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 $g \supseteq G_i$.

4. SYSTEM ARCHITECTURE

GC+ is a scalable semantic cache for subgraph/supergraph queries, consisting of four subsystems Data Manager, Cache Manager, Query Processing Runtime and Method M, as shown by Figure 1. The first three are GC+ internal and

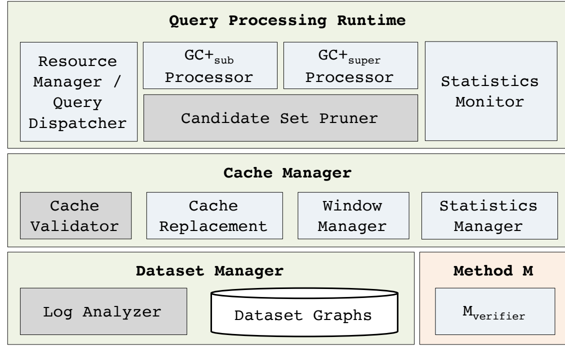


Figure 1: GC+ System Architecture

Method M is the external SI method that GC+ is called to expedite. Method M subsystem includes an SI implementation, denoted $M_{verifier}$, sub-iso testing candidate set M_{CS} (the whole dataset when GC+ is not used).

Dataset Manager subsystem is GC+ specific, containing a component Log Analyzer to handle dataset logs. Cache Manager is responsible for the management of data and metadata stored in the cache. Besides the cache replacement mechanisms, a Window Manager for cache admission control and a Statistics Manager for metadata that work as usual as in GC, Cache Manager of GC+ boasts an additional component Cache Validator, which cooperates with Log Analyzer to ensure the cache consistency. Both Log Analyzer and Cache Validator launch cache model dependent mechanisms to cope with cache consistency (see §5).

Query Processing Runtime subsystem takes charge of query execution and metrics monitoring. Like in GC, it comprises a resource/thread manager, the GC+ 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. Please note that GC+'s logic of Candidate Set Pruner is different and more complicated than that in GC, though the former could be viewed as adapting from the latter. The logic of GC+ shall be presented and proved in §6.

When a query g arrives, Dataset Manager subsystem first identifies whether the dataset has been changed recently. If so, Cache Validator is triggered to reflect these changes to previous queries residing in cache and window (where queries are batched to enter cache; in this work, cached graphs/queries by default cover those previous queries in both cache and window). Then, g is sent to Query Processing Runtime subsystem for query execution. Specifically, GC+ calls processors (GC+$_{sub}$ and GC+$_{super}$) to discover whether g shares subgraph/supergraph relations with cached graphs – each hit graph in cache then contribute its valid part of answer set to prune g's *candidate set*. Finally, the issued query g, together with its answer set and statistics pertaining to the contribution of cached graphs, is fed to the Cache Manager subsystem, where admission control and cache replacement are performed, concurrently with the Query Processing Runtime subsystem executing subsequent queries.

5. CACHE CONSISTENCY

5.1 EVI Cache Model

To address the said challenge arising from dynamic graph dataset, a straightforward solution is to abandon the vague

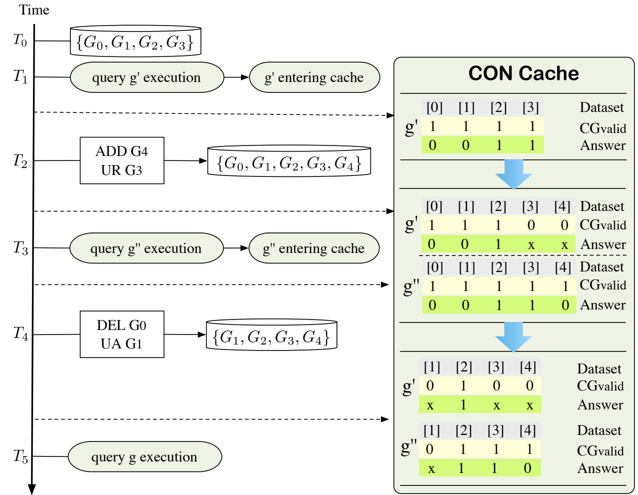


Figure 2: A CON Cache Running Example with Timeline

cache, i.e., evicting (*EVI*) graph cache whenever dataset changes. Therefore, Log Analyzer has to do nothing but raising a flag indicating the dataset is changed, and Cache Validator then clears cached contents indiscriminately. In this way, the caching system in [26] 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 each dataset change.

