Fast Search for Dynamic Multi-Relational Graphs

Sutanay Choudhury
Pacific Northwest National
Laboratory, USA
sutanay.choudhury@pnnl.gov

Lawrence Holder
Washington State University,
USA
holder@wsu.edu

George Chin
Pacific Northwest National
Laboratory, USA
george.chin@pnnl.gov

John Feo
Pacific Northwest National
Laboratory, USA
john.feo@pnnl.gov

ABSTRACT

Acting on time-critical events by processing ever growing social media or news streams is a major technical challenge. Many of these data sources can be modeled as multi-relational graphs. Continuous queries or techniques to search for rare events that typically arise in monitoring applications have been studied extensively for relational databases. This work is dedicated to answer the question that emerges naturally: *how can we efficiently execute a continuous query on a dynamic graph?* This paper presents an exact subgraph search algorithm that exploits the temporal characteristics of representative queries for online news or social media monitoring. The algorithm is based on a novel data structure called the *Subgraph Join Tree (SJ-Tree)* that leverages the structural and semantic characteristics of the underlying multi-relational graph. The paper concludes with extensive experimentation on several real-world datasets that demonstrates the validity of this approach.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

Continuous Queries; Dynamic Graph Search; Subgraph Matching

1. INTRODUCTION

Social networks, social media websites and mainstream news media are driving an exponential growth in online content. This information barrage presents both a formidable challenge and an opportunity to applications that thrive on situational awareness. Examples of such applications include emergency response, cyber security, intelligence and finance where the data stream is monitored continuously for specific events. Timeliness of the detection carries paramount importance for such applications. The applications derive their competitive edge from fast detection as late detection may not have much value due to incurred damage to resources. Our work is motivated by queries that look for rare events, have a time constraint on the time to discovery and never need a bulk retrieval of historic data due to their monitoring nature.
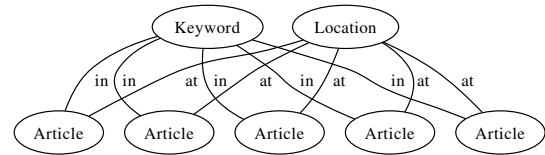
Figure 1: A graph query for monitoring emergencies in social media and news streams.

Continuous queries evolved in the field of relational databases to address applications with precisely the above characteristics. A *continuous query system* is defined as one where a query logically runs continuously over time as opposed to being executed intermittently [1, 2]. Thus, continuous query processing is data-driven or trigger oriented. Many of the prominent news or social media streams can be represented as multi-relational data sources. Multi-relational graphs are often an attractive representation for data sources with sparsity. The problem of monitoring events in such data streams can be viewed as continuously searching the dynamic graph for patterns that represent events of interest.

Fig. 1 shows a graph pattern that represents such an event. An operator may substitute the "keyword" with "fire" or "accident" and register several queries. Articles refer to articles in news as well as social media posts. This query will capture events that are reported from the same location. Observe that we specify the label for only one vertex in this query, the rest of the vertices have only type specified. Therefore, we are using the labeled vertex to anchor into a context and report when multiple events with that context are detected in the data stream.

Graph search involves finding exact or approximate matches for a query subgraph in a larger graph. It has been studied extensively and is formally defined as the problem of *subgraph isomorphism*: given a pattern or query graph (henceforth described as query graph) G_q and a larger input graph (henceforth described as the data graph) G_d, find all isomorphisms of G_q in G_d. Following the definition of isomorphism, the matching involves finding a one-to-one correspondence between the vertices of a subgraph of G_d and vertices of G_q such that all vertex adjacencies are preserved. Now consider the challenges in applying traditional graph search techniques to this problem. Unless carefully adapted, a standard search function will search the entire data graph repeatedly and retrieve the same search results. Also, many of the best performing graph search algorithms rely on indexing the graph. Even with an interval as large as 5 minutes, rebuilding the index of a massive graph repeatedly is infeasible. This motivates exploration of in-

cremental algorithms for continuous queries, although the general problem of incremental subgraph isomorphism is proven to be NP-complete as well [3].

Queries like the one shown in Fig. 1 share a number of common attributes. First, they all involve an implicit time window to suggest the timeliness aspect associated with the query. Clearly, the length of the time window varies depending on the application context. The average monitoring time window for a high volume social media stream may be in tens of minutes whereas the equivalent period for online news may be in hours or days. Second, all these queries aim to discover a number of temporal events that share the same context, such as a common set of keywords and location. Lastly, a multi-relational graph often takes the form of k-partite graphs [4,5] where each partite set represents a group of entities of the same type. For queries as ones described in Fig. 1, each event that is represented by an article or a tweet can be viewed as a k-partitite subgraph.

We exploit these three features to implement a continuous query processing framework for multi-relational graphs. First, by utilizing a rolling time window we continuously prune partial search results that would otherwise need to be tracked and would contribute to the combinatorial growth in memory utilization. Second, the temporal property of the vertices and edges representing events suggests that it is logical to search for distinct subgraphs where such "temporal" vertices or edges are ordered, thus significantly reducing the search space. Finally, we take advantage of the multi-relational structure of the data and the characteristics of temporal events to avoid expensive joins. Given a multi-relational query graph we decompose it in a hierarchical fashion. We design a data structure called the *Subgraph Join Tree*, or henceforth referred as the *SJ-Tree* to model the hierarchical decomposition and store matches with various subgraphs of the query graph as represented in the tree. This paper demonstrates the validity of this decomposition approach towards query processing. Automated construction of SJ-Tree from an arbitrary query graph is not discussed in this paper. We refer to the smallest units of the decomposed query graph as "search primitives", which almost always consists of more than one edge. As new edges arrive over time, we continuously perform (a) "local searches" to look for matches with the search primitives and (b) use the decomposition structure to "join" them into progressively larger matches. This represents a middle ground between the periodic application of a graph search algorithm on the data graph and the approach that would have been employed by a traditional stream database. Stream databases have no alternative but to model each edge in the query graph as a separate join operator. Our model can express this degenerate case where an edge is represented as a search primitive in the SJ-Tree, but the performance is extremely poor. By grouping subgraphs into search primitives, we can simplify the query plan, significantly improve performance by multiple orders of magnitude, and perhaps most importantly, reason about the trade-offs involved and explore a large space of possible optimizations.

