GraphCache: A Caching System for Graph 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 query processing is essential for graph analytics, but can be very time-consuming as it entails the NP-Complete problem of subgraph isomorphism. Traditionally, caching plays a key role in expediting query processing. We thus put forth GraphCache (GC), the first full-fledged caching system for general subgraph/supergraph queries. We contribute the overall system architecture and implementation of GC. We study a number of novel graph cache replacement policies and show that different policies win over different graph datasets and/or queries; we therefore contribute a novel hybrid graph replacement policy that is always the best or near-best performer. Moreover, we discover the related problem of cache pollution and propose a novel cache admission control mechanism to avoid cache pollution. Furthermore, we show that GC can be used as a front end, complementing any graph query processing method as a pluggable component. Currently, GC comes bundled with 3 top-performing filter-then-verify (FTV) subgraph query methods and 3 well-established direct subgraph-isomorphism (SI) algorithms – representing different categories of graph query processing research. Finally, we contribute a comprehensive performance evaluation of GC. We employ more than 6 million queries, generated using different workload generators, and executed against both real-world and synthetic graph datasets of different characteristics, quantifying the benefits and overheads, emphasizing the non-trivial lessons learned.

CCS Concepts

•**Information systems** → **Database query processing;**

Keywords

Graph analytics; Graph queries, Caching system

1. INTRODUCTION

Graph datasets are proliferating nowadays, due to their ability to capture and allow for the analysis of complex relations among objects, by modelling entities with nodes and their relations/interactions with edges. Graphs have thus been used, with great success, in a wide variety of application areas, from chemical and bioinformatics datasets to social networks. 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 dataset graphs that contain g (subgraph query) or are contained in g (supergraph query), aptly named the *answer set* of g. Unfortunately, these operations can be very costly, as they entail the NP-Complete problem of subgraph isomorphism[5] and even the popular algorithms [4, 18, 31] are known to be computationally expensive. To this end, the research community has contributed a number of innovative solutions over the last few years. A large number of these follow 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 (abbreviated as *sub-iso* or *SI* in the rest of this work). However, recently extensive evaluations of FTV methods [9, 12] show significant performance limitations.

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 we can make is that in many real-world applications, graph queries submitted in the past bear subgraph or supergraph relations with future queries. These relationships arise naturally. Queries against a biochemical dataset range from queries for simple molecules and aminoacids, all the way to queries for proteins of multicell organisms. In exploratory smart-city analytics, queries referring to road networks may pertain to neighbourhoods, 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.

Based on these observations, we proposed in [34] a new graph query processing method, in which queries (and their answers) are indexed and used to expedite future query processing with FTV methods. Underpinned by our method in [34], this work presents a novel full-fledged caching system, where any general subgraph/supergraph query method in

the literature could be plugged in, and overall contributes:

- GraphCache (GC), a full-fledged caching system for sub-/supergraph queries, with detailed discussions of design issues, its architecture and implementation, dealing with resource management (memory and threads) and dynamic management of the cache index;
- A fresh perspective to expedite state-of-the-art solutions for the general subgraph isomorphism problems (SI methods) by GC (in addition to FTV methods);
- A semantic graph cache which harness sub/supergraph cache hits, extending the traditional exact-match-only hit and leading to significant speedup for GC;
- A number of graph cache replacement strategies with different trade-offs, including a novel hybrid graph cache replacement policy with performance always better or on par with the best alternative;
- A novel cache admission control mechanism enhancing the performance gains of GC;
- Comprehensive evaluations (with millions of queries) utilising well-established FTV and SI methods, against real-world and synthetic datasets with different characteristics and different workload generators, quantifying benefits/overheads and uncovering key insights.

To the best of our knowledge, GC is the first caching system in the literature for general subgraph/supergraph queries.

2. RELATED WORK

Subgraph/supergraph queries entail the subgraph isomorphism problem, which has two versions. 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. However, sub-iso tests are costly, being NP-Complete[5]. Several heuristic algorithms have been proposed over the years. [9] provides an insightful presentation and comparison of several such (SI) algorithms (which could be integrated within GC).

SI algorithms deteriorate when the dataset is comprised of a large number of graphs, as each graph has to be tested. Thus appeared the "filter-then-verify" (FTV) paradigm. FTV methods try to reduce the set of graphs against which to run the sub-iso 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. Briefly, dataset graphs are decomposed into features (i.e., paths, trees, cycles or arbitrary subgraphs), which are then recorded in an indexing structure (e.g., trie [2, 6], hash-based bitmap [14], etc.). Query processing then proceeds in two stages. First, in the *filtering stage*, the query graph g is decomposed to its features, which are then used to retrieve from the dataset index the IDs of those graphs containing all of them; the result is a subset of the dataset graphs, named the *candidate set* of g. Then, in the *verification stage*, g undergoes sub-iso testing against each graph in the *candidate set*.

Similarly, [29] presents 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; the main focus of this work is on reconstructing minimal snapshots around candidate matching nodes, by using a set of indices allowing for the retrieval of nodes with specific labels/neighbourhoods at given time points. [9] presents an insightful performance evaluation and [12] provides a systematic performance and scalability study of subgraph FTV methods. Though we are not aware of similar in-depth studies on supergraph FTV solutions, [36] provides a concise overview for studies published prior to 2013 and recently [20] proposes an efficient solution for supergraph queries. GC is capable of expediting query processing for both subgraph and supergraph FTV methods.