The problem of EVI cache lies in that it fails to differentiate the validity of cached results. For example, when a dataset graph undergoes some change, its relations with all previous queries in cache (reflected by the cached query results) are affected – some still hold, while others fail. Of course, invalid contents cannot be leveraged to accelerate future queries and should be abandoned. However, the purge in EVI throws away those valid contents as well, making the cache efficiency truncated.

5.2 CON Cache Model

To solve the aforementioned problem of EVI, we develop another cache model named *CON*, which beavers away to identify valid contents in cache. We shall use an example to illustrate how the CON model works for subgraph queries. Mechanism for supergraph queries is similar and is omitted for space reason.

Figure 2 depicts an example, where GC+ 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 and was executed. Assuming that g' passed the admission control and entered the cache later. CON cache thus recorded that g' holds validity towards its relation with all graphs in current dataset (i.e., GC_{valid} covers dataset graphs with id 0, 1, 2, 3), among which $g' \nsubseteq G_0$, $g' \nsubseteq G_1$, $g' \subseteq G_2$ and $g' \subseteq G_3$. 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 regarding its relation 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 containing five graphs. Again, the dataset

Algorithm 1 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 $\mathbb{R}$ from L
6: **for all** $r \in \mathbb{R}$ **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

changed at time T_4 – graph G_0 was deleted and G_1 was updated by edge additions. Regarding 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 on 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.

5.2.1 Analyzing Dataset Log

GC+ 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 develop CON specific mechanisms.

First, Dataset Manager subsystem employs the component Log Analyzer to categorize dataset changes, acting as a preprocessing step for validating CON cache. Algorithm 1 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-exclusive-counter C_A and UR-exclusive-counter C_R.

5.2.2 Validating CON Cache

Then, 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 targets at the biggest possible benefit for query processing. The Cache Validator hence strives to exploit useful previous query result for CON. In GC+, once a query is executed, its answer set is finalized, which snapshots the query's relation against 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's relation towards a dataset graph.

Algorithm 2 Refreshing a cached graph's validity indicator

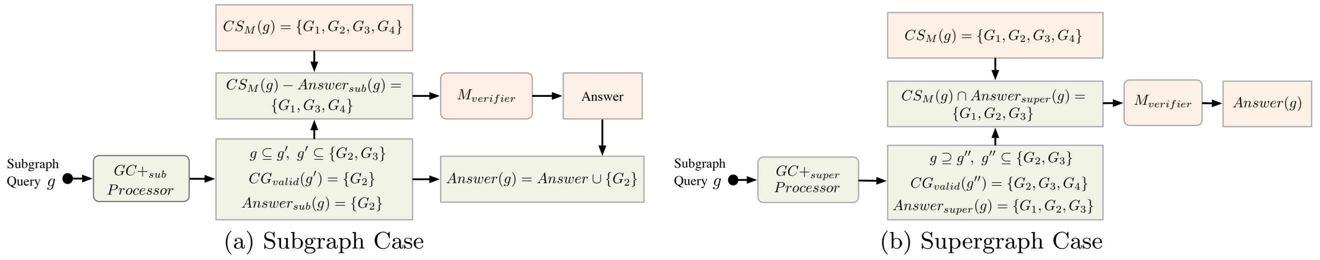
1: Input: Counter container C (containing C_T, C_A and C_R), currently maximum graph id m in dataset, stored info of a cached graph (with its dataset-graph-validity-indicator CG_{valid} and query result $Answer$, both structured by BitSet)
2: Function: Updating CG_{valid}
3:
4: **if** $(m+1) > CG_{valid}.size$ **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**

Algorithm 2 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 relation between g and 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) keeps valid. Whereas other operations fade g's validity on G_i if applicable (line 17). By harnessing the validity 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.