1.1 Contributions

Our contributions from this research are summarized below.

1. We introduce a data structure called *SJ-Tree* for query graph decomposition (section 4) and present a novel subgraph search algorithm (5) for continuous queries on dynamic multi-relational graphs.

2. We compare our performance with the incremental subgraph isomorphism algorithm developed by Fan et al. [3] and show that our approach provides improvements by multiple orders of magnitude (section 6).

3. We present a series of experiments on representative online

news (New York Times), co-authorship networks (DBLP) and social media data sources (Tencent Weibo) modeled as multi-relational graphs. The scale of these datasets are orders of magnitude larger than previously reported research [3,6] in the literature (section 6).

4. We present a theoretical model for complexity analysis of the search algorithm. We also provide an extensive experimental analysis of the algorithm's performance as a function of the frequency distribution of vertex labels for verification of the theoretical model.
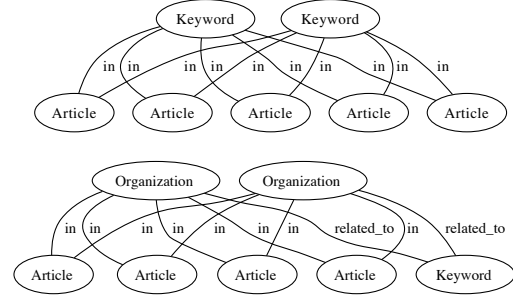

Figure 2: More examples of monitoring queries on multi-relational graphs. The query at the top can be used to discover events in a certain context. Set one of the keywords to "Oil" and run the query to discover various events that center around oil, such as price movements, discovery, accident etc. By setting the keyword to "buyout", the bottom query can be used to detect when news surface about a merger between two companies.

1.2 Problem Statement

Every edge in a dynamic graph has a timestamp associated with it and therefore, for any subgraph g of a dynamic graph we can define a time duration $\tau(g)$ which is equal to the duration between the earliest and latest edge belonging to g. Given a dynamic multi-relational graph G_d, a query graph G_q and a time window t_W, we report whenever a subgraph g_d that is isomorphic to G_q appears in G_d such that $\tau(g_d) < t_W$. The isomorphic subgraphs are also referred to as *matches* in the subsequent discussions. If $M(G_d^k)$ is the cumulative set of all matches discovered until time step k and E_{k+1} is the set of edges that arrive at time step $k+1$, we present an algorithm to compute a function $f(G_d, G_q, E_{k+1})$ which returns the incremental set of matches that result from updating G_d with E_{k+1} and is equal to $M(G_d^{k+1}) - M(G_d^k)$. We assume that the graph only receives edge inserts and no deletions.

2. BACKGROUND

2.1 Multi-Relational Graphs

Single relational graphs have been widely used to model systems comprised of homogeneous elements related by a single type of relation. A social network where vertices represent people and edges represent connections between people is an example of a single-relational graph. A multi-relational graph becomes a useful construct for modeling heterogeneous relations between a possibly heterogeneous set of entities.

DEFINITION 2.1.1 **Multi-Relational Graph** A multi-relational graph denoted as $G = (V, E)$, is a graph representation of a multi-relational database. If the database contains K entity types as $\xi_1, ...\xi_K$, then the vertex set $V(G)$ is partitioned into K sets $V_1, ..., V_K$. For any vertex $v \in V_k$, the label for the vertex is represented from the domain of the entity type ξ_k. The edges of the graph are the

relations between various entities as indicated in an entity-relation model. Thus, an edge in the graph $e \in E(G), e = (v_i, v_j)$ is an instance of a relation R_{ij} between entities ξ_i and ξ_j.

A graph representation of such a multi-relational database takes the form of a K-partite graph [5], if there are no relations between homogeneous entities or equivalently, if there are no edges between vertices that belong to the same partite set. In practice, such relationships are not rare. Examples of such linkages are citation links between articles and social ties between two members in a network. However, we omit unary relationships from our multi-relational model. Our omission of unary relationships is driven by usability and a desire for simplicity. Fig. 1 and 2 show a number of examples embodying a range of events. Consider the example in Fig. 2 that detects a series of articles that refer to the same set of keywords; one may wish to introduce unary relationships in the graph to indicate citation between articles and thus, focus only on articles with high citation counts. However, such queries can be alternatively represented by adding a query constraint to the vertices that require them to have a minimum degree. Or, such relations could be represented using an intermediate vertex of a different type. Thus, for the scope of this work we define patterns of interest as query graphs that are subgraphs of the K-partite multi-relational graph.

