Effective Caching Techniques for Accelerating Pattern Matching Queries

Arash Fard, Satya Manda, Lakshmish Ramaswamy, and John A. Miller
Computer Science Department
The University of Georgia
Athens, GA, USA
Email: {ar, manda, laks, jam}@cs.uga.edu

***Abstract*—Using caching techniques to improve response time of queries is a proven approach in many contexts. However, it is not well explored for subgraph pattern matching queries, mainly because of subtleties enforced by traditional pattern matching models. Indeed, efficient caching can greatly impact the query answering performance for massive graphs in any query engine whether it is centralized or distributed. This paper investigates the capabilities of the newly introduced pattern matching models in graph simulation family for this purpose. We propose a novel caching technique, and show how the results of a query can be used to answer the new similar queries according to the similarity measure that is introduced. Using large real-world graphs, we experimentally verify the efficiency of the proposed technique in answering subgraph pattern matching queries.**

Keywords*—*data-intensive computing; subgraph pattern matching; caching technique; graph simulation; subgraph isomorphism

I. Introduction

Recently, there has been renewed research interest in developing platforms, algorithms and techniques for scalable processing of graphs. This has largely been driven by many emerging applications such as web data analytics and social network mining which are characterized by massive graphs. One important class of graph queries is the *subgraph pattern matching* [1]. At a very high-level, subgraph pattern matching seeks to find subgraphs of a data graph that are *similar* to a given query graph. Subgraph pattern matching queries are being increasingly employed in analysis of social networks [2], [3].

Early research on subgraph pattern matching focused on the *subgraph isomorphism* model, which searches for exact matches. However, subgraph isomorphism is proven to be NP-complete [4]. More recently, researchers have explored *graph simulation models* as alternatives to subgraph isomorphism [5], [6], [7], [3]. Unlike subgraph isomorphism, these models are tractable (have polynomial time complexities). Furthermore, graph simulation models are known to capture semantically meaningful similarity features better than their traditional counterparts [2], [8], [9], [10], [11]. Thus, graph simulation models are increasingly being adopted for applications such as social network analytics. The state-of-the-art in this family is the *tight simulation* model [5].

Although graph simulation models are tractable, they are still highly computation-intensive for massive graphs. Researchers have adopted techniques such as smarter indexing strategies and parallel algorithms to achieve better scalability. Surprisingly, very few researchers have explored the idea of utilizing results of an earlier query graph to answer subsequent pattern matching queries. In other words, most existing subgraph pattern matching systems treat each incoming query completely independently from previous queries. We contend that reusing query results can yield significant performance benefits. This is because results from a previous query can help considerably narrow down the search space for related subsequent queries. Intuitively, it will be much more efficient to search for the results of a new query within a relatively-small graph resulting from a structurally-similar prior query rather than repeatedly searching in the massive data graph (as long as it can be guaranteed that the result from the prior query contains all matches of the incoming query). Similarity among queries is quite common in real-life scenarios. For example, there are small, highly popular regions in social-network graphs. Studies on Facebook reveal that despite users having links to numerous objects, less than 10% of data were accessed during a 6-day trace [12].

While some preliminary ideas for results reuse has been recently explored in the context of SPARQL queries [13], [14], their applicability for subgraph pattern matching in generic graphs has not been studied. The other related work was published very recently [15]. However, this work adopts *pre-computed views* rather than reusing query results. Designing effective caching strategies for subgraph pattern matching is inherently challenging on many fronts. First and foremost, we need to address the *pattern containment* problem; i.e., we should determine the similarity conditions under which it would be guaranteed that all the results of a query are contained in the results of another query. Second, we should store the results of a query graph in such a way that it utilizes less memory, while preserves pattern containment conditions and improves cache hit rate. Third, we need techniques to quickly determine which previous query results (if any) can be used to answer an incoming query.

This paper presents a unique query-results caching and reuse framework for subgraph pattern matching. To the best of our knowledge, this is one of the first research works to explore caching as a mechanism for enhancing the scalability and performance of subgraph pattern matching. Our framework embodies four major research contributions.

- First, we present a novel *pattern containment scheme* based on *tight simulation* model to quickly and efficiently determine whether an incoming query can be

answered by the results of the previous queries that are already present in the cache. This technique not only is useful for accelerating a tight simulation-based query engine, but also can be easily adapted for speeding up subgraph isomorphism queries.

- Second, we introduce novel variants of dual and tight simulation, which produce significantly more meaningful results in addition to being more amenable for caching in real-world data graphs.
- Third, we present several important properties of graph simulation models that we have discovered while designing our caching technique and proving its correctness. We believe these important properties will have considerable impact on future research in the field of graph pattern matching.
- Fourth, we present a detailed experimental study involving several real-world graphs. Through these experiments, we not only demonstrate the effectiveness of the proposed techniques, but also investigate the effects of various key parameters on the scalability and performance of caching frameworks for subgraph pattern matching.

Our caching framework can be easily integrated with existing distributed graph pattern matching systems [6]. We believe that such an integrated platform will be very attractive for modern applications such as real-time social network querying and analytics.

The remainder of this paper is organized as follows. In the next section, we review several pattern matching concepts related to this research. Motivation of this work and its design overview are explained in section III. In section IV, we introduce new versions of dual and tight simulation. The pattern containment conditions and the caching mechanism are explained in section V. Section VI is dedicated to empirical studies. It is followed by a section on related work. The paper ends with a conclusions section.

II. Preliminaries

In this research, we consider a *data graph* as a massive labeled directed-graph notated by $G(V_G, E_G, l_G)$ where V_G is its set of vertices, $E_G \subseteq V_G \times V_G$ is its set of directed edges, and l_G is a function that maps each vertex to its label. For simplicity, we assume that there are no multiple edges in the graph. For any $u \in V_G$, we call $v \in V_G$ its *child* when $(u, v) \in E_G$; the same way, we call $w \in V_G$ its *parent* when $(w, u) \in E_G$. Each *query graph*, like $Q(V_Q, E_Q, l_Q)$, is a fairly small directed graph without any multiple edges. Without loss of generality, we assume that each query is a connected graph. The goal of subgraph pattern matching is to find all subgraphs of G that match to Q. Processing these queries is usually very time consuming. In the presence of popular queries, reusing the results of the old queries to answer a portion of new queries can boost the performance. The goal of this research is investigating this technique, which can be eventually used in a cache system for answering subgraph pattern matching queries. In this section, we review several subgraph pattern matching models that are closely related to this research.