The community has also looked into subgraph queries against a single, very large graph (consisting of possibly billions of nodes). [16] and [30] employ scale-out architectures and large memory clusters with massive parallelism respectively. [8] and [28] provide a centralised solution to the same problem, via advanced pruning approaches addressing the matching order issues faced by most other SI algorithms. GC does not target such use cases for the time being and extending our system to queries against a single massive graph or distributed operation is left for future work.

Caching of query results has long been a mainstay in data management systems, from filesystem block caching to web proxy caching and the cache of query result sets in relational databases. In the realm of graph-structured queries, however, little work has been done. For XML datasets, views have been used to accelerate path/tree queries [1, 19, 21]; Besides, [17] firstly proposed the MCR (maximally contained rewriting) approach for tree pattern queries and [33] revisited it by providing alternatives; both exhibit false negatives for the query answer. Our GC does not produce any false negative or false positive (formal proof of correctness in [34]). Also, GC 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. [22] introduced the first SPARQL cache, where a relational DB was employed to store the metadata. [27] contributed a cache for SPARQL queries based on a novel canonical labelling scheme (to identify cache hits) and on a popular dynamic programming planner [23]. Similar to GC, query optimization in [27] 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 [13, 30]; 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, GC discovers subgraph, supergraph, and exact-match relationships between a new query and the queries in the cache, something that the canonical labelling scheme in [27] fails to achieve. SPARQL query processing also aims at optimizing join execution plans [7] (based on join selectivity estimator statistics and related cost functions), and the cache in [27] is focusing on this goal, whereas GC 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 GC and the way cache contents are exploited differs from that in [27] and in related SPARQL result caching solutions.

Finally, [15] 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. [15] 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, [15] does not deal with central issues of a cache system (cache replacement, admission control, overall architecture/design, etc.).

3. DESIGN ISSUES AND GOALS

GraphCache is implemented for undirected labelled graphs, as is typical in the literature (e.g., [2, 10, 14]). For simplicity, we assume that only vertices have labels; all our results straightforwardly generalize to directed graphs and/or graphs with edge labels.

Formally, a labelled graph $G = (V, E, l)$ consists of a set of vertices V and edges $E = \{(u, v), u, v \in V\}$, and a function $l : V \rightarrow U$, where U is the domain of labels. 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, and we say that G_i is a subgraph of (contained in) G_j, or that G_j is a supergraph of (contains) G_i denoted by $G_j \supseteq G_i$. As is common in the relevant literature, we focus on non-induced subgraph isomorphism. Last, the subgraph (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 $g \subseteq G_i$ ($g \supseteq G_i$, respectively).

In designing GraphCache, we identified a set of design issues and goals, pertaining to the characteristics of (i) the query workloads, (ii) the underlying graph datasets, and (iii) the algorithmic and system context within which GC will operate (e.g., categories of research methods GC will complement). Overall, GC 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 [2, 6, 9, 14, 35] 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 GC this in effect refers to the popularity of query graphs or of regions of the dataset graphs. GC 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: workloads 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, we utilized well over 6 million queries for our performance evaluation.

Graph Datasets. Fortunately there exist a number of real-world graph datasets commonly used in related research.

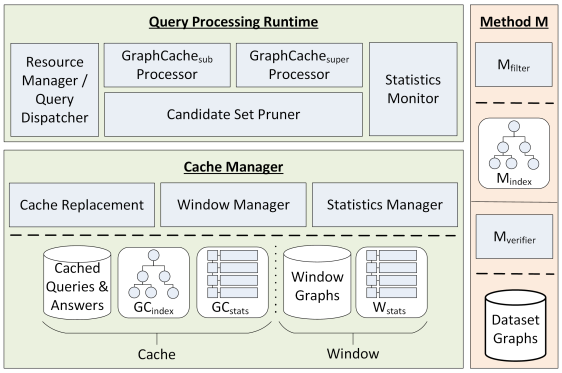


Figure 1: GraphCache System Architecture

These help concretize the effects of any solution on real-world data and allow direct comparison of methods and result repeatability. For this reason we will report evaluations conducted over three popular graph datasets: AIDS[24], PDBS[11], and PCM[32]. However, it is worth creating additional synthetic datasets so to perform evaluations under characteristics unseen in the real-world datasets. Specifically, we created a synthetic dataset presenting interesting characteristics regarding the number, size and node degrees of graphs in the dataset. Interestingly, with respect to dataset with graphs having a high average node degree, we found that GC needs special mechanisms without which its performance benefits degrade.

Algorithmic Context. GraphCache is intended to be a general-purpose front-end for graph query processing. GC entails a query indexing strategy that, as explained in [34], 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. 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 1 shows its main architectural components, comprising three major subsystems: Method M, Query Processing Runtime, and Cache Manager. The last two are internal subsystems of GC; the first is the method that GC is called to expedite and hence external to GC.

The Method M subsystem includes, at a minimum, the base graph dataset and a sub-iso test implementation, denoted $\mathrm{M}_{verifier}$. 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 1 for simplicity). When GC is not used, sub-/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 $\mathrm{M}_{verifier}$ 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 dataset.

