A Selectivity based approach to Continuous Pattern Detection in Streaming 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

Khushbu Agarwal
Pacific Northwest National
Laboratory, USA
khushbu.agarwal@pnnl.gov

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

ABSTRACT

Cyber security is one of the most significant technical challenges in current times. Detecting adversarial activities, prevention of theft of intellectual properties and customer data is a high priority for corporations and government agencies around the world. Cyber defenders need to analyze massive-scale, high-resolution network flows to identify, categorize, and mitigate attacks involving networks spanning institutional and national boundaries. Many of the cyber attacks can be described as subgraph patterns, with prominent examples being insider infiltrations (path queries), denial of service (parallel paths) and malicious spreads (tree queries). This motivates us to explore subgraph matching on streaming graphs in a continuous setting.

The novelty of our work lies in using the subgraph distributional statistics collected from the streaming graph to determine the query processing strategy. We introduce a "Lazy Search" algorithm where the search strategy is decided on a vertex-to-vertex basis depending on the likelihood of a match in the vertex neighborhood. We also propose a metric named "Relative Selectivity" that is used to select between different query processing strategies. Our experiments performed on real online news, network traffic stream and a synthetic social network benchmark demonstrate 10-100x speedups over selectivity agnostic approaches.

1. INTRODUCTION

Social media streams and cyber data sources such as computer network traffic are prominent examples of high throughput, dynamic graphs. Application domains such as cyber security, emergency response, national security put a premium on discovering critical events as soon as they emerge in the data. Thus, processing streaming updates to a dynamic graph database for real-time situational awareness is an important research problem. These particular data sources are also distinguished by their natural representation as heterogeneous or multi-relational graphs. For example, a social media data stream contains a diverse set of entity types such as person, movie, images etc. and relations such as (*friendship, like etc.*). For cyber-security, a network traffic dataset can be modeled as a graph where vertices represent IP addresses and edges are typed by classes of network traffic [12]. Our work is focused on continuous querying of these dynamic, multi-relational graphs.
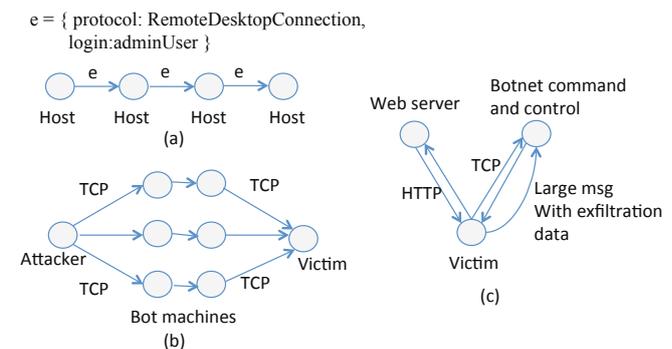
Figure 1: Graph based descriptions of attack patterns. a) Insider infiltration: This pattern shows how an attacker may move laterally inside an enterprise, b) Denial of Service attack, c) Information exfiltration: Victim browses a compromised website. This downloads a script which establishes communication with the botnet command and control.

For social networks, we are often inundated with the stream of updates. Unless we choose to stay constantly connected to the social networks, it is highly desirable to report only the important patterns/events as they occur in the data; for example, we may choose to ask "tell me when two friends are meeting at a nearby location". The stakes are much higher in the cyber-security domain. As the volume and throughput of network traffic or event log datasets rise exponentially, the lack of ability to detect adversarial actions in real-time provides an asymmetric advantage to attackers. Internet backbone traffic collected by CAIDA [1]), which we use later as a dataset in our experiments typically accumulate 40 million packets every minute. In a study titled "Data Breach Investigations Report", US communications company Verizon analyzed 100,000 security incidents from the past decade and concluded that 90% of the incidents fell into ten attack patterns. A number of these attacks

[1] http://www.caida.org

can be naturally described as graph patterns. Figure 1 shows graph based patterns for a number of these attacks. Organizations such as internet service providers, content delivery networks etc. that receive network traffic from a wide area network are ideally poised to search for these attack patterns. Although there exists a significant number of graph databases and graph processing frameworks that scale to billion edge graphs, none of them support real-time subgraph pattern matching as a primary feature. Periodic export of network traffic flow or event alerts from log aggregation tools to a graph database, followed by post-attack querying on the static graph database is the most common workflow today. Despite cyber security being a multi-billion dollar market worldwide, the research on providing real-time querying capability on a **single, large streaming graph** is rather scarce.