A. Pattern Matching Models

Subgraph isomorphism is the most famous pattern matching model that retrieves the exact topological matches. Nevertheless, it is NP-complete in the general case [4], and it can potentially produce an exponential number of subgraph results. Therefore, it may not be a feasible model for massive graphs. Graph simulation [16], as an alternative model, relaxes the restrictions enforced by subgraph isomorphism and provides a quadratic algorithm [17] for finding a set of all results. In simple words, a vertex of the data graph, u', is in a graph-simulation relation to a vertex of the query, u, if it has the same label and it has children that are in graph-simulation relation with the children of that vertex. Intuitively, graph simulation model only preserves the child relationships of each vertex. Dual simulation [7] is an extension to graph simulation model. It improves the result of graph simulation by taking into account not only the children of each query vertex, but also its parents. Moreover, the time complexity of its algorithm is cubic. The difference of models is illustrated through an example in figure 1. This example is inspired from Amazon product co-purchasing network [18], where if a product i is frequently co-purchased with product j, the graph contains a directed edge from i to j. Here, each letter inside the vertex is the category of the product and represents its label. Moreover, each number beside a vertex represents its ID number. The subgraph matching results of this example are displayed in table I.

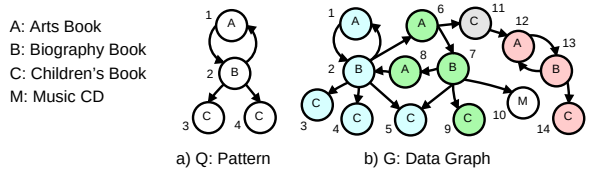


Fig. 1. An example for different subgraph pattern matching models

TABLE I
Results of different pattern matching models

Model	Symbol	Subgraph Results
subgraph isomorphism	$Q \trianglelefteq_{iso} G$	$f(1,2,3,4) \rightarrow (1,2,3,4)$, $(1,2,3,5), (1,2,4,5)$
graph simulation	$Q \trianglelefteq_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8,12\}, \{2,7,13\}$, $\{3,4,5,9,11,14\}, \{3,4,5,9,11,14\})$
dual simulation	$Q \trianglelefteq^{D}_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8,12\}, \{2,7,13\}$, $\{3,4,5,9,14\}, \{3,4,5,9,14\})$
tight simulation (based on $\trianglelefteq^{D}_{sim}$)	$Q \trianglelefteq^{T}_{sim} G$	$R(1,2,3,4) \rightarrow (1,2,\{3,4,5\}$, $\{3,4,5\}), (12,13,14,14)$
CAR-dual simulation	$Q \trianglelefteq^{CD}_{sim} G$	$R(1,2,3,4) \rightarrow (\{1,6,8\}, \{2,7\}$, $\{3,4,5,9\}, \{3,4,5,9\})$
CAR-tight simulation (based on $\trianglelefteq^{CD}_{sim}$)	$Q \trianglelefteq^{CT}_{sim} G$	$R(1,2,3,4) \rightarrow (1,2,\{3,4,5\}$, $\{3,4,5\})$

In figure 1, the subgraph isomorphic match of pattern Q to data graph G has three subgraph results. In comparison, graph simulation and dual simulation always produce a single subgraph result although it might be a disconnected subgraph. An important difference from subgraph isomorphism is that the relationship from the vertices in the query graph to the data graph is not a function; i.e., a vertex in the query graph can be related to several vertices of the subgraph result. As it

is displayed in table I, the result of graph simulation contains all the vertices of G except 10 because they all have the same label and meet the child relationship constraint with respect to their counterpart vertices in Q. Only vertex 11 will be omitted in the result of dual simulation because it does not satisfy the parent relationship constraint.

In this paper, we mainly use tight simulation [5], which is the state-of-the-art model in graph simulation family. This model adds a locality condition to dual simulation to improve the quality of its result, while the time complexity of its algorithm still remains cubic. In simple terms, a few *candidate vertices* on the graph result of dual simulation are selected, and then for each candidate vertex its neighborhood is extracted as a potential subgraph result. Dual simulation is applied again on these subgraphs to find the final results. The *candidate vertices* in G (2, 7, and 13 in figure 1, for example) are those that are dual match to the center of Q (2 in our example). Moreover, the distance for finding the neighbor vertices equals to the radius of Q (one in our example). It should be noticed that the vertices of a graph with minimum eccentricity are the centers of the graph, and the value of their eccentricity is the radius of the graph [19]. Similar to subgraph isomorphism, tight simulation may return several subgraph results. It is also proved that these subgraph results contain all the results of subgraph isomorphism [5]. As it is displayed in table I, the result of tight simulation in our example will be two subgraphs.

III. Motivation and Overview

In this section, we first review the motivation of a caching system for subgraph pattern matching queries, and try to identify its related problems. Then, we give a brief overview of our proposed system.

A. Motivation and challenges

Given a set of query graphs and a data graph, the idea is to store their results of subgraph pattern matching in a cache-like system, and use them to answer new queries. This technique can reduce the average response time of queries provided that there are popular new queries that their answers are contained in the answer of the old queries. Hence, the main challenge in designing an effective content-aware cache system for subgraph pattern matching is the problem of pattern containment.

The main focus of this paper is about pattern containment problem. Let us assume that Q_{old} is an old query graph on G and its set of results is A_{old}. The subproblems are (1) how to store the pair of Q_{old}, A_{old} in the cache space; (2) receiving a new query Q_{new} different from Q_{old}, how we can evaluate the relation between the two to realize if the answer of the new query is contained in A_{old}. The goal is to answer greater number of new queries without referring to G.