6. QUERY PROCESSING

This section shall illustrate the specific logic of Candidate Set Pruner in GC+. For space reason, we only present for subgraph queries (supergraph queries follow the exact inverse logic). As shown in Figure 2, when a query g arrives,



(a) Subgraph Case (b) Supergraph Case

Figure 3: GC+ Processing of a Subgraph Query g

GC+ discovers whether g is a subgraph or supergraph of cached queries concurrently by processors GC+$_{sub}$/GC+$_{super}$, referred as subgraph/supergraph case.

6.1 Subgraph Case

For example, in Figure 3(a), the new query g comes with candidate set $CS_M(g)$(the current dataset) containing four graphs $\{G_1, G_2, G_3, G_4\}$. 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 well as its up-to-time validity indicator $CG_{valid} = \{G_2\}$, is retrieved. (As mentioned in Algorithm 2, both $Answer(g')$ and $CG_{valid}(g')$ are BitSet structures; here, we employ a set containing dataset-graph-id to help illustrate.)

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

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:

$$Answer_{sub}(g) = \bigcup_{g'_i \in Result_{sub}(g)} CG_{valid}(g'_i) \ \cap Answer(g'_i) \tag{1}$$

where 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:

$$CS_{GC+_{sub}}(g) = CS_M(g) \ \setminus Answer_{sub}(g) \tag{2}$$

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:

$$Answer(g) = Answer_{GC+_{sub}}(g) \cup Answer_{sub}(g) \tag{3}$$

Lemma 1. *The final answer of GC+ in the subgraph case does not contain false positives.*

Proof. Assume false positives are possible and consider the first ever false positive produced by GC+; 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 (3). Furthermore, formula (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 2. *The final answer of GC+ in the subgraph case does not introduce false negatives.*

Proof. Assume false negatives are possible and consider the first ever false negative produced by GC+ 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 GC+ 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 (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 (3), G_{FN} will be added to $Answer_{sub}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). ☐

Theorem 3. *The final answer of GC+ in the subgraph case is correct.*

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

6.2 Supergraph Case

Figure 3(b) depicts an example of supergraph case in GC+, 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 $CG_{valid}(g'')$ ($\{G_2,\ G_3,\ G_4\}$) are retrieved. Again, the new query g comes with candidate set $CS_M(g)$ ($\{G_1, G_2, G_3, G_4\}$).

Compared with subgraph case, reasoning the logic of GC+ in supergraph case is more interesting. (i) Consider graph $G_1 \in CS_M(g)$: G_1 does not hold validity regarding its relation with g'', hence no previous query result about G_1 could be utilized. g has to rest on the sub-iso test to identify whether G_1 is in the final answer set. (ii) For graph $G_2 \in CS_M(g)$: $g'' \subseteq G_2$ holds, which does not make G_2 sub-iso test free though providing $g'' \subseteq g$, as the relation between g and G_2 is still obscure and has to be verified by sub-iso test. Similarly, G_3 is not free of sub-iso test either. (iii) For graph $G_4 \in CS_M(g)$, G_4 holds validity for its relation with g'' as $g'' \nsubseteq G_4$. Since $g'' \subseteq g$, if $g \subseteq G_4$ were to be true, then 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 $CS_M(g)$. 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.,

$$g''.Answer_{super}(g) = (\overline{CG_{valid}(g'')} \cup Answer(g'') \quad (4)$$