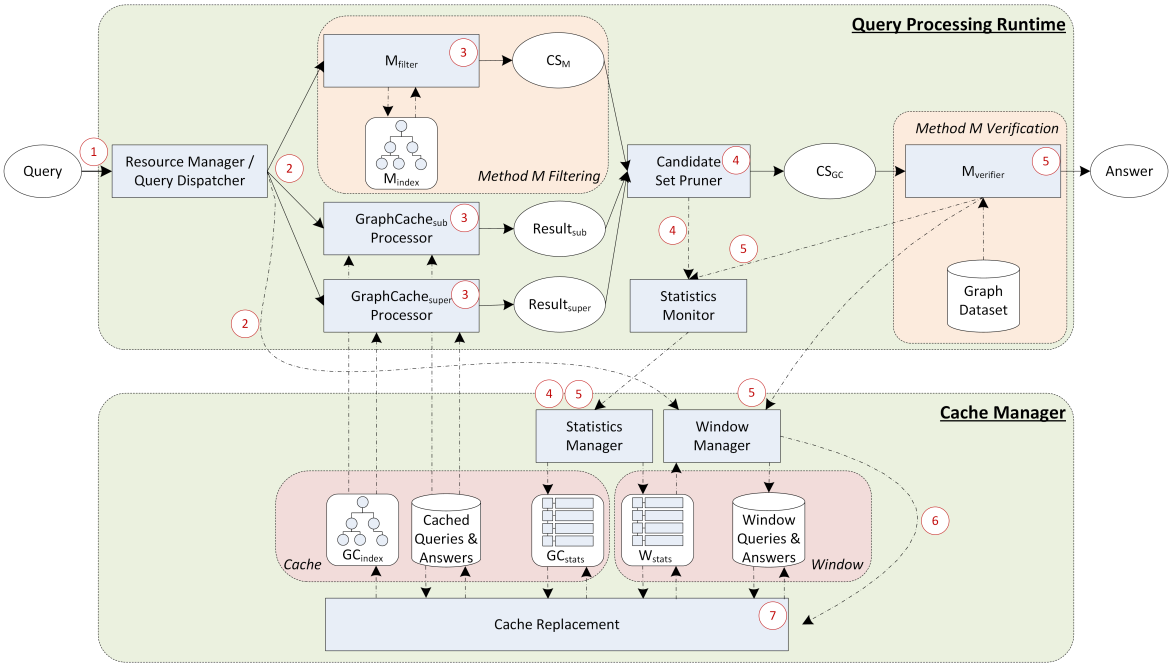


Figure 2: GraphCache System Data and Control Flow

Within GC, the Query Processing Runtime is responsible for the execution of queries and the monitoring of key operational metrics. It comprises: a resource/thread manager dispatching queries to the various filtering/verification modules, the internal subgraph/supergraph query processors, the logic for GC's candidate set pruning, and a statistics monitor. These components communicate with Method M and the Cache Manager via well-defined APIs.

In turn, the Cache Manager deals with the management of data and metadata stored in the cache. It comprises the cache replacement mechanisms, a Window Manager responsible for cache admission control and maintenance of the cache contents, a Statistics Manager responsible for metadata pertaining to past or current queries, as well as the stores for all GC-related data including cached queries and their answer sets, currently executing (not cached) queries, and metadata/statistics for both past and current queries.

Figure 2 depicts the flow of control and data in GC during processing of a query. 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 (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.

5. QUERY PROCESSING

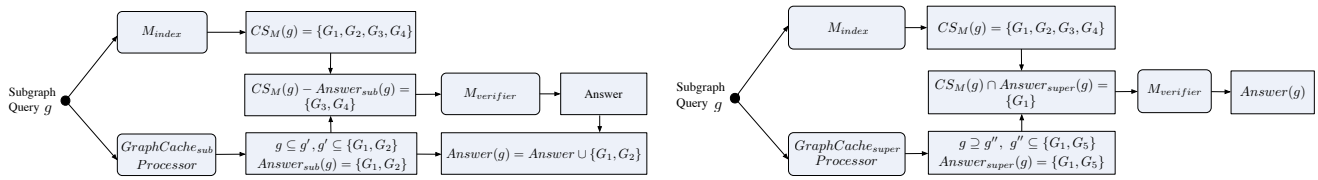
This section discusses the design and implementation of GraphCache's Query Processing subsystem, responsible for the execution of queries and the monitoring of key operational metrics. For the sake of clarity we first describe GC's operation when caching subgraph queries; we shall then discuss how GC can be used for supergraph queries as well.

5.1 Candidate Set Pruning

This subsection overviews the essence of [34]. For more details and formal proofs of correctness, please refer to [34]. 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.

$GraphCache_{sub}$ Processor. The $GraphCache_{sub}$ Processor is responsible for identifying when a new query g is a subgraph of a previous query g'. When g' was executed , GC indexed g''s features in GC_{index} and stored its result set and relevant statistics in the cache data stores.

Figure 3(a) depicts an example flowchart for this 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, determining 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, $g \subseteq G_2$). Therefore, we



(a) Subgraph Case (b) Supergraph Case

Figure 3: GraphCache Processing of a Subgraph Query g

can safely remove G_1 and G_2 from $CS_M(g)$ and add them directly to the final answer set. 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:

$$CS_{GC_{sub}}(g) = CS_M(g) \setminus \bigcup_{g'_i \in Result_{sub}(g)} Answer(g'_i) \quad (1)$$

where $\text{Result}_{sub}(g)$ contains all query graphs currently in GC_{index} of which g is a subgraph.