Each time a new query is received, it should be compared to all old queries in the cache to find out if there is any match. For a large graph, it is likely that after awhile the number of stored queries in the cache grows to a large number; therefore, the overhead of the search process in the cache can become restricting. We have designed simple methods to filter the queries in the cache space when we search for a match to a new query. The implemented mechanism for searching the cache is shown to be very efficient in our experiments.

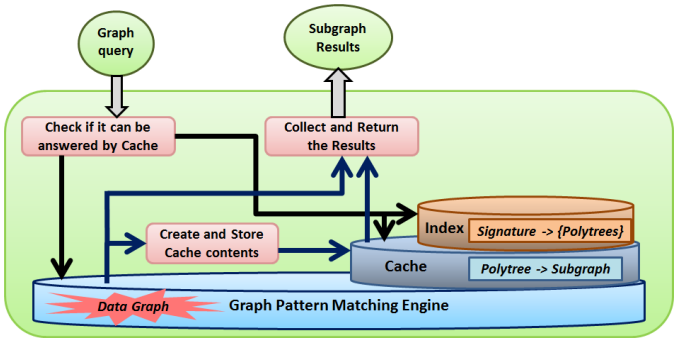


Fig. 2. The overall architecture of the proposed cache system

When the cache space becomes full, an appropriate replacement mechanism should be available to remove some of the old contents from the cache in order to open space for new queries. Cache replacement strategies are extensively studied in other contexts [20]. The most popular cache replacement policies are based on replacing *least frequently used (LFU)*, or *least recently used (LRU)* items. As a future work, it might be worthwhile to design a specific replacement policy for pattern matching queries. At this work, we have only implemented a simple replacement mechanism which works based on LFU policy in order to test our proposed caching mechanism. We have not tested LRU policy, and we do not expect it to significantly affect the results.

B. Architectural overview

The caching system proposed in this paper works based on the tight simulation model; nevertheless, it can also be used to retrieve subgraph isomorphic matches because the result of tight simulation always contains the result of subgraph isomorphism. The overall architecture of the proposed cache system is depicted in figure 2. The block, which is labeled Graph Pattern Matching Engine, is the main query processing engine that maintains the data graph. It can be a centralized or a distributed system. The cache system resides on top of this engine.

At the warm-up phase of the system, or when a query cannot be answered using the contents of the cache, it will be submitted to the main query engine. Then its result will be stored in the cache space for the future usage. Nevertheless, instead of storing the original query, we retain its *spanning polytree*. Using the *polytree* has several advantages. First, it can be used to answer a wider range of new queries because it is more generic. Second, a polytree has a number of properties that make pattern-containment conditions simpler, which will improve the cache hit rate. Moreover, it is possible to compute dual-simulation of a polytree instead of tight-simulation for our purpose. Dual-simulation is always faster than tight-simulation. We explain these properties in detail in section V-B.

The data structure of the cache space is a key-value map from a polytree to a subgraph. To find the appropriate map-value for this polytree, we first find its tight-simulation result in the data graph and then extract the induced subgraph

corresponding to all the vertices in this result. The pair of polytree and its corresponding induced subgraph is stored in the cache space. In order to facilitate the search in the cache space, we have designed a simple indexing system based on the *signature* of the polytrees. The data structure of the index is a key-value map from a signature to a set of polytrees. When a new query matches to a polytree based on pattern containment conditions, all of its expected results can be retrieved from the polytree's correspondent subgraph; therefore, there would be no need to refer to the original data graph. We will explain signature and pattern containment conditions in section V-A.

IV. Cardinality Restricted Simulation

As it was explained in the previous section, we store key-value pairs in the cache space, where a polytree is the key and its corresponding subgraph is the value. Regarding the fact that the memory size of a cache is always limited, it is appealing to shrink the size of the subgraphs stored in this space. Moreover, a smaller subgraph means less computation time when the query result is retrieved from cache. In this section, we propose new versions of dual and tight simulations, in which we shrink the size of their results, and improve their expressiveness as well.

Dual simulation, the way it was first introduced in [7], does not consider cardinality of vertices in the child or parent relationships. Referring back to the example displayed in figure 1, vertex 13 in G is a dual match to vertex 2 in Q because the vertex 14 is a dual match to both vertices 3 and 4 at the same time. We found that in many real-world graphs, like co-purchasing and citation networks, distribution of labels are very skewed; e.g., there are communities that many vertices have the same label. Dual simulation produces less meaningful results in these regions because when several vertices in the query have the same label, they all may match to a single vertex or a cycle of two vertices with the same label in the data graph.

Here, we introduce a modified version, called *cardinality restricted* or *CAR-dual simulation*, in which the number of matched children or parents with the same label in the data graph should not be less than their correspondents in the query. This extra condition does not increase the time complexity of dual simulation, but it improves its expressiveness. Our experiments show a significant drop in the number of vertices in the results of CAR-dual simulation in comparison to dual simulation. All the models defined so far based on dual simulation; e.g., strong [7], strict [6], and tight simulation [5], can be revised to employ CAR-dual simulation. This modification, not only improves the quality of their results, but also can potentially impact their performance because they will be constructed on less number of vertices. In example of figure 1, vertices 12, 13, and 14 are excluded in the result of CAR-dual simulation. Consequently, *CAR-tight simulation*, which is defined based on the new dual simulation, will have a single subgraph result as it is displayed in table I. A smaller result is clearly more amenable for cache storage. We now present the formal definition of these new models.