2.2 Continuous Queries

A continuous query can be described as computing a function f over a stream S continuously over time and notifying the user whenever the output of f satisfies a user-defined constraint [1]. They are distinguished from ad-hoc query processing by their high selectivity (looking for unique events) and need to detect newer updates of interest as opposed to retrieving lots of past information. In this paradigm the primary objective is to notify a listener as soon as the query is matched. One may view conventional databases as passive repositories with large collections of data that work in a request-response model whereas continuous queries are data-driven or trigger oriented. These features coupled with real-time demands challenge many of the fundamental assumptions for conventional databases and establish continuous query processing on relational data streams as a major research area. The literature on database research from the past two decades is abundant with work on continuous query systems [7,8]. Babcock et al. [9] provide an excellent overview of continuous query systems and their design challenges.

2.3 Graph Queries

Graph querying techniques have been studied extensively in the field of pattern recognition over nearly four decades [10]. Our work is focused on *subgraph isomorphism* which is as defined as follows.

DEFINITION 2.2.1 **Subgraph Isomorphism** Given the query graph G_q and a matching subgraph of the data graph (G_d) denoted as G'_d, a matching between G_q and G'_d involves finding a bijective function $f : V(G_q) \to V(G'_d)$ such that for any two vertices $u_1, u_2 \in V(G_q), (u_1, u_2) \in E(G_q) \Rightarrow (f(u_1), f(u_2)) \in E(G'_d)$.

Two popular subgraph isomorphism algorithms were developed by Ullman [11] and Cordella et al. [12]. The VF2 algorithm [12] employs a filtering and verification strategy and outperforms the original algorithm by Ullman. Over the past decade, the database community has focused strongly on developing indexing and query optimization techniques to speed up the searching process. A common theme of such approaches is to index vertices based on k-hop neighborhood signatures derived from labels and other properties such as degrees, spectral properties and centrality [13–16]. Other major areas of work involve join-order optimization [17] and search
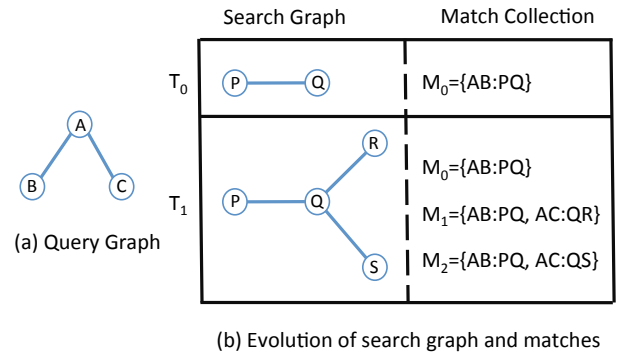

(a) Query Graph

(b) Evolution of search graph and matches

Figure 3: Illustration of a naive incremental algorithm. Assme AB matches with PQ, and AC matches with both QR and QS.

techniques for alternative representations such as similarity search in a multi-dimensional vector space [18].

3. RELATED WORK

Investigation of subgraph isomorphism for dynamic graphs did not receive much attention until recently. It introduces new algorithmic challenges because we can-not afford to index a dynamic graph frequently enough for applications with real-time constraints. In fact this is a problem with searches on large static graphs as well [19]. There are two alternatives in that direction. We can search for a pattern repeatedly or we can adopt an incremental approach. The work by Fan et al. [3] presents incremental algorithms for graph pattern matching. However, their solution to subgraph isomorphism is based on the repeated search strategy. Chen et al. [6] proposed a feature structure called the *node-neighbor tree* to search multiple graph streams using a vector space approach. They relax the exact match requirement and require significant preprocessing on the graph stream. Our work is distinguished by its focus on temporal queries and handling of partial matches as they are tracked over time using a novel data structure. From a data-organization perspective, the SJ-Tree approach has similarities with the Closure-Tree [20]. However, the closure-tree approach assumes a database of independent graphs and the underlying data is not dynamic. There are strong parallels between our algorithm and the very recent work by Sun et al. [19], where they implement a query-decomposition based algorithm for searching a large static graph in a distributed environment. Our work is distinguished by the focus on continuous queries that involves maintenance of partial matches as driven by the query decomposition structure, and optimizations for real-time query processing.

4. INCREMENTAL QUERY PROCESSING

4.1 Naive Approach

We begin with a simplistic solution to motivate an incremental approach for continuous query processing. For every new edge that is added to G_d, we detect if the edge matches any edge in the query graph. This check can be performed minimally by examining 1) if there are edges in the query graph with the same type as the new edge and 2) if the endpoint vertices of the new edge match with the corresponding edges in the query graph based on their types and attributes. Once an edge is considered as a matching candidate, the next step is to consider different combinations of matches it can participate in.

A simple illustration of this matching process is shown in Fig. 3. While intuitively simple, this approach falls prey to combinatorial explosion very quickly. It finds the match with the query graph at the cost of creating many partial matches. Assume that the G_d receives a large number of edges that match with the query graph edge between A and B (Fig. 3a). Let's denote this edge as e_{AB}. Therefore, a large number of partial matches will be created with mapping information for e_{AB}. Subsequently, every future edge that matches with e_{AC} will need to be matched or checked against all the existing partial matches for augmenting into a larger match. While the subgraph matching problem has an inherent exponential nature associated with it, a better algorithm will restrict the growth of the number of partial matches to track and still produce the correct result.

4.2 Our Approach

Our objective is to introduce an approach that guides the search process to look for specific subgraphs of the query graph and follow specific transitions from small to larger matches. Following are the main intuitions that drive this approach,

1. Instead of looking for a match with the entire graph or just any edge of the query graph, partition the query graph into smaller subgraphs and search for them.