GraphCache$_{super}$ Processor. In turn, the GC_{super} Processor is responsible for identifying when a new query g is a supergraph of a previous query g''. Figure 3(b) depicts an example flowchart for this case. 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 $\{G_1, G_5\}$. 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 $CS_M(g)$. 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:

$$CS_{GC_{super}}(g) = CS_M(g) \cap \bigcap_{g''_j \in Result_{super}(g)} Answer(g''_j) \quad (2)$$

where $\text{Result}_{super}(g)$ contains all query graphs currently contained in GC_{index} of which g is a supergraph.

Putting It All Together. The Candidate Set Pruner collects CS_M and the results of the above two Processors; it then first applies equation (1) on CS_M, then applies (2) on the result of the previous operation. The end result is a reduced candidate set, which is then sub-iso tested by $\text{M}_{verifier}$.

Two Special Cases. Additionally, there are two cases that warrant attention since they yield the greatest possible gains.

First, note that GC can easily recognize the case where a new query, g, is isomorphic to a previous cached query. For connected query graphs, this holds when $\exists g' \in GC_{index}$ such that $g \subseteq g'$ or $g \supseteq g'$, and g and g' have the same number of nodes and edges. Thus, GC can return the cached result of g' directly and completely avoid any further processing.

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

Supergraph Query Processing. As mentioned earlier, GC can expedite both subgraph *and* supergraph query processing. In the latter case, the filtering components of GC 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 $Result_{super}(g)$ are removed from $CS_M(g)$ and added to $Answer_{GC_{super}}(g)$, and the graphs not appearing in the intersection of the answer sets of graphs in $Result_{sub}(g)$ are completely subtracted from $CS_M(g)$. Also, the first special case still holds, but for the second special case processing terminates when $\exists g' \in Result_{sub}(g)$ such that $Answer(g') = \emptyset$.

5.2 Statistics Monitoring

The final component of this 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 §6.1) and to communicate them to the Statistics Manager component of the Cache Manager subsystem. Currently the following quantities are monitored:

- 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.
- Break-down of total filtering times of the query to the three filtering components.
- Number of times the query was matched by either GC Processors and number of special-case matches.
- Most recent time a cached query was hit, expressed as the serial no. of last benefited query.
- Total reduction in the candidate set size of new queries. This statistic is easily monitored, as the Candidate Set Pruner knows exactly which graphs from the answer set of each matched cached query where removed from the candidate set of any given new query (through application of equations (1) and (2)).
- Total time saving due to the cached query. This statistic is computed as the sum of the estimated costs of all sub-iso tests alleviated, as mentioned above. The estimation of the individual sub-iso test time $c(g, G)$ for a query graph g against a dataset graph G, is performed using the formula[34]: $c(g, G) = \frac{N \times N!}{L^{n+1} \times (N-n)!}$, where L is the number of distinct labels, n the number of nodes in g, and $N \geq n$ the number of nodes in G.

6. CACHE MANAGEMENT

GC's Cache Manager subsystem, running in parallel with the Query Processing Runtime subsystem, deals with the management of the data and metadata stored in the cache.

We first discuss the various data stores handled by this subsystem, then dive into the design of its various components.

6.1 Data Layer

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

The Cache stores include three components: First, a component storing copies of cached queries (i.e., the actual graph submitted as a query to GC) alongside their result sets (i.e., 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[2], 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 GC. 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 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 §5.1); last (most recent) time a query contributes, expressed as the serial number of the benfited query; total contribution of the cached query in reducing the candidate sets and processing times of future queries, expressed as the number of dataset graphs removed from the candidate set of queries due to their being in the cached query's answer set and the cumulative sub-iso test time alleviated; etc.

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).

6.2 Window Manager with Admission Control

The Window Manager, implemented as a separate thread, is the brain of the Cache Manager subsystem. It keeps 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. It also executes the Cache Replacement algorithms when the Window is full, and rebuilds GC_{index} to reflect any changes in the cached queries store. In the latter case, the Window Manager first computes the new contents of the cache (by replacing evicted queries with admitted Window queries) and invokes the indexing mechanism; queries arriving at the system while this procedure is taking place, continue being served by the old index and update the old statistics. Once the re-indexing is over, the new cache contents and index are swapped in place of the old ones, and any statistics entries corresponding to evicted queries are removed lazily from the statistics 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. Nevertheless, our design 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, actually implemented as simple in-memory reference (pointer) swaps), trading off some possible cache hits against window queries.