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:

$$CS_{GC+_{super}}(g) = CS_M(g) \cap \bigcap_{g''_j \in Result_{super}(g)} g''_j.Answer_{super}(g) \quad (5)$$

where $\text{Result}_{super}(g)$ 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 4. *The final answer of GC+ 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 5. *The final answer of GC+ in the supergraph case does not introduce false negatives.*

PROOF. Assume false negatives are possible and consider the first ever false negative produced by GC+; 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). This implies that $\exists g''$ such that $G_{FN} \notin g''.Answer_{super}(g)$ and $g'' \subseteq g$. By formula (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., $G_{FN} \not\subseteq g''$. But $g \subseteq G_{FN}$ and $g'' \subseteq g \Rightarrow g'' \subseteq G_{FN}$ (a contradiction). □

THEOREM 6. *The final answer of GC+ in the supergraph case is correct.*

PROOF. There are only two possibilities for error; GC+ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 4 and 5. □

6.3 Putting It All Together and Optimal Cases

The Query Processing Runtime subsystem first applies equation (2) on CS_M and then applies (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 GC+ 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, GC+ 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 GC+ that $g'' \subseteq g$, $Answer(g'') = \emptyset$ and g'' holds validity on all graphs currently in dataset. Thus, GC+ 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.

7. PERFORMANCE EVALUATION

7.1 Experimental Setup

GC+ is implemented in Java, by extending the graph caching system GC [26]. 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. Following GC, the default value in GC+ for the upper limit on the sizes of Cache and the Window stores were 100 and 20 respectively.

Method M We used three well-established SI methods, including GraphQL (GQL) provided by [14], a modified VF2[3] (denoted VF2+) provided by [11], for being well-established and good performers [14, 8]; we also used vanilla VF2[3] as it is extensively used in FTV methods.

Dataset We employ a real-world dataset AIDS [19] (the Antiviral Screen Dataset of the National Cancer Institute) that is prevalently used in literature [14, 7, 1, 11]. AIDS contains 40,000 graphs, each with on average ≈45 vertices (std.dev.: 22, max: 245) and ≈47 edges (std.dev.: 23, max: 250), whereby the few largest graphs have an order of magnitude more vertices and edges.

Dataset Change Plan Dataset change operations are performed in batches, with occurrence time indicated by the id of queries in workload (recall Figure 2). The plan we used for AIDS 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; then, 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.

We follow the established principle for the generation of our workloads, using two different algorithms to synthesize queries from the initial dataset graphs. Each workload consists of 10,000 queries with typical sizes in literature [11, 27] – 4, 8, 12, 16 and 20 edges.

Type A Workloads Queries of these workloads are generated in the following manner: first, a source graph is randomly selected from dataset graphs; then, a node is selected randomly in the said graph; finally, a query size is selected uniformly at randomly from given sizes 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, we have used two different distributions; namely, Uniform (U) and Zipf (Z), with the probability density function of the latter given by $p(x) = x^{-\alpha}/\zeta(\alpha)$, where ζ is the Riemann Zeta

function[21]. Ultimately, we had 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, we first create two query pools: a 10,000-query pool with queries with non-empty answer sets against the initial dataset, and a second 3,000-query pool with no match in any untreated 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: we continuously relabel 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, we generate workloads 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. We thus have three categories of Type B workloads: "0%", "20%" and "50%", denoting the above probability used.

We use Zipf $\alpha = 1.4$ by default; as a reference point, web page popularities follow a Zipf distribution with $\alpha = 2.4$ [21]. We only allow one for one Window (i.e., 20 queries) before starting measuring GC+'s performance. We report on both the benefits and the overheads of GC+. Reported metrics include query time and number of sub-iso tests per query, 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). As a yardstick, [17] (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.