2. Track the matches with individual subgraphs and combine them to produce progressively larger matches.

3. Define a *join order* in which the individual matching subgraphs will be combined. Do not look for every possible way to combine the matching subgraphs.

Although the current work is completely focused on temporal queries, the graph decomposition approach is suited for a broader class of applications and queries. The key aspect here is to search for substructures without incurring too much cost. Even if some subgraphs of the query graph are matched in the data, we will not attempt to assemble the matches together without following the join order. Thus, if there are substructures that are too frequent, joining them and producing larger partial matches will be too expensive without a stronger guarantee of finding a complete match. On the other hand, if there is a substructure in the query that is rare or indicates high selectivity, we should start assembling the partial matches together only after that substructure is matched.

4.3 Subgraph Join Tree (SJ-Tree)

We introduce a tree structure called *Subgraph Join Tree (SJ-Tree)* that supports the above intuitions for implementing a join order based on selectivity of substructures of the query graph.

DEFINITION 4.1.1 A SJ-Tree T is defined as a binary tree comprised of the node set N_T. Each $n \in N_T$ corresponds to a subgraph of the query graph G_q. Let's assume V_{SG} is the set of corresponding subgraphs and $|V_{SG}| = |N_T|$. Additional properties of the SJ-Tree are defined below.

PROPERTY 1. The subgraph corresponding to the root of the SJ-Tree is isomorphic to the query graph. Thus, for $n_r = root\{T\}$, $V_{SG}\{n_r\} \equiv G_q$.

PROPERTY 2. The subgraph corresponding to any internal node of T is isomorphic to the output of the join operation between the subgraphs corresponding to its children. If n_l and n_r are the left and right child of n, then $V_{SG}\{n\} = V_{SG}\{n_l\} \bowtie V_{SG}\{n_r\}$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the join operation is defined as $G_3 = G_1 \bowtie G_2$, such that $G_3 = (V_3, E_3)$ where $V_3 = V_1 \cup V_2$ and $E_3 = E_1 \cup E_2$.
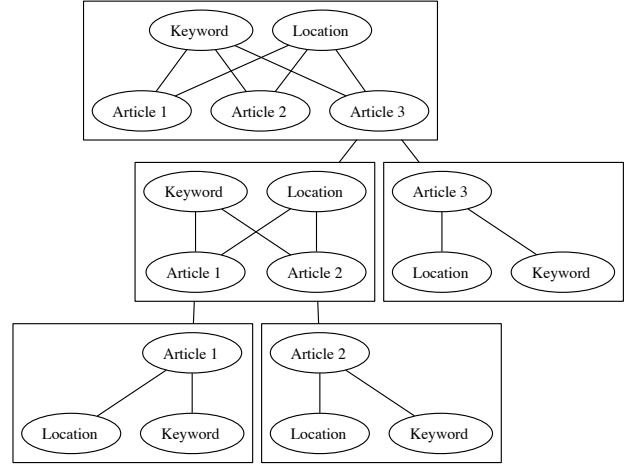

Figure 4: Illustration of query decomposition in SJ-Tree.

PROPERTY 3. Each node in the SJ-Tree maintains a set of matching subgraphs. We define a function $matches(n)$ that for any node $n \in N_T$, returns a set of subgraphs of the data graph. If $M = matches(n)$, then $\forall G_m \in M, G_m \equiv V_{SG}\{n\}$.

PROPERTY 4. Each internal node n in the SJ-Tree maintains a subgraph, CUT-SUBGRAPH(n) that equals the *intersection* of the query subgraphs of its child nodes.

For any internal node $n \in N_T$ such that CUT-SUBGRAPH(n) $\neq \emptyset$, we also define a *projection operator* Π. Assume that G_1 and G_2 are isomorphic, $G_1 \equiv G_2$. Also define Φ_V and Φ_E as functions that define the bijective mapping between the vertices and edges of G_1 and G_2. Consider g_1, a subgraph of G_1: $g_1 \subseteq G_1$. Then $g_2 = \Pi(G_2, g_1)$ is a subgraph of G_2 such that $V(g_2) = \Phi_V(V(g_1))$ and $E(g_2) = \Phi_E(E(g_1))$.

Conceptually, the SJ-Tree is an index structure where keys track the occurrence of matching subgraphs in the data graph. Our decision to use a binary tree as opposed to an n-ary tree is influenced by the simplicity and lowering the combinatorial cost of joining matches from multiple children. Analyzing the trade-offs between different tree models (e.g. left-deep vs bushy) is part of future work.

5. CONTINUOUS QUERY ALGORITHM

We present a subgraph search algorithm (Algo. 1 and 2) that utilizes the SJ-Tree structure (referred to as T). The search process is illustrated in Fig. 5. The input to PROCESS-CONT-QUERY is the dynamic graph G_d, the SJ-Tree (T) corresponding to the query graph and the set of incoming edges. Each leaf of the SJ-Tree represents an unique subgraph of the query graph. Lines 4-6 in Algo. 1 describe the search for each of these unique subgraphs around every incoming edge. Any discovered match is added to the SJ-Tree (line 9).