Continuous querying of a dynamic graph raises a number of unique challenges. Indexing techniques that preprocess a graph and speed up queries are expensive to periodically recompute in a dynamic setting. Periodic execution of the query is an obvious solution under this condition, but the effectiveness of this approach will reduce as the interval between query executions shrinks. Also, periodic searching of the entire graph can be wasteful where the query match emerges slowly because we will find a partial match for the query every time we search and potentially redo the work numerous times. Very recent publications by Gao et al [7] and Mondal and Deshpande [15] presents algorithms for implementing continuous queries on graphs. This motivates us to study the problem of subgraph pattern matching in a streaming setting. We want to register a pattern as a graph query and continuously perform the query on the data graph as it evolves over time.

In addition to the cyber attack patterns in Figure 1, social queries are also drawn from LSBench, a benchmark for reasoning on streaming SPARQL data. A common theme that emerges is that all these query graphs are heterogenous in nature. They are composed of different edge types (in cyber security) as well as different node and edge types (in social media). None of the previous work on continuous pattern detection has addressed this issue of heterogeneity. Exploiting the heterogeneity in both the query graph and the data graph stream, and improving over heterogeneity agnostic continuous pattern detection approaches is the primary contribution of our research. The primary ideas behind our approach is described below. We believe the simplicity of our approach is its greatest strength, and it will allow easy adoption of our optimizations into the distributed system implementations developed by others in the field.
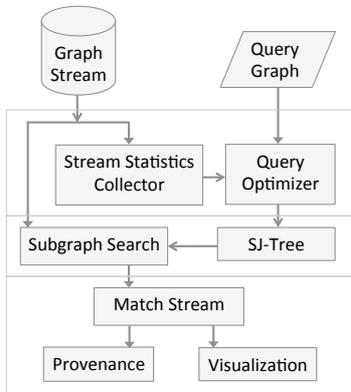
Figure 2: Framework for subgraph pattern matching on streaming graphs.

Figure 2 provides an overview of our approach. We approach the problem from an incremental processing perspective where search happens locally on every edge arrival. We do not search for the entire query graph around every new edge arriving in the stream. Given a query graph, the *query optimizer* decomposes it into smaller subgraphs as ordered by their *selectivity*. The selectivity information is obtained using the single-edge level and 2-edge path distribution obtained from the graph stream (section 5). We store the resulting decomposition into a data structure named SJ-Tree (Subgraph Join Tree) (section 3) that tracks matching subgraphs in the data graph. For a new edge in the graph, we always search for the *most selective* subgraph of the query graph. For other subgraphs of the query graph, a search is triggered if and only if a match for the previous subgraph in the selectivity order was obtained in the neighborhood of the new edge. This algorithm named "Lazy Search" is described in section 4. We introduce two metrics, *Expected and Relative Selectivity*, that captures the effectiveness of a given query decomposition (section 5). Further, we demonstrate how these metrics can be used to reason about the performance from different decompositions and select the best performing strategy.

1.1 Contributions

The most important takeaway from our work is that even as the subgraph isomorphism problem is NP-complete, it is possible to perform efficient continuous queries on dynamic graphs by exploiting the heterogeneity in the data and query graph. More specific contributions from the paper are listed below.

1. We present a dynamic graph search algorithm that demonstrates speedup of multiple orders of magnitude with respect to the state of the art.

2. We introduce two selectivity metrics for query graphs that are estimated using efficiently obtainable distributional statistics of single edge and 2-edge subgraphs from the graph stream.

3. We present an automatic query decomposition algorithm that selects the best performing strategy using the aforementioned graph stream statistics and *Relative Selectivity*.

Our observations are supported by experiments on datasets from three diverse domains (online news, computer network traffic and a social media stream).

2. BACKGROUND AND RELATED WORK

This section is aimed at providing an overview of the related field and provide the context for the studied problem. We begin with introducing the key concepts.

Multi-Relational Graphs We define a graph G as an ordered-pair $G = (V, E)$ where V is the set of vertices and the E is the set of edges that connect the vertices. In the following, we use $V(G)$ and $E(G)$ to indicate the set of vertices and edges associated with a graph G. A *labeled graph* is a six-tuple $G = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where Σ_V and Σ_E are sets of distinct labels for vertices and edges. λ_V and λ_E are vertex and edge labeling functions, i.e. $\lambda_V : V \to \Sigma_V$ and $\lambda_E : E \to \Sigma_E$.

Dynamic Graphs We define *dynamic graphs* as graphs that are changing over time through edge insertion or deletion. 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 interval $\tau(g)$ which is equal to the interval between the earliest and latest edge belonging to g. We focus on directed, labeled dynamic graphs with multi-edges in this work. The graph is maintained as

a window in time. Given a time window t_W, edges are deleted as they become older than $t_{last} - t_W$, where t_{last} is the timestamp of the newest edge in the graph.