Cache Replacement Policy GC+ incorporates all the replacement policies developed in GC. Here, we use the coined hybrid (HD) policy for experiments, as its performance is always better or on par with the best alternative [26]. Specifically, HD coalesces another two GC/GC+ exclusive policies PIN and PINC. PIN scores each graph in cache by considering the total number of subgraph isomorphism tests alleviated by the said graph (coined R). PINC extends the mentioned ranking by taking into account the possibly vast differences in query execution time (denoted C), where cost is estimated by a heuristic [25]. As C is estimated, using it in PINC does not always lead to improvements in query time. And through a large number of experiments, we have observed that when the values of the R exhibit a high variability, they are discriminative enough on their own, where considering C can actually lead to lower time gains (i.e., PIN performs better than PINC). However, when the values of R exhibit a low variability, adding in the C component leads to considerable query time improvements. When the HD policy is invoked, it first retrieves the R from Statistics Manager and computes its variability [20] by using the (squared) coefficient of variation (CoV). CoV is defined as the ratio of the (square of the) standard deviation over the (square of the) mean of the distribution. When $CoV > 1$, the associated distribution is deemed of high variability, as exponential distributions have $CoV = 1$ and typically hyper-exponential distributions (which capture many high-variance, heavy tailed distributions) have $CoV > 1$. In this case, HD performs cache eviction using PIN's scoring scheme; otherwise, it turns to PINC's scoring scheme.

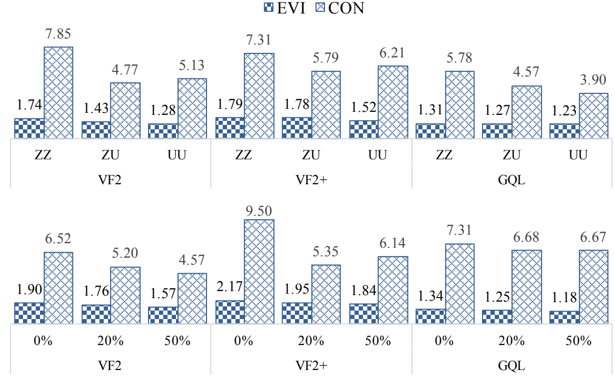


Figure 4: GC+ Speedup in Query Time

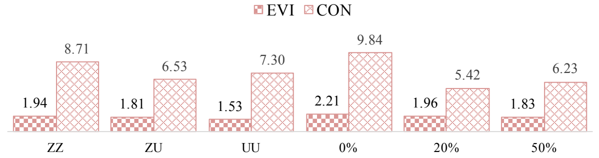


Figure 5: GC+ Speedup in Number of Sub-iso Tests

7.2 Results and Insights

Figure 4 depicts 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. Note the interesting finding that speedups for UU workload are close to those of ZU (e.g., 4.77 vs 5.13 against VF2 base method), 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. And it does: we measured 2.5X the number of exact-match cache hits in ZU vs UU. However, in GC+, exact-match cache hit does not necessarily introduce sub-iso test free (see §6.3) – those resulting zero sub-iso test merely account 4% among exact-match cache hits of the said ZU workload (vs 11% in UU). Moreover, recall that GC+ exploits also subgraph/supergraph hits. Indeed, we measured circa 2X such matches 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 vs each saved subgraph/supergraph match. But the key insight here is that ***by utilizing exact-matches and subgraph/supergraph matches, GC+ can benefit both skewed and non-skewed workloads***.

Figure 5 shows the speedups in the number of sub-iso tests performed. Please note that ***under a given 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), whatever SI method being the Method M, GC+ results exactly the same pruned candidate set for each query***. Therefore, Figure 5 is independent of the three methods considered in this work. Juxtaposing Figures 4 and 5