Cache Admission Control. While experimenting with different workloads and datasets we observed that often the performance of GraphCache would be lower than expected; that is, although GraphCache benefited the majority of queries, the overall speedup achieved was very low (close to 1). The reason behind this proved to be that the cache was *polluted*, storing and improving the performance primarily of inexpensive graph queries. To alleviate this situation, we make the natural conjecture that past expensive (time-wise) queries are more likely to benefit later coming expensive queries as they will help in alleviating more expensive sub-iso tests (and vice-versa for inexpensive queries). We therefore propose a novel admission control mechanism, part of the Window Manager component, which optimises the graph cache by preventing inexpensive queries from being added to the cache. To quantify the expensiveness of a

Table 1: Running Example: Cached Query Statistics

SerialNo / Query ID	Last Hit	Number of Hits	CS_M Reduction	SI Cost Reduction
11	91	23	170	2600
13	51	32	80	1200
37	69	26	76	780
53	78	13	210	360
82	90	5	120	150
91	95	4	10	270

query graph, we use the ratio of its verification time over its filtering time. 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 (a threshold value of 0 disables this component). To compute said threshold, our mechanism examines the queries in the first few *windows* and computes an expensiveness value which would result in a predefined percentage of queries being classified as expensive. We have also experimented with more dynamic approaches (e.g., greedily adapting the threshold using an exponential back-off approach until the achieved time speedup reaches a local maximum); without loss of generality and due to lack of space, we omit further discussion of these techniques. The reasoning behind the above lies in the fact that, given a graph query processing framework, the filtering time is relatively constant across queries, in contrast to the dramatic variance of verification times. Moreover, the verification stage is known to dominate the query time[9, 12], and the larger the verification time the more overwhelming this dominance. Thus, the above mechanism is a simple yet effective technique to guarantee that more complex queries are prioritised.

6.3 Cache Replacement Policies

[34] used a specific graph replacement policy (PINC). We have developed and tested a number of new different cache replacement strategies (POP, PIN and HD), each offering different trade-offs and performance characteristics for different datasets and query workloads. We describe the various strategies here and report on their relative performance in §7. In all cases, the replacement strategies access query statistics through the Statistics Manager's key-value store interface, and return the IDs of queries to be cached out. In order to compute this set, queries are assigned a "utility" value and those with the lowest such values are cached out.

Below we present all cache replacement algorithms considered in this work. We use Table 1, presenting a snapshot of GC_{stats} for a number of hypothetical cached queries, as our running example. In all cases, assume that the replacement algorithm is invoked at time point 100 (i.e., right after the query with serial number 100 was executed) 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. The utility of each cached graph is its last hit time, i.e., the serial no. of the last query that is expedited by said cached graph. In our 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 of temporal locality of reference and thus fails to identify cases of queries which have contributed huge savings to query processing although not having been used in a while. In our example, we can see that query 13 has contributed 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 here: POP (short for Popularity-based Ranking). This policy assigns each cached graph a utility value equal to H/A, where H is the number of times a query has contributed and A is its age in the cache, computed as the difference between the last serial number assigned to any query and the cached query's own serial number; this function then manages to combine query popularity and age. In our example, this policy would evict queries 11 and 53.

POP + Number of Sub-Iso Tests (PIN). As mentioned, unlike traditional exact-match caches in which a cache hit saves a disk/network IO, cache hits in GraphCache may result into vastly different reductions in query processing times. One of the reasons why this is so, is 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) Instead of looking just at the number of hits H of a cached query, PIN assigns each cached graph a utility value equal to R/A, where R is the total number of subgraph isomorphism tests alleviated by said cached query, and A is the same aging factor as above. The utility formula of PIN can also be rewritten as: $\frac{R}{A} = \frac{H}{A} \cdot \frac{R}{H}$, which can be interpreted as the probability of the query being a hit (i.e., its popularity), times the average savings in number of subgraph isomorphism test per hit. In our running example, this policy would evict queries 13 and 91.

PIN + Sub-Iso Tests Costs (PINC). PIN takes into account the number of sub-iso tests alleviated. Another GraphCache-exclusive replacement policy PINC further considers the possibly vast differences in query execution times. PINC assigns each cached query a utility value equal to C/A, where A is the same aging factor as above, and C is the total decrease in query processing time due to the cached query. Alas, this figure cannot be computed unless the query graph is sub-iso tested against all graphs in its candidate set, which is a moot point in our case; instead, as mentioned (§5.2), we use a heuristic to estimate this cost. PINC may improve upon PIN's utility value computation by considering the actual (estimated) time cost of alleviated sub-iso tests instead of deeming them all equivalent. PINC's utility formula can be rewritten as: $\frac{C}{A} = \frac{H}{A} \cdot \frac{R}{H} \cdot \frac{C}{R}$, 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. In our running example, PINC would evict queries 53 and 82.

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 PINC). However, when the values of R exhibit a low variability, adding in the 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* policy (HD), coalesces both PIN and PINC. More specifically, when the HD policy is invoked, it first retrieves the R component from GC_{stats} and computes its variability[25] 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 uses PINC's scoring scheme.

In our running example, the mean R value is $\mu = 111$ and its standard deviation $\sigma \approx 72$; then $CoV = \sigma/\mu \approx 0.65 < 1$ and thus HD will use PINC and evict queries 53 and 82.

7. PERFORMANCE EVALUATION

7.1 System Setup

We have implemented all aforementioned components and subsystems of GraphCache 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.

We used GraphCache on top of three subgraph FTV and three SI methods (due to space limitations, we only present results for subgraph queries). 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; we also experimented with other values for both C (200, 300) and W (50, 100, 200) to test their impact on GC's performance. Last, the sizes of the various thread pools are all set to 1 so as to show just the benefits of using a graph query cache. For the FTV methods we chose GraphGrepSX [2] (GGSX), Grapes [6], and CT-Index [14], specifically because they are proven to be top performers in their class[12]. 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, we examine two alternatives, Grapes1 and Grapes6, with 1 and 6 threads respectively. To be fair, we altered the code of Grapes so to stop query processing after the first match in each dataset graph. Please note that all mentioned values match their default configurations in [2, 6, 14]. For the SI methods we used GraphQL[10] as provided by [18] and a modified version of VF2[4] (denoted VF2+) provided by [14], again for being well-established and good performers[9, 12]; we also used vanilla VF2[4] since it has been used by several FTV implementations [2, 6, 9]. GC uses 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 GC. This diversity in the implementation languages of the incorporated methods attests to GC's flexibility.