5.1 Local Search

The LOCAL-SEARCH function performs a subgraph isomorphism check around the neighborhood of every incoming edge e. The query decomposition often reduces the local search to performing star queries where the center of the query is the vertex representing a temporal event. The peripheral vertices of the star query are the other entities that represent various attributes of the event. Further, in the context of real-time search, if the current time is t and the query specifies a time window of length t_W then all edges that have a timestamp older than $(t - t_W)$ are ignored from
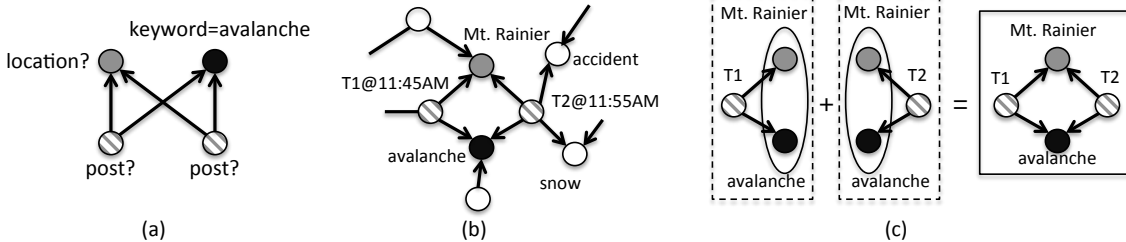
Figure 5: a) Example query b) Occurrence of a match in the search graph c) Combining two partial matches to form the complete match

---

Algorithm 1 PROCESS-CONT-QUERY(G_d, T, edges)

1: $leaf\text{-}nodes$ =GET-LEAF-NODES(T)
2: **for all** $e_s \in edges$ **do**
3:     UPDATE-GRAPH(G_d, e_s)
4:     **for all** $n \in leaf\text{-}nodes$ **do**
5:         g_{sub}^q =GET-QUERY-SUBGRAPH(T, n)
6:         $matches$ =LOCAL-SEARCH(G_d, g_{sub}^q, e)
7:         **if** $matches \neq \emptyset$ **then**
8:             **for all** $m \in matches$ **do**
9:                 T.UPDATE-SJ-TREE(n, m)

---

the search. In addition to filtering search candidates, we also periodically prune the SJ-Tree to remove partial matches that are older than t_W from the current time.

Algorithm 2 UPDATE-SJ-TREE($node, m$)

1: $sibling = sibling[node]$
2: $parent = parent[node]$
3: k =GET-JOIN-KEY(CUT-SUBGRAPH$[parent], m$)
4: H_s = match-tables$[sibling]$
5: M_s^k = GET(H_s, k)
6: **for all** $m_s \in M_s^k$ **do**
7:     m_{sup} = JOIN($m_s m$)
8:     **if** $parent = root$ **then**
9:         PRINT('MATCH FOUND : ', m_{sup})
10:     **else**
11:         UPDATE-SJ-TREE($parent, m_{sup}$)
12: ADD($matches[node], m$)
13: ADD($match-tables[node], k, m$)

5.2 Partial Match Join and Aggregation

This subsection describes the process outlined in UPDATE-SJ-TREE. The SJ-Tree data structure maintains the sibling and parent information for every node as distinct arrays (Algo. 2, line 1-2). Each node in the SJ-Tree maintains a hash table that supports storing multiple objects with the same key. This collection of tables are denoted by the match-tables property of the SJ-Tree (Algo 2., line 4). GET() and ADD() provides lookup and update operations on the hash tables. Whenever a new matching subgraph g (available as a property of the partial match m) is added to a node in the SJ-Tree, we compute a key using its projection ($\Pi(g)$) and insert the key and the matching subgraph into the hash table. The projection is obtained by hashing a string representation of the subgraph.

When a new match is inserted into a leaf node we check to see if it can be combined (referred as JOIN()) with any matches that are contained in the collection maintained at its sibling node. A suc-

cessful combination of matching subgraphs between the leaf and its sibling node leads to the insertion of a larger match at the parent node. This process is repeated as long as larger matching subgraphs can be produced by moving up in the SJ-Tree. A complete match is found when two matches belonging to the children of the root node are combined successfully.

The JOIN operation between partial matches is critical to the overall query processing performance. Suppose we have a query that finds a sequence of two events with a common set of attributes. Assume that two matching events $event_1$ and $event_2$ are found with timestamps τ_1 and τ_2 respectively, with $\tau_1 < \tau_2$. For all practical purposes we can report the sequence $\{event_1, event_2\}$ and ignore the out of order combination. Therefore, given two partial matches M_1 and M_2 with edge sets $\{E_i, E_j\}$ and $\{E_m, E_n\}$ respectively, the join algorithm rejects all combinations of these two sets that do not represent a monotonic order based on timestamps. This is accomplished by computing a range of timestamps for each partial match. If $t_{low}[M_i]$ and $t_{high}[M_i]$ are the lowest and highest timestamp for match M_i, then we require that $t_{high}[M_1] < t_{low}[M_2]$ for joining M_1 and M_2.

5.3 Complexity Analysis

There are two primary tasks in processing every edge in the continuous query algorithm, (1) performing a local search for a small subgraph of the query graph and in case of a successful search, (2) updating the SJ-Tree with the partial match. For the multi-relational queries described in this paper, the local search reduces to performing a star query. Assuming the LOCAL-SEARCH is cheap for star queries, we can approximate the cost of the continuous query processing for every edge to a small constant in case of a failed local search and to that of the UPDATE-SJ-TREE() for a successful search. If C_k is the expected value for the cost of insert and joins at node k in the SJ-Tree, then the complexity of updating the tree is $O((\bar{C_k})^h)$, where $\bar{C_k}$ denotes the expected value of C_k over all nodes. This is not a tight bound, and a more accurate bound can be obtained estimating C_k based on the frequency of the subgraphs satisfying the label constraints in the query graph. Accurate estimation of the frequency of an arbitrary subgraph is hard. Therefore, we resort to obtaining a loose bound in terms of the label constraints. Assume that v_q has the lowest degree among all labeled vertices in the query graph and v_s is the corresponding vertex in the data graph. Then n_k is bounded by $\binom{d_s}{d_q}$, where d_s and d_q are the degrees of v_s and v_q. The storage complexity for SJ-Tree is $O(h\bar{C_k}|E(G_q)|)$.