1) CAR-dual Simulation: We first present formal mathematical definition of dual simulation, and based on that we define CAR-dual simulation. By definition [7], pattern Q matches data graph G via *dual simulation*, denoted by $Q \preceq^{D}_{sim} G$, if there is a binary relation $R_D \subseteq V_Q \times V_G$ such that it meets the following conditions: (1) Having the same label: $(u, u') \in R_D \Rightarrow l_Q(u) = l_G(u')$; (2) All vertices of Q are covered: $\forall u \in V_Q, \exists u' \in V_G : (u, u') \in R_D$; (3) Child relationship: $\forall (u, u') \in R_D[(u, v) \in E_Q \Rightarrow \exists v' \in V_G : (v, v') \in R_D \wedge (u', v') \in E_G]$ (4) Parent relationship: $\forall (u, u') \in R_D[(w, u) \in E_Q \Rightarrow \exists w' \in V_G : (w, w') \in R_D \wedge (w', u') \in E_G]$.

The result of this model is defined as a *maximum dual match set*, $R_D \subseteq V_Q \times V_G$, which is the largest relation set between Q and G with respect to $Q \preceq^{D}_{sim} G$. The *result dual match graph*, $G_D(V_D, E_D, l_D)$, for this model is a subgraph of G that can represent R_D. By definition, G_D is a subgraph of G that satisfies these conditions: (1) $(u, u') \in R_D \Leftrightarrow u' \in V_D$; (2) $\forall (u, u')(v, v') \in R_D[(u, v) \in E_Q \Leftrightarrow (u', v') \in E_D]$.

To define CAR-dual simulation, we add cardinality condition to the children and parents with the same labels. The other definitions remain the same.

Definition 1: Pattern Q matches data graph G via *CAR-dual simulation*, denoted by $Q \preceq^{CD}_{sim} G$, if there is a relation R_D with respect to $Q \preceq^{D}_{sim} G$, and for every $(u, u') \in R_D$: (1) the number of children of u' in G_D with a particular label is not less than the number of children of u in Q with the same label; (2) the number of parents of u' in G_D with a particular label is not less than the number of parents of u in Q with the same label. □

2) CAR-tight Simulation: The only difference between CAR-tight simulation and tight simulation, defined in [5], is its construction based on CAR-dual simulation instead of dual simulation. To find the neighborhood of a vertex to enforce locality condition, it uses a concept called *ball* [7]. A ball b in G, denoted by $\hat{G}[c, r]$, is a subgraph of G that contains all vertices within distance r from the vertex c (including c). The vertex c is called the center of the ball, and the integer value r is called the radius of the ball. Moreover, the ball contains all edges in G that connect these vertices (i.e., it is an induced connected subgraph). Given two vertices u and v in a connected graph, the *distance* between them is defined as the minimum number of edges in an undirected path that connects them.

Definition 2: Pattern Q matches data graph G via *CAR-tight simulation*, denoted by $Q \preceq^{CT}_{sim} G$, if there are vertices $u \in Q$ and $u' \in G$ such that (1) u is a center of Q with highest defined selectivity; (2) $(u, u') \in R_D$ where R_D is maximum dual match set with respect to $Q \preceq^{CD}_{sim} G$; (3) $Q \preceq^{CD}_{sim} \hat{G}_D[u', r_Q]$ with maximum dual match set R^b_D, where $\hat{G}_D[u', r_Q]$ is a ball extracted from $G_D(V_D, E_D, l_D)$. G_D is the *result dual match graph* with respect to $Q \preceq^{CD}_{sim} G$, and r_Q is the radius of Q; (4) u' is a member of at least one of the pairs in R^b_D. □

The connected part of the result match graph of each ball with respect to its R^b_D that contains u' is called a *maximum perfect subgraph (MaxPG)* of G with respect to Q. The subgraph results of the model are actually these MaxPGs. The criterion for selectivity of u in Q is the ratio of its degree to its label frequency.

After this point, when we mention dual or tight simulation, we mean their new forms unless it is explicitly expressed. It is

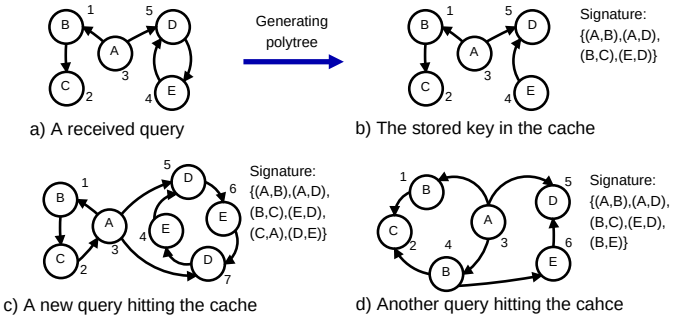


Fig. 3. An example for the procedure of caching and reusing the results

straightforward to show that *CAR-tight simulation*, preserves all the nice features of the old tight simulation including the fact that it contains all the results of subgraph isomorphism.

V. Pattern containment and results reuse

In this section, we first introduce our mechanism based on pattern containment for reusing the results of old subgraph pattern matching queries to answer other new queries. Then, we present the theoretical detail to show why our approach can always retrieve all the correct results.

A. The caching mechanism

As it was mentioned earlier, we store pairs of polytree-subgraph in the cache space. We show that any new query that is a *cover-tight match* to a polytree can be answered using its corresponding subgraph. We call graph Q_2 *cover-tight match* to graph Q_1, if there is a tight-simulation relation from Q_1 to Q_2 and this relation covers all the vertices in Q_2.

Figure 3 shows the main idea. The initially received query is displayed in subfigure 3(a). Its extracted polytree is illustrated in part (b). After storing the pair of this polytree and its pattern matching result based on tight simulation in the cache, queries in (c) and (d) can be answered without using the data graph because they are *cover-tight matches* to the polytree.

The designed system has two main modules, each containing several steps as follows.