7.2 Datasets and Query Workloads

We employ three real-world (AIDS, PDBS, PCM) and one synthetic graph datasets with different characteristics. More specifically, AIDS[24] – 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[11] 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[32] 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. Last, the Synthetic dataset was created using [3] and contained 1,000 graphs with on average ≈892 nodes (std.dev.: 417, max: 7,135) and ≈7,991 edges (std.dev.: 5.09, max: 8,007) per graph. We created this dataset 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. 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).

We follow the established principle for the generation of our workloads, using two different algorithms to synthesize queries from the dataset graphs, outlined below.

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 several pre-defined 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[26]. 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 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: 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; we also use $\alpha = 1.1$ representing a smaller skewness and $\alpha = 1.7$ for a higher skewness. As a reference point, web page popularities follow a Zipf distribution with $\alpha = 2.4$ [26]. 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 [6, 14, 35]. 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. 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). The results were produced over more than 6 million queries! As a yardstick, [21] (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.

7.3 Results and Takeaways

Figure 4 depicts the speedups attained by GraphCache when CT-Index and GGSX where used as Method M (results for other FTV and SI methods showed similar trends and are thus omitted for space reasons). We can see that GraphCache attains significant speedups (up to 10× lower query processing times in this case), and that it is always one of the GC-exclusive policies (PIN, PINC) 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, 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 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. For the remainder of this section we will be using HD as the replacement policy; results for other caching policies show similar trends and are thus omitted for space reasons.

Figure 5 depicts speedups in query processing time against all FTV methods for queries on the PDBS dataset (results for other datasets are similar). Query processing time speedups range from 1.60× (i.e., 37.5% lower processing time) to more than 42×. A similar picture is drawn in Figure 6 for speedups in the number of sub-iso tests performed. Juxtaposing Figure 5 and 6 leads to the following interesting insight: ***Reductions in the number of sub-iso 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 sub-iso tests performed***.

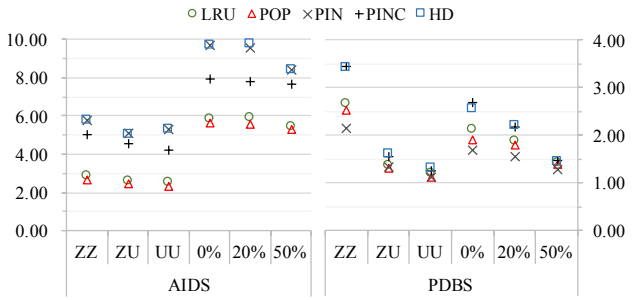
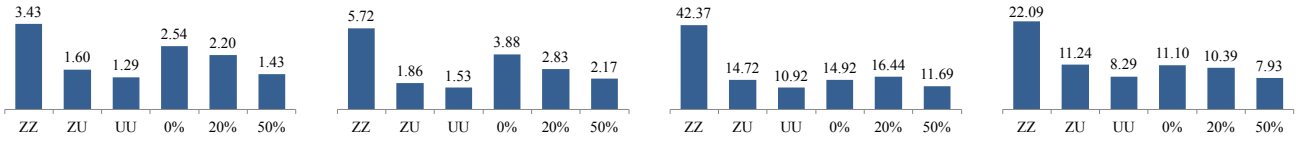


Figure 4: Query Processing Time Speedups over CT-Index Across Replacement Policies

Figure 7 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 omitted for space reasons). 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 5 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.

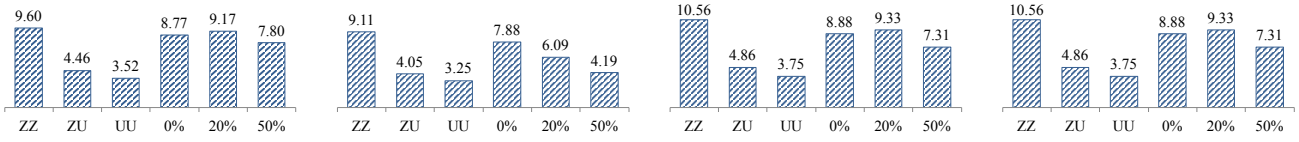
Figure 8 shows the performance of GC against GGSX for queries on AIDS and PDBS, for varying cache sizes (results for other methods and datasets show similar trends). We can see that ***increasing the cache size improves the performance of the cache***. However, this does not mean that one can increase the size of the cache indefinitely; the size of the cache is first limited by the amount of main memory available for GC, then by the overhead associated with updating the cache contents (more on this shortly).