6. EXPERIMENTAL RESULTS

This section is dedicated to answering two questions: 1) How does our continuous graph query algorithm compare with the state of the art? To answer this, we compare our algorithm's perfor-
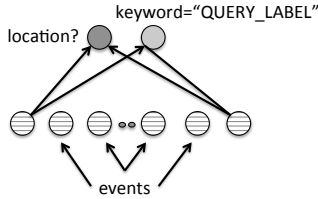
Figure 6: The query template used to find temporal events. Experiments are performed using queries with 4 event vertices and 2 feature vertices. One feature vertex is labeled and all other vertices specify only types.

mance with the IncIsoMatch algorithm presented in [3]. 2) How does our query algorithm perform on real-world datasets? We provide the answers from exhaustive experimentation on three real-world datasets through systematic query selection.

Graph dataset	vertices	edges	vertex types	edge types
New York Times	39,523	68,682	4	4
DBLP	3.158M	3.26M	2	1
Tencent Weibo	2.5M	89.6M	4	5

Our metric is the time to process increments of a fixed number of edges (1k or 100k) whichever is closer to 1% of the test dataset size. The times reported only include the time spent in the query processing section. We use the query template as shown in Fig. 6. To develop a performance model in terms of the label distribution, we sample the degree distribution of every vertex type and divide the range of the degree distributions into ten intervals. For each interval, five closest candidate vertices are selected for testing purposes. Selection of multiple vertices around each bin allows us to systematically observe the impact of increasing the degree of the labeled vertex in the query graph.

6.1 Experimental Setup

The results were obtained by using a single core on a 48-core shared memory system comprising 2.3 GHz Opteron 6176 SE processors and 256 GB RAM. The processor cache size is 512KB and each system node has 32 GB RAM. The code was compiled with g++ 4.1.2-52 with -O3 optimization flag on Linux 2.6.18.

6.2 New York Times

We use a news dataset from New York Times collected over Aug-Oct 2011 [1]. Each article in the dataset contains a number of facets that belong to four type of entities. Each of the articles and facets are represented as vertices in the graph. Each edge that connects an article with a facet carries a timestamp that is the publication time of the article. Following the template shown in Fig. 6, we run a query that finds four articles where all the articles have a common keyword and location. For the location vertex we specify the labels shown in the top diagram in Fig. 7 and observe the performance. As the figures indicate, selecting labels that correspond to vertices with increasing degrees increases the running time of the query.

Next, we compare our approach with the IncIsoMatch algorithm described by Fan et al. [3]. The VF2 algorithm [12] is adapted to implement the graph search functionality as outlined in IncIsoMatch. We specify a label on the feature marked with † and select a label with one of the highest degrees for that vertex type. The queries are as follows: 1) Find four articles with a common keyword and a common organization†, 2) Find four articles with a
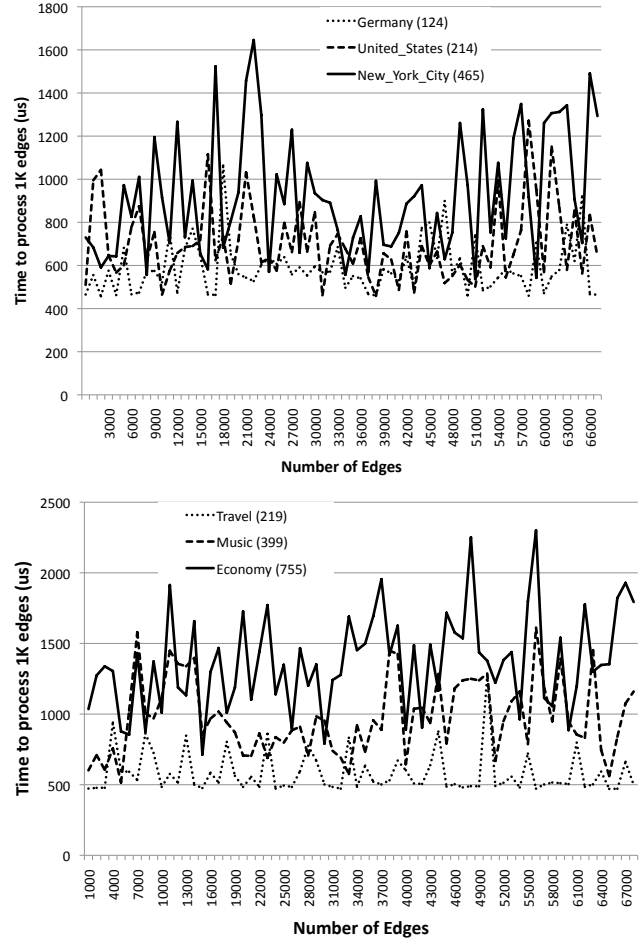
---