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 $f : V(G_q) \rightarrow V(G_d')$ such that for any two vertices $u_1, u_2 \in V(G_q)$, $(u_1, u_2) \in E(G_q) \Rightarrow (f(u_1), f(u_2)) \in E(G_d')$.

2.1 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. Assume that G_d^k is the data graph at time step k. 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 $M(G_d^{k+1}) - M(G_d^k)$.

2.2 Related Work

Graph querying techniques have been studied extensively in the field of pattern recognition over nearly four decades [4]. Two popular subgraph isomorphism algorithms were developed by Ullman [20] and Cordella et al. [5]. The VF2 algorithm [5] 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 and centrality [17, 18, 23]. Other major areas of work involve exploration of subgraph equivalence classes [8] and search techniques for alternative representations such as similarity search in a multi-dimensional vector space [13]. Apart from neighborhood based signatures, *graph sketches* is an important area that focuses on generating different synopses of a graph data set [22]. Development of efficient graph sketching algorithms and their applications into query estimation is expected to gain prominence in the near future.

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 [16]. 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. [6] presents incremental algorithms for graph pattern matching. However, their solution to subgraph isomorphism is based on the repeated search strategy. Chen et al. [2] 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 pre-processing 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 [9]. 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. [16], where they implement a query-decomposition based algorithm for searching a large static graph in a distributed environment. Here 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. Mondal and Deshpande [15] propose solutions to supporting continuous ego-centric queries in a dynamic graph, Our work focuses on subgraph isomorphism, while [15] is primarily focused on aggregate queries. We view this as complementary to our work, and it affirms our belief that continuous queries on graphs is an important problem area, and new algorithms and data structures are required for its development.

The query pattern matching approach recently proposed in [7] is most closely related to our work with some important distinctions. The authors build a vertex centric, query processing engine for dynamic graphs on top of Apache Giraph, a distributed computing framework inspired by the Pregel framework. Their query decomposition approach is based on identifying optimal sub-DAGs (directed acyclic graph) in the query graph. The DAGs' are then traversed to identify source and sink vertices to define message transition rules in the Giraph framework. Although they address significant challenges inherent of processing dynamic graphs, it is not suitable for all types of queries. Specifically, queries that have cyclic communications, such as infiltration attack query in Figure 1 cannot be decomposed in DAG to find exact matches. Also, in our work we exclusively focus on query graphs with labeled edges with specific constraints. This are not addressed in the framework proposed in [7]. Our work makes no assumptions about the query graph structure and will find exact matches even when there is no apparent sink vertices. Moreover, the focus in [7] is on distributed implementation, while we focus on selectivity based query decomposition - that can improve performance for heterogeneous graphs. We show via edge distribution and selectivity plots that real world heterogeneous graphs have a strong skew in subgraph selectivity. The novelty of our work lies in estimating the selectivity of subgraphs from the graph stream and using the selectivity to determine the subgraph search strategy.

In summary, we consider these works to pursue two related but distinct directions that needs to be implemented in a scalable system.

3. A QUERY DECOMPOSITION APPROACH

We 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.

Figure 3 shows an illustration of the idea. Although the current work is completely focused on temporal queries, the graph decomposition approach is suited for a broader class of applications and
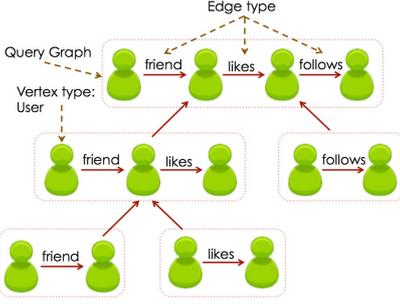
Figure 3: Illustration of the decomposition of a social query in SJ-Tree.

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.

The query decomposition approach can still suffer from having to maintain too many partial matches. If a subgraph of the query graph is highly *frequent*, we will end up tracking a large number of partial matches corresponding to that subgraph. Unless we have quantitative knowledge about how these partial matches transition into larger matches, we face the risk of tracking a large number of non-promising matching subgraphs. The "Lazy Search" approach outlined earlier in the introduction enhances this further. For any new edge, we search for a query subgraph if and only if it is the most selective subgraph in the query or if one of the either vertices in that edge participates in a match with the preceding (query) subgraph in the join order.