Figure 9 shows the speedups in query time (9(a)) and number of sub-iso tests (9(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 9(b). For better understanding of this trend, let us concentrate on



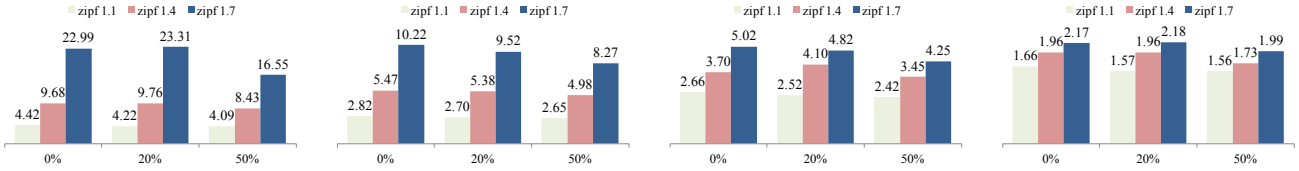
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 5: GraphCache Speedup in Query Time for PDBS Across All Methods M



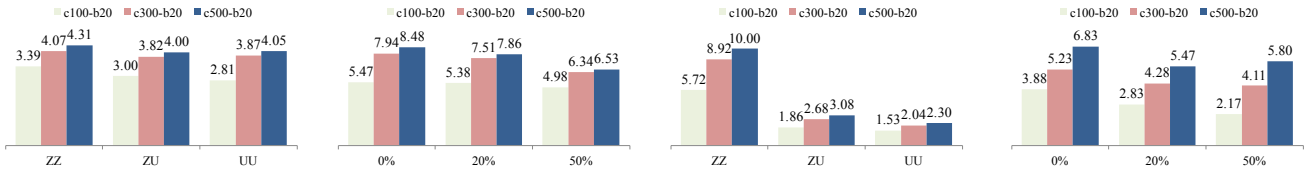
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 6: GraphCache Speedup in Number of Sub-iso Tests for PDBS across all Methods M



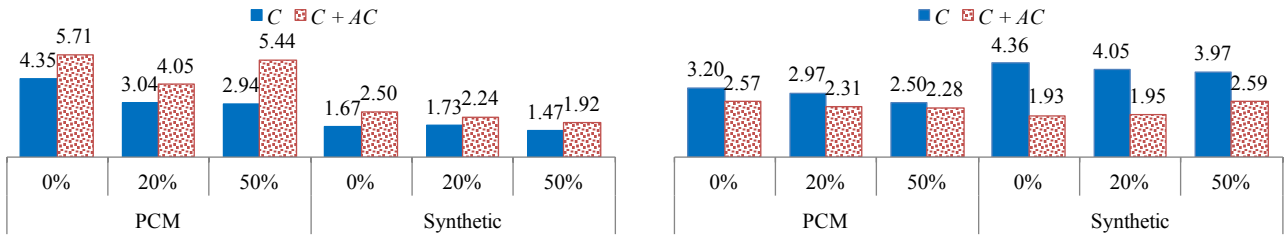
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 7: GraphCache Speedup in Query Time for Type B workloads on the AIDS dataset, for various value of Zipf α



(a) AIDS/Type A Workloads (b) AIDS/Type B Workloads (c) PDBS/Type A Workloads (d) PDBS/Type B Workloads

Figure 8: GraphCache Speedup in Query Time against GGSX with various cache sizes



(a) Query Time Speedups (b) Speedups in Number of Sub-Iso Tests

Figure 9: GraphCache Performance vs Grapes6 for Type B Workloads on PCM/Synthetic Datasets

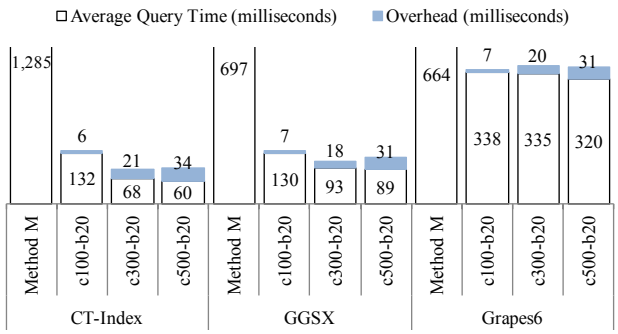


Figure 10: Average Execution Time and Overhead (milliseconds) per Query for the 20% workload on AIDS