1) Storing the result of an old query: When a query cannot be answered using the cache contents, we store its corresponding key-value item in the cache space. In order to create the key-value element:

- We first find the spanning polytree of the query graph.
- We perform a CAR-dual simulation for the polytree on the data graph, and find the set of vertices in the data graph that are present in the resulting dual-relation set. We will show in the next section that this set of vertices are the same as the set of vertices in the maximum perfect subgraphs if we had performed CAR-tight simulation.
- We extract an induced subgraph of the data graph using the set of vertices found in the previous step.
- The polytree and the induced subgraph will be stored in the cache space as a key-value pair.
- In order to facilitate the later search in the cache, the *signature* of the new polytree is calculated. Here, we define the *signature* of each graph as its set of *label-edges*. An *label-edge* for an edge is the pair of the labels of its source vertex and its target vertex. The signature of the graphs in figure 3 are displayed for example. We store a map from any created signature to the set of polytrees with the same signature. Therefore, the new polytree will be added to such a set of polytrees with the same signature.

2) Search in the cache: When a new query is received, the cache space should be searched for any old polytree that is a cover-tight match to this query. Clearly, it can be very time consuming to check all the polytrees that are available in the cache space. Therefore, we use the stored signatures for a simple filtering technique. The main steps are:

- The signature of the new query is calculated.
- The signature of the new query is compared against all the signatures stored in the cache to find the set of candidate match polytrees. There are two comparison conditions: (1) the two signatures must have exactly the same set of individual labels; (2) the signature of the polytree must be a subset of the signature of the new query.
- Any polytree found through the previous step as a candidate match will be compared with the new query to check for cover-tight match. Indeed, the new query might be cover-tight match with several polytrees; this means that any of them can be used to answer the new query. Therefore, when the first tight-cover match is found the other candidates will be ignored.
- When a cover-tight match is found among candidate polytrees, its corresponding value in the cache space, which is an induced subgraph of the data graph, will be used to perform tight simulation and the results will be returned as the final results.
- When there is no cover-tight match present in the cache for the new query, it should be answered using the original data graph.

B. Proof of correctness

In this section, we present the theoretical concepts used to prove the correctness of our approach. Here, we explain the terms only for CAR-dual and CAR-tight simulation, but they are also valid for the previous version of these models. Theorem 1 guarantees existence of pattern containment under the aforementioned conditions. In order to prove the theorem, we first need to define a few new concepts and prove a few lemmas.

Definition 3: A graph Q_2 is a *cover-tight match* to another graph Q_1 when all of its vertices are present in the MaxPGs returned by $Q_1 \trianglelefteq^{CT}_{sim} Q_2$. □

Definition 4: Assuming R_D as the maximum dual match set for $Q \trianglelefteq^{CD}_{sim} G$, we define $R_{minD} \subseteq R_D$ as a *minimum dual match set* when (1) it is a complete match set; i.e., $u \in V_Q \Rightarrow \exists u' \in G : (u, u') \in R_{minD}$ (2) removing any vertex of G that

participate in the match set from R_{minD} makes it incomplete. We also call the subgraph of G corresponding to a R_{minD}, a *minimum dual match subgraph*. □

Lemma 1: Any maximum perfect subgraph produced by CAR-tight simulation is a union of some minimum dual match subgraphs. □

Lemma 2: When a query graph Q is a polytree with diameter d_Q, none of its minimum dual match subgraphs on G has a diameter bigger than d_Q. □

Proposition 1: When Q is a polytree, the set of vertices in the MaxPGs resulting by $Q \trianglelefteq^{CT}_{sim} G$ is the same as the set of vertices in the result dual match graph resulting by $Q \trianglelefteq^{CD}_{sim} G$. □

Lemmas 1 and 2 are used to prove proposition 1, and then the proposition is used to prove theorem 1. Moreover, this proposition indicates that for extracting the induced subgraph, which should be stored for a polytree in the cache, it would be enough to run dual simulation instead of tight simulation. It should be noticed that dual simulation is a preliminary step in calculating tight simulation; therefore, the extra cost of ball creation would be avoided. For the same reason, it would be enough to run dual simulation when a new query should be compared to the existent polytree to realize if it is a cover-tight match for that polytree.

Theorem 1: Consider G as a data graph and Q_1 as a query graph. Given that P is a spanning polytree of Q_1, and V_P is the set of all vertices in the MaxPGs resulting by $P \trianglelefteq^{CT}_{sim} G$, for Q_1 or any other new query Q_i that is a cover-tight match to P, the set of the vertices, V_R, in the MaxPGs resulting by $Q_i \trianglelefteq^{CT}_{sim} G$ is a subset of V_P ($V_R \subseteq V_P$). □

VI. Empirical studies

We have conducted several sets of experiments to evaluate the effectiveness of the proposed caching technique. We call a query that can be answered by cache a *hit query*. We calculate catch hit-rate by finding the percentage of hit queries in a workload of queries. Moreover, the *response time* of a query is the elapsed time from submitting the query until receiving its results either from the cache or main engine.

A. Experimental setup

We have implemented a basic cache system in Java. A data graph is provided in adjacency-list format from a text file. The queries are also fed to the system by their files, which have the same format. All the experiments were performed on a machine that has 128GB DDR3 RAM, and two 2GHz Intel Xeon E5-2620 CPUs, each with 6 cores. Moreover, we used JDK-1.7.0_55 to compile and run our program.

We have used three real-world labeled graphs for our experiments: (a) Patent graph [21], a dataset on US patents where each vertex represents a patent and an edge from i to j means that patent i cites patent j. There are about 3.7M vertices and 16M edges in this graph. The label of the vertices are 37 different subcategories of the patents. (b) Citation graph [22], a citation network of papers where an edge from vertex i to vertex j means paper i has cited paper j. It has about 2.2M vertices and 4.3M edges. We selected publishing year of the papers as the label of vertices, which means 80 different labels. (c) Amazon product co-purchase network [18], with about 548K vertices and 1.7M edges. Each vertex is a product, and an edge from i to j means that product i is frequently co-purchased with product j. We used the first category of each product as its label, which led to total of 104 distinct labels.

To generate a query with guaranteed result in the data graph, we randomly extract a connected subgraph from a given dataset. Our query generator has a pair of input parameters $(n, \bar{d})$, where n is the number of vertices in the query, and $\bar{d}$ is the desired average degree of each vertex. The program first randomly picks a vertex of the data graph, and then randomly extracts between 0 and $\bar{d}$ neighbors of this vertex regardless of the edge-direction. It continues extracting the local vertices in a BFS-fashion until the requested number of vertices are added to the subgraph.