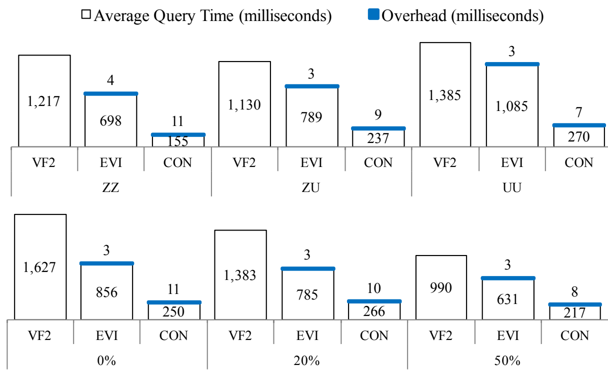


Figure 6: Average Execution Time and Overhead per Query

leads to the interesting insight: ***Reductions in the number of sub-iso tests do not translate directly into reductions in query time***, echoing the claim that graph cache hits render different benefits [26]. In all cases, though, ***GC+ achieves significant improvements in both query processing time and number of sub-iso tests performed***.

Figure 6 depicts a break-down of query processing time for method M, EVI and CON. 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 the time of analyzing dataset log and validating cache (see Algorithm 1 and 2) – 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 1 and 2 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 Figure 4 aside Figure 5, it is obvious that ***CON sweeps EVI in query processing speedup with a negligible additional overhead***.

8. CONCLUSIONS

We presented a systematic solution to handle graph cache consistency, by providing an upgraded system GC+, which is capable of expediting general subgraph/supergraph queries with dataset changes. We developed two GC+ exclusive cache models EVI and CON with different mechanisms on ensuring cache consistency. We illustrated the specific logic of GC+ in reducing candidate set for query execution and formally proved its correctness. Our performance evaluation has demonstrated the considerable speedup achieved by CON. Future works include further optimizing CON cache with retrospective validating mechanisms, developing a distributed/decentralized version of GC+ and extending GC+ to benefit subgraph queries when finding all occurrences of a query graph against a single massive graph.

9. REFERENCES

[1] V. Bonnici et al. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010.

[2] S. Bottcher. Cache consistency in mobile XML databases. In *Proc. WAIM*, pages 300–312, 2006.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[4] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[5] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *Proc. EDBT*, 2014.

[6] H. Guo et al. Relaxed currency and consistency: How to say "good enough" in SQL. In *Proc. SIGMOD*, 2004.

[7] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[8] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12):1566–1577, 2015.

[9] F. Katsarou, N. Ntarmos, and P. Triantafillou. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *Proc. EDBT*, 2017.

[10] J. Kim, H. Shin, and W.-S. Han. Taming subgraph isomorphism for RDF query processing. *PVLDB*, 8(11):1238–1249, 2015.

[11] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. ICDE*, 2011.

[12] L. Lai et al. Scalable subgraph enumeration in MapReduce. *PVLDB*, 8(10):974–985, 2015.

[13] L. V. S. Lakshmanan et al. Answering tree pattern queries using views. In *Proc. VLDB*, 2006.

[14] J. Lee et al. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[15] K. Lillis and E. Pitoura. Cooperative XPath caching. In *Proc. SIGMOD*, 2008.

[16] J. Lorey et al. Caching and prefetching strategies for SPARQL queries. In *Proc. USEWOD*, 2013.

[17] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. VLDB*, 2005.

[18] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query caching. In *Proc. ESWC*, 2010.

[19] NCI - DTP AIDS antiviral screen dataset. http://dtp.nci.nih.gov/docs/aids/aids_data.html.

[20] R. Nelson. *Probability, Stochastic Processes, and Queueing Theory*. Springer Verlag, 1995.

[21] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[22] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware , workload-adaptive SPARQL query caching. In *Proc. SIGMOD*, 2015.

[23] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. ICDE*, 2016.

[24] Z. Sun et al. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[25] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *Proc. EDBT*, 2016.

[26] J. Wang, N. Ntarmos, and P. Triantafillou. GraphCache: a caching system for graph queries. In *Proc. EDBT*, 2017.

[27] X. Yan et al. Graph indexing: a frequent structure-based approach. In *Proc. SIGMOD*, 2004.