http://data.nytimes.com

Figure 7: Results from queries finding four articles with a common keyword and location. Legends indicate degree of query label.
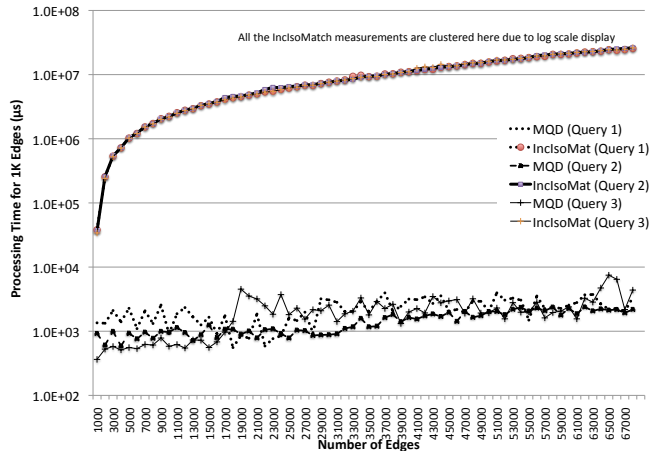

Figure 8: Comparison with IncIsoMatch. MQD refers to the Multi-Relational Query Decomposition algorithm from this paper.
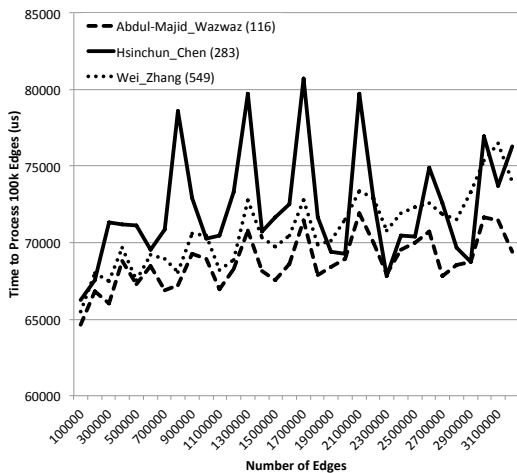
Figure 9: Performance results for queries on the DBLP dataset. The spikes in the plot can be attributed to the bursty nature of scientific publishing where authors target the same group of conferences and journals every year.

Figure 10: Query processing time for the Tencent Weibo dataset for queries with varying selectivity.

common entity and a common keyword † and 3) Find four articles with a common entity and a common location †. Fig. 8 shows a performance improvement from our algorithm by several orders of magnitude. The multiple orders of improvement in performance is attributed to the strictly ordered aggregation of partial matches in the SJ-Tree and the temporal property based optimizations. The performance gap between the processing time of the two algorithms increases as the graph grows larger. We attribute this to the nature of the IncIsoMatch where it performs a search around every new edge in the graph. The search spans all vertices around the endpoints of the new edge as long as they are within k-hops, where k is the diameter of the query graph. As the data graph grows denser, even for a query graph with small or modest size, the k-hop subgraph accumulates a large number of edges and the search becomes increasingly expensive.

6.3 DBLP Co-Authorship Network

We build a multi-relational graph representation of the DBLP citation network [1] with two types of entities: authors and articles. The author name and the title of the article are stored as labels of respective vertices. We run a query to find an author (author 1) who has co-authored four papers with a specified author (author 2). Following the previously shown query template, our query graph has four article vertices and two author vertices. Only one author vertex is labeled. We observe the degree distribution of the "author" vertices and select names with progressively increasing degrees. The results are shown in Fig. 9. It can be seen that the performance of the algorithm is quite stable for a modestly large network with nearly 3M+ edges. Additionally, the results show that even though vertex degree is a good indicator of the query performance, there are other factors at play. The graph describes author-article relationship; therefore, the degree of an author vertex provides the number of authored articles. It does not provide the information about the number of co-authors of a person. Searching for a person who publishes a given number of articles with fewer co-authors will lead to more partial matches and increase per-edge query processing time. The consistent high processing times for the query
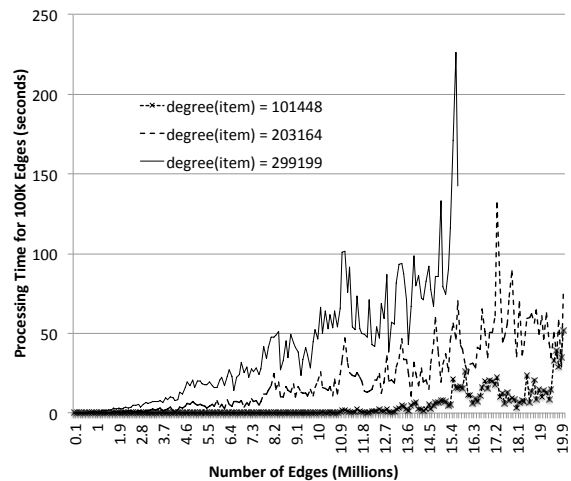
---

[1]dblp.uni-trier.de/xml

containing "Hsinchun-Chen" despite smaller degree in the graph is a result of this aspect.

6.4 Social Media

Finally, we present our results on a data set collected from Tencent Weibo, a Chinese microblogging social network[2]. The data set provides a temporal history of item recommendations to registered users of the social network. We build a graph with 4 vertex types (users, items, keywords and categories) and 5 edge types (item-in-category, item-has-keyword, item-reco-accept, item-reco-reject and user-profile-has-keyword).

Our test query is to detect a series of item acceptances by a group of users described by a common keyword. Following the previously shown query template, our query graph has four user vertices and one item and keyword vertex. We specify a label on the item and seek to discover the keyword that characterize the users accepting or rejecting that item. The results are shown in Fig. 10. The figure suggests a clear trend. It shows that as the graph grows large the query processing time eventually rises sharply. It also shows that the rise happens earlier for low-selectivity queries where the specified label has higher degree in the graph. This is because the number of partial matches grows rapidly in the event of a successful search around a high degree vertex. Every partial match from the past can potentially be merged with the latest partial match, and the partial match collection grows combinatorially over time.