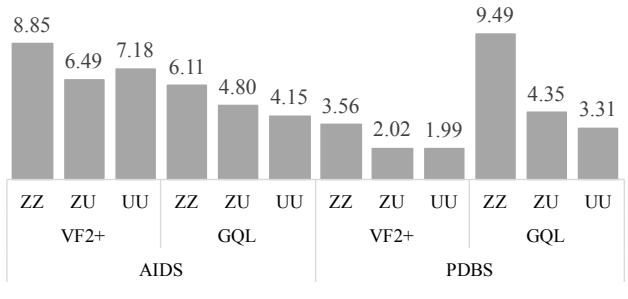


Figure 11: GC Query Time Speedups vs SI Methods

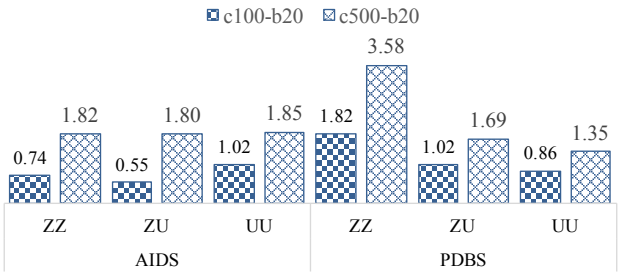


Figure 12: Query Time Speedups of GC/VF2+ vs CTIndex

the Synthetic-50% workload: GC without admission control yields a speedup as high as ≈4× in the number of sub-iso tests, but the resulting query time speedup is only ≈1.5×. 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 ≈16.5 seconds with Grapes6, going down to ≈15 seconds for GraphCache without admission control – a 1.1× speedup; the remaining 99% "inexpensive" queries enjoy speedups of 2×, going from ≈0.200 seconds down to ≈0.100 seconds, but they account for a much smaller percentage of the overall query processing time compared to the top-1% ones. When we enable the admission control mechanism, these top-1% expensive queries are prioritized, with their average query processing time going down considerably to ≈10 seconds – a much improved 1.65× speedup. Hence, ***despite the lower speedup in number of sub-iso tests, the overall query processing time benefits greatly***.

We have shown so far that GraphCache leads to significant decreases in the query processing time and number of sub-iso tests of FTV (and SI) methods. We know that sub-iso 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 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). 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, GC accomplishes its speedup for a ***negligible space overhead***; for example, for the AIDS dataset the memory and disk space required by GraphCache was just over 1% of the space required for the indexes of the various FTV methods, but leading to time speedups of up to 40× (figure omitted for space reasons).

Figure 10 depicts a break-down of query processing time for FTV methods and GraphCache, showing how much of GC 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 8), 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 we can see that what we lose in maintenance overhead, we gain in query time. That means that, if we had designed our architecture to update the cache contents in-line (i.e., not in parallel) with query processing, we would see diminishing returns with larger cache sizes. Our current design does not suffer from this problem; however, we expect that, for considerably larger cache sizes, this overhead may outgrow the time required for the Window to fill (and thus for a new replacement/re-indexing round to begin). 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***.

Figure 11 depicts the query processing speedups of GC over the two well-established SI methods considered in this work – GQL and VF2+ (vanilla VF2 results where similar and are omitted for space and readability reasons). We can see that ***GC improves the performance of well-established SI methods***, with the same meagre 100-query cache configuration as above. ***This is significant in that GC 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.*** Note the interesting finding 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 §7.2). And it does: we measured circa 2.5X the number of exact-match cache hits in ZU vs UU. However, recall that GC exploits also sub/supergraph hits. When exact-matches are not frequent, GC loads graphs in the cache that can help with their sub/supergraph relationships. 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 sub/supergraph match. But the key insight here is that ***by utilizing exact-matches and sub/supergraph matches, GC can introduce significant benefits in both skewed and non-skewed workloads***.

Let us now take a step back and look at how FTV methods and GC 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 GC operating on top of a simple SI method? Figure 12 shows the results when comparing GC on top of VF2+ against CT-Index (also using VF2+ for its verification chores), across several datasets and Type-A workloads (results for Type-B workloads omitted for space reasons). For the small 100-query cache, GC performs on par or better than CT-Index in six out of nine 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 GC's space requirements are under ≈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 GC matches or outperforms CT-Index across the board (by a factor of 1.8× on average). Note that even for this "larger" cache, GC's space requirements are less than

≈70% of CT-Index's index size for PDBS and less than 1% for AIDS (and comparable to the latter against GGSX and Grapes). The conclusion is then that ***GC 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.

8. CONCLUSIONS

We presented GraphCache, to the best of our knowledge the first full-fledged caching system for general subgraph/supergraph query processing, including its architecture meeting demanding design goals, a number of GC-exclusive graph-query-aware cache replacement policies, and an accompanying cache admission control mechanism. The proposed system can be used to expedite all current FTV and SI methods (bridging these two, alas, separate threads of research so far), and is applicable for both subgraph and supergraph queries. Our extensive performance evaluation has proven the applicability and appropriateness of our approach. GC achieves considerable improvements in query processing time for meagre space overheads. Our work also revealed a number of key lessons, pertaining to graph caching and query processing. Future work currently focuses on two big ticket items: first, to develop a distributed/decentralized version of GraphCache; second, to extend GraphCache to benefit subgraph queries when finding all occurrences of a query graph against a single massive stored graph.

9. REFERENCES

[1] A. Balmin et al. A framework for using materialized XPath views in XML query processing. In *Proc. VLDB*, 2004.

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

[3] J. Cheng, Y. Ke, and W. Ng. GraphGen. http://www.cse.ust.hk/graphgen/.

[4] 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.

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

[6] R. Giugno et al. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS One*, 8(10):e76911, 2013.

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

[8] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{ISO}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. SIGMOD*, 2013.

[9] 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.

[10] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proc. SIGMOD*, 2008.

[11] Y. He et al. Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. *PNAS*, 99:10325–10329, 2002.

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

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

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

[15] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *Proc. GRADES*, 2013.

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

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

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

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

[20] B. Lyu et al. Scalable supergraph search in large graph databases. In *Proc. ICDE*, 2016.

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

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

[23] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proc. SIGMOD*, 2008.

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

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

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

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

[28] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

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

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

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[32] C. Vehlow et al. CMView: Interactive contact map visualization and analysis. *Bioinformatics*, 27:1573–1577, 2011.

[33] J. Wang, J. Li, and J. X. Yu. Answering tree pattern queries using views: a revisit. In *Proc. EDBT*, 2011.

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

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

[36] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *PVLDB*, 6(10):829–840, 2013.