B. Experimental results

The experiments are categorized in three groups. First, we study the potential speedup that can be achieved by reusing the results of queries. Then, we examine cache hit rate and performance improvement achieved by our caching technique for a synthesized workload. Third, we show the effect of *Cardinality Restriction* on the size of output results and the running time of the algorithms.

For our experiments, we randomly extracted a set of base queries from each data graph. They are from five different sizes: $(10, 3)$, $(15, 3)$, $(20, 4)$, $(25, 5)$, and $(30, 5)$. We randomly extracted 100 queries for each size; hence, each base query set contains 500 unique queries.

1) Speedup: We use the base query set of each data graph to test the potential speedup that can be achieved by our caching mechanism. We first measure the response time of each individual query without using cache system. Then, we submit each query again when its corresponding cache contents is created and stored in the cache space. The *cache speedup* for each query equals to the ratio of its response times without and with cache hit. Table II displays the average speedup of queries for each data graph. The speedup depends on many features in both data graph and query graph. Intuitively, one can expect a relation between the *number of vertices in the result of a query* and its *cache speedup*. The Pearson correlation of these two variables are displayed in table II. The degrees of freedom in this experiment is 488. Considering a significance level 0.01 and two-tailed statistical test, the critical value is 0.115. It indicates a statistically significant negative correlation between the two variables. We can use this property to determine a criterion for the queries that are not worth storing their results in the cache.

Data points in the charts of figure 4 show the cache speedup of base queries versus the number of vertices in their results, which is normalized with the total number of vertices in the data graph. We found that the Power regression is a good choice for presenting these data points. After drawing the Power-regression curve, we can select 0.7% as a common optimal criterion for storing the results of queries in the cache; i.e., we do not consider the queries in which the normalized number of vertices in their results is bigger than this limit. In our experiments, the percentage of the queries in the base

query set that should not be considered for caching according to this criterion, are 9.6%, 1.2%, and 8.8% respectively for Amazon, Citation, and Patent graphs.

TABLE II
SPEEDUP ACHIEVED BY CACHING

Data graph	Amazon	Citation	Patent
Average speedup	99	487	887
Pearson correlation between the *number of vertices in the results* and the *cache speedup* of individual queries	-0.37	-0.39	-0.29

2) Cache Hit: To evaluate the proposed caching technique, an appropriate workload of queries is needed. Because of the lack of such a workload, we had to create our own. In order to provide an appropriate *test-set* of queries for each data graph, we use the set of base queries to create similar queries. We incrementally add a new vertex to each base query and connect it to a random number of old vertices with random directions. The label of the new vertex is also randomly selected from the set of available labels. We continue this incremental process until 5 steps; therefore, we synthesize 5 new query graphs for each base query. At the end, we have 2500 newly synthesized queries. This approach for generating new queries resembles the behavior of real users who tweak their old queries to explore more interesting patterns. It is noteworthy to mention that there is no guarantee that newly synthesized queries meet the similarity condition (cover-tight match) in order to be eligible for pattern containment. Moreover, many of these synthesized queries may not have any match in the data graph similar to the real-life situations. We form the *test-set* from the union of the base and the synthesized queries; hence, it contains 3000 unique queries. The workload is a sequence of queries randomly sampled from the the *test-set* with replacement. In a real-life situation, popular queries may be submitted many times. To simulate this situation, we set the number of queries in the workload 5 times bigger than the number of unique queries; i.e., the workload comprises of 15000 queries.

Figure 5a illustrates the measured hit-rate versus different cache sizes. The horizontal axis displays the cache size based on its ratio to the total number of unique queries in the *test-set*, and the vertical axis shows the percentage of hit queries to the total number of queries in the workload. *Least frequently used (LFU)* policy is implemented for *replacement* when the cache space becomes full. As it is expected, the hit-rate goes towards saturation after some point.

Improvement in the average response time of CAR-tight simulation queries versus the cache size is displayed in figure 5b. It can be observed that the behavior of Citation graph has been slightly different from the other two data graphs; e.g., slower increase in the hit-rate, and higher ratio between average response times in minimum and maximum cache sizes. This difference can be explained using features of the graph; i.e., Citation graph is sparser than the others and has less number of undirected cycles. These features cause shorter response times, less number of vertices in the results, and less similarity among the queries in the *test-set*. Less similarity among the queries will lead to lower hit-rate, and having smaller subgraphs corresponding to the polytrees makes response times from cache even faster.

Table III presents more information about this experiment. It shows that the main overhead of the cache system, which is search time, is negligible in our tests. Average store time includes the spent times for extracting the polytree, finding its CAR-tight simulation results in data graph, and extracting corresponding induced subgraph. These steps can be executed in parallel to the main engine; hence, the store time is not considered in the response time.

TABLE III
MORE INFORMATION ABOUT CACHE BEHAVIOR

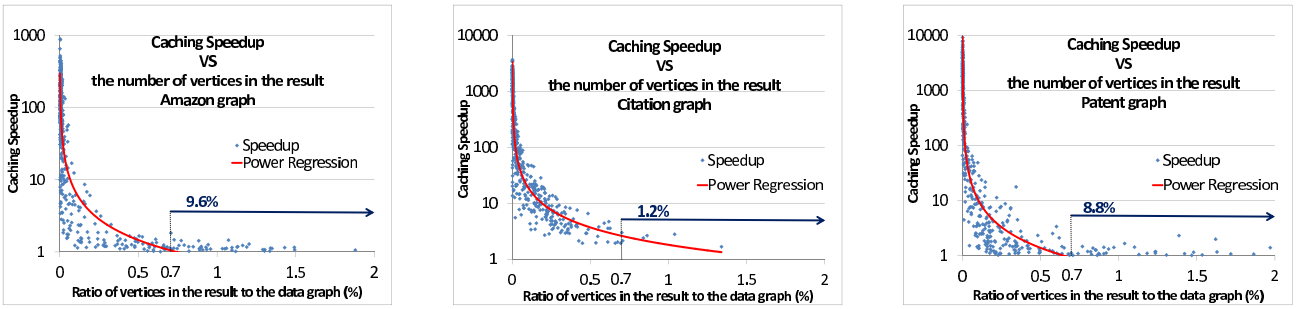
Data graph	Amazon	Citation	Patent
Average number of vertices in cache for each polytree	990	478	5281
Average search time in cache (msec)	2	6	2
Average store time in cache (msec)	567	793	4315