This brings us to implementing the temporal window based pruning. We select the query with the highest degree label (degree(item) = 299199, Fig. 10) for which the rise in the processing time was sharpest. We set the time window t_W to 1 day and prune the SJ-Tree after processing every 5 million edges. The results from the windowing enabled search is shown in Fig. 11. Observe that the peaks in the processing time are smaller than ones observed in Fig. 10 by an order of magnitude.

This is an extremely promising result for practical applications. Figure 11 suggests that it would take 10 seconds on average to process 100k edges for a query with very low selectivity. This translates into a throughput of 0.01 million edges/second or 864 million edges per day. At the time of this writing, high volume data streams such as Twitter receive nearly 300-400 million posts ev-

---

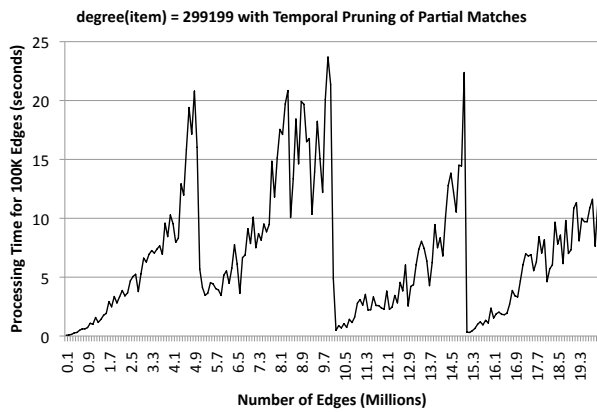[2]www.kddcup2012.org/c/kddcup2012-track1

Figure 11: Query processing time for the Tencent Weibo dataset with temporal match pruning applied on every 5 million edges.

ery day. Considering that every user action translates into multiple edges in a graph, one may expect around billions of edges everyday. The throughput can be expected to be much higher for a query with moderate selectivity. Thus, we believe this level of throughput on a very low-selectivity query gets us close to executing real-time graph queries on such high volume data streams.

7. CONCLUSION AND FUTURE WORK

We present a novel graph decomposition based approach for continuous queries on multi-relational graphs. We introduce the SJ-Tree structure, whose nodes represent the hierarchical decomposition of the query graph. The SJ-Tree systematically tracks the evolving matches in the data graph as they transition from smaller to larger matches based on the query graph decomposition. We present experimental analysis on several real-world datasets such as New York Times, DBLP and Tencent Weibo and show that our SJ-Tree based algorithm coupled with temporal optimizations clearly outperforms the state of the art [3] by multiple orders of magnitude. Our experiments demonstrate that it is possible to execute complex multi-relational graph queries in a real-time setting. To our knowledge, the results presented in this paper are the best reported performance for such queries. These initial results are highly promising in that they suggest possible ways of auto-selecting optimal values for query processing parameters based on the data distribution. Our main theoretical contribution is to demonstrate that for a prominent class of multi-relational queries where the *local search* is cheap, we can execute graph queries in time that is exponential to the height of the SJ-Tree.

Development of query planning algorithms to generate a SJ-Tree for any query graph by exploiting its structural and semantic characteristics is the next logical step. Query planning for 1) complex graph queries where a complete temporal ordering may not be possible, 2) trade-offs between different query decomposition strategies and 3) exploring different query classes and determining the optimal trade-off between local search and joins in the SJ-Tree represent areas of future work.

Acknowledgment

8. REFERENCES

[1] Y.-N. Law, H. Wang, and C. Zaniolo, "Relational languages and data models for continuous queries on sequences and data streams," *ACM Trans. Database Syst.*, June 2011.

[2] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," *SIGMOD Rec.*, 1992.

[3] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," ser. SIGMOD '11, 2011.

[4] P. Zhao, X. Li, D. Xin, and J. Han, "Graph cube: on warehousing and olap multidimensional networks," in *SIGMOD '11*.

[5] E. Spyropoulou and T. D. Bie, "Interesting multi-relational patterns," in *ICDM*, 2011, pp. 675–684.

[6] L. Chen and C. Wang, "Continuous subgraph pattern search over certain and uncertain graph streams," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, no. 8, pp. 1093–1109, Aug. 2010.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," ser. SIGMOD '03.

[8] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: The stanford stream data manager," in *SIGMOD '03*.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," ser. PODS '02.

[10] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Intl. Journal of Pattern Recognition and Artificial Intelligence*, 2004.

[11] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, pp. 31–42, January 1976.

[12] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Trans. on Pattern Analysis and Machine Intelli.*, 2004.

[13] Y. Tian and J. Patel, "Tale: A tool for approximate large graph matching," in *ICDE '08*.

[14] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," ser. KDD '07.

[15] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB.*, vol. 3, pp. 340–351, September 2010.

[16] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, and X. Lin, "High efficiency and quality: large graphs matching," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, ser. CIKM '11.

[17] L. Zou, L. Chen, and M. T. Özsu, "Distance-join: pattern match query in a large graph database," *PVLDB*, vol. 2, no. 1, Aug. 2009.

[18] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao, "Neighborhood based fast graph search in large networks," ser. SIGMOD '11.

[19] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, 2012.

[20] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," ser. ICDE '06.