We have also tested cache performance for subgraph isomorphism. We used *DualIso* algorithm [23] to find the first 1000 subgraph isomorphic matches. Exactly the same caching mechanism used in this test; however, we ran *DualIso* on data graph when the query did not hit the cache, and on its correspondent subgraph when it hit the cache. The average response time versus the size of the cache is illustrated in figure 5c.

Figure 5d shows the warm-up behavior of the cache for different data graphs. The horizontal axis is the percentage of queries submitted from the workload, and the vertical axis is the cumulative percentage of the queries that hit the cache. As one may expect, there is a transition phase for warm-up and then the curve becomes linear. Comparing with figure 5a, it can also be observed that the pace of the transition is faster for a data graph with higher hit-rate.

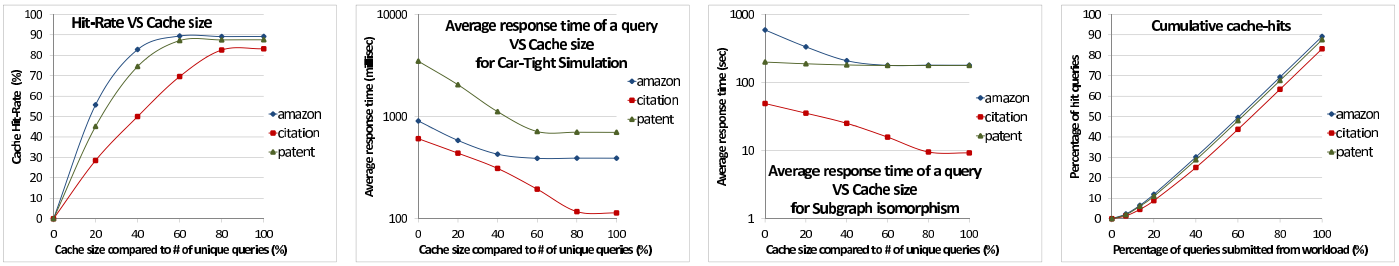
3) Cardinality Restricted models: As expected, our experiments summarized in figure 6 show that the results of CAR-dual and CAR-tight simulation models are more stringent than their older counterparts. We used the base query set and performed dual, CAR-dual, tight, and CAR-tight simulation for each query graph, and measured the percentage of decrease in the number of vertices in the results. For CAR-tight simulation, we also measured the decrease in the number of MaxPGs. As it is displayed in figure 6a, we observed more than 25% decrease in the average number of vertices in the results of both models for all the three data graphs. Moreover, the average number of MaxPGs in CAR-tight simulation decreased significantly for Amazon and Patent data graphs. The number of MaxPGs has dropped faster than the number of their containing vertices because of the post-processing phase of tight simulation. At this phase, all the MaxPGs that are the superset of any other one are filtered. While some MaxPGs share many vertices in common, they may not be filtered because of their small differences. CAR-tight simulation usually removes these small differences; therefore, a higher number of MaxPGs are filtered.

The cost of achieving more meaningful and more stringent results by CAR modification is longer running time. Figure 6b shows the percentage of increase in the running time of dual and tight simulation after CAR modification. Clearly, the running time of the algorithm for CAR-dual is longer than dual because it needs to check extra conditions. However, it is more likely that a decrease in the number of balls can compensate the extra cost of CAR-dual to some extent. Indeed, we could



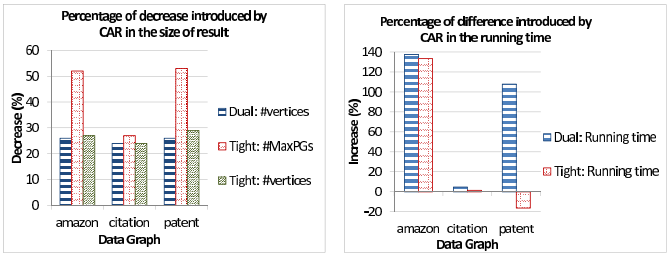
(a) Amazon graph (b) Citation graph (c) Patent graph

Fig. 4. Cache speedup



(a) Hit-rate (b) Average response time for CAR-tight simulation (c) Average response time for subgraph isomorphism (d) Cumulative cache hits

Fig. 5. Cache performance



(a) More stringent results) (b) Difference in running time

Fig. 6. Effect of CAR modification

observe that the running time of CAR-tight decreased about 16% on average for Patent data graph.

VII. Related Work

Subgraph pattern matching is extensively studied during last decades [24], [1]. Nevertheless, the recently proposed pattern matching models in graph simulation family promise a new avenue to tackle old problems [3]. They are especially shown to be useful for processing massive graphs of social networks [2].

The concept of dual simulation first was introduced in [7] and used to define strong simulation model. Several other more efficient subgraph pattern matching models such as strict [6] and tight [5] simulations are also defined based on dual simulation. The new CAR-dual simulation model, introduced in this paper, improves the expressiveness of the original model. Moreover, all the other models relying on that can be updated to have more stringent results. In this paper, we only present the updated version of tight simulation, called CAR-tight.

There are rare previous studies about caching techniques for generic subgraph pattern matching queries. In the field of semantic web, many RDF engines cache intermediate results of SPARQL queries in order to speedup the computation of other queries when they have common triple patterns [25]. These techniques rely on triple structure of RDF datasets and the fact that a SPARQL query is mainly composed of triple patterns. Then, they try to mitigate the cost of expensive joins between the results of common triple patterns. Martin et al. [13] have also proposed a caching system for SPARQL queries where they cache the entire query. Nevertheless, their cache system is *content-blind*; i.e., a new query can be answered using the cache only if it is identical to an old query stored in the cache. Indeed, their contribution is about cache maintenance; that is, updating the cache contents according to the changes in the underlying knowledge bases.

A *content-aware* caching technique for SPARQL queries is introduced in [14]. The authors propose a few conditions for containment checking of conjunctive SPARQL queries with simple filter conditions. The first difference from our work is that their work is about SPARQL queries. Properties of SPARQL queries, which fall in the category of path pattern queries, are different from the type of subgraph pattern queries that we study in this research. Because of the nature of SPARQL queries and the way they store their results, they also need to define *evaluability* notion because containment conditions cannot guarantee that a new query can be answered using the cache contents.

Another related work has been published very recently

in [15]. The authors have investigated pattern containment problem for *graph simulation* [17] and *bounded simulation* [3] models. They statically generate a set of *views* from a given data graph. Each view represents an interesting part of the data graph. Then, they define their containment conditions and show how the queries can be evaluated to learn if their answers are contained in a view. Their experiments eventually shows that the response time of the queries will improve when they can be answered using initially defined views. The main difference to our work lies in the intrinsic differences of *view* and *cache*. A cache space for graph queries has a dynamic characteristic; therefore, new issues like cache size, cache hit rate, and query replacement arise that are not considered under concept of views. Moreover, our work is based on a different pattern matching model that suits a completely different set of applications.

VIII. Conclusions and Future Work

The main focus of this research is about graph pattern containment. We have proposed a novel pattern containment approach based on tight simulation model. Furthermore, we have designed and implemented an efficient cache system based on this technique. This is one of the first works that investigates caching techniques for subgraph pattern matching queries.

Although the employed filtering mechanism for searching the cache has been very efficient in our experiments, it might be worthwhile for future work to investigate more complex indexing algorithms in order to support very heavy workloads in extremely massive datasets. This problem is very similar to the problem that is studied under the title of *supergraph query processing* in the literature [26]. However, the problem in the context of our research has two main distinguishing differences: (1) All the available index systems are only based on subgraph isomorphism; (2) unlike the previous works the index should be formed and evolved dynamically for graphs in a cache space.

The cache replacement policy that we have implemented in this project is very similar to the policies in other contexts. Nevertheless, it seems appealing to design new replacement policies specifically revised for subgraph pattern matching.

Real-world data graphs are time evolving [11]; i.e., there are minor changes in their structure through the time. The content of cache for pattern matching queries should be updated according to these changes using incremental algorithms in order to support time evolving data graphs. Although there are some preliminary studies [8], incremental algorithms for subgraph pattern matching are in their infancy stages.

Acknowledgment

This research has been partially funded by the National Science Foundation under Grant Number CNS-1338276. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the NSF.

References

[1] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," *AAAI FS*, vol. 6, pp. 45–53, 2006.

[2] J. Brynielsson, J. Hogberg, L. Kaati, C. Mårtenson, and P. Svenson, "Detecting social positions using simulation," in *ASONAM*. IEEE, 2010, pp. 48–55.

[3] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 264–275, Sep. 2010.

[4] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[5] A. Fard, M. U. Nisar, J. A. Miller, and L. Ramaswamy, "Distributed and scalable graph pattern matching: Models and algorithms," *International Journal of Big Data (IJBD)*, vol. 1, no. 1, 2014.

[6] A. Fard, M. Nisar, L. Ramaswamy, J. Miller, and M. Saltz, "A distributed vertex-centric approach for pattern matching in massive graphs," in *Big Data Conference*, Oct 2013, pp. 403–411.

[7] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 310–321, Dec. 2011.

[8] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *ACM SIGMOD*, 2011, pp. 925–936.

[9] W. Fan, "Graph pattern matching revised for social network analysis," in *ICDT*, 2012, pp. 8–21.

[10] M. Nisar, A. Fard, and J. Miller, "Techniques for graph analytics on big data," in *BigData Congress*, June 2013, pp. 255–262.

[11] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller, "Towards efficient query processing on massive time-evolving graphs," in *CollaborateCom*, oct. 2012, pp. 567 –574.

[12] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," in *ACM SIGMOD*, 2013, pp. 1185–1196.

[13] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with sparql query caching," in *The Semantic Web: Research and Applications*, 2010, vol. 6089, pp. 304–318.

[14] Y. Shu, M. Compton, H. Mller, and K. Taylor, "Towards content-aware sparql query caching for semantic web applications," in *WISE*, 2013, vol. 8180, pp. 320–329.

[15] W. Fan, X. Wang, and Y. Wu, "Answering graph pattern queries using views," in *ICDE*, March 2014, pp. 184–195.

[16] R. Milner, *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[17] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *FOCS*, 1995, pp. 453–.

[18] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. Web*, vol. 1, no. 1, May 2007.

[19] M. Farber, "On diameters and radii of bridged graphs," *Discrete Mathematics*, vol. 73, no. 3, pp. 249–260, 1989.

[20] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, Dec. 2003.

[21] B. H. Hall, A. B. Jaffe, and M. Trajtenberg, "The nber patent citation data file: Lessons, insights and methodological tools," NBER Working Paper 8498, Tech. Rep., 2001.

[22] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, "Arnetminer: Extraction and mining of academic social networks," in *ACM SIGKDD*, 2008, pp. 990–998.

[23] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs," in *BigData Congress*. IEEE, 2014, pp. 498–505.

[24] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[25] T. Lampo, M.-E. Vidal, J. Danilow, and E. Ruckhaus, "To cache or not to cache: The effects of warming cache in complex sparql queries," in *OTM*, 2011, vol. 7045, pp. 716–733.

[26] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise, "Similarity search on supergraph containment," in *ICDE*, March 2010, pp. 637–648.