This section is dedicated towards introducing the data structures and algorithms for dynamic graph search. We begin with introducing the SJ-Tree structure (section 3.1) and then proceed to present the basic algorithms (Algorithm 1 and 2). The "Lazy Search"-enhanced version is introduced later in section 4. Automated generation of SJ-Tree is covered in section 5.

3.1 Subgraph Join Tree (SJ-Tree)

We introduce a tree structure called *Subgraph Join Tree (SJ-Tree)*. SJ-Tree defines the decomposition of the query graph into smaller subgraphs and is responsible for storing the partial matches to the query. Figure 3 shows the decomposition of an example query. Each of the rectangular boxes with dotted lines will be represented as a node in the SJ-Tree. The query subgraphs shown inside each "box" will be stored as a node property described below.

DEFINITION 3.1.1 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.

DEFINITION 3.1.2 A *Match* or a *Partial Match* is as a set of edge pairs. Each edge pair represents a mapping between an edge in a query graph and its corresponding edge in the data graph.

DEFINITION 3.1.3 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 $E_3 = E_1 \cup E_2$.

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 $V_{SG}\{n\} = V_{SG}\{n_l\} \bowtie V_{SG}\{n_r\}$.

Therefore, each leaf of the SJ-Tree represent subgraphs that we want to search for (perform subgraph isomorphism) on the streaming updates. Internal nodes in the SJ-Tree represents subgraphs that result from the joining of subgraphs returned by the subgraph isomorphism operations.

PROPERTY 3. Each node in the SJ-Tree maintains a set of *matches*. 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 $E(g_2) = \Phi_E(E(g_1))$.

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. With the properties of the SJ-Tree defined, we are now ready to describe the graph search algorithm.

3.2 Dynamic Graph Search Algorithm

Algorithm 1 DYNAMIC-GRAPH-SEARCH(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:         $matches$ =SUBGRAPH-ISO(G_d, g_{sub}^q, e_s)
7:         **if** $matches \neq \emptyset$ **then**
8:             **for all** $m \in matches$ **do**
9:                 UPDATE-SJ-TREE(T, n, m)

We begin with describing our dynamic graph search algorithm (Algorithm 1 and 2). The input to DYNAMIC-GRAPH-SEARCH is the dynamic graph so far G_d, the SJ-Tree (T) corresponding to the query graph and the set of incoming edges. Every incoming edge is first added to the graph (Algorithm 1, line 3). Next, we iterate over all the query subgraphs to search for matches containing the new edge (line 5-6). Any discovered match is added to the SJ-Tree (line 9).

Next, we describe the UPDATE-SJ-TREE function. Each node in the SJ-Tree maintains its sibling and parent node information (Algorithm 2, line 1-2). Also, each node in the SJ-Tree maintains a hash table (referred by the *match-tables* property in Algorithm 2, line 4). GET() and ADD() provides lookup and update operations on the hash tables. Each entry in the hash table refers to a *Match*. Whenever a new matching subgraph g 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 corresponding hash table (line 12). 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 successful 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 recursively (line 11) 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.

EXAMPLE Let us revisit Figure 3 for an example. Assuming we find a match with the query subgraph containing a single "friend" edge (e.g. {("George", "friend", "John")}), we will probe the hash table in the leaf node with "likes" edges. If the hash table stored a subgraph such as {("John", "likes", "Santana")}, the JOIN() will produce a 2-edge subgraph {("George", "friend", "John"), ("John", "likes", "Santana")}. Next, it will be inserted into the parent node with 2-edges. The same process will be subsequently repeated, beginning with the probing of the hash table storing matches with subgraphs with a "follows" edge.

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: $\quad m_{sup} = $JOIN($m_s, m$)
8: $\quad$ **if** parent = root **then**
9: $\quad\quad$ PRINT('MATCH FOUND : ', m_{sup})
10: $\quad$ **else**
11: $\quad\quad$ UPDATE-SJ-TREE($parent, m_{sup}$)
12: ADD(match-tables$[node], k, m$)

---

4. LAZY SEARCH

Revisiting our example from Figure 3, it is reasonable to assume that the "friend" relation is highly frequent in the data. If we decomposed the query graph all the way to single edges then we will be tracking all edges that match "friend". Clearly, this is wasteful. One may suggest decomposing the query to larger subgraphs. However, it will also increase the average time incurred in performing subgraph isomorphism. Deciding the right granularity of decomposition requires significant knowledge about the dynamic graph. This motivates us to introduce a new algorithmic extension.

Assume the query graph G_q is partitioned into two subgraphs g_1 and G_q^1. We use the notation G_q^k to indicate what remains of G_q after the k-th iteration in the decomposition process. If the probability of finding a match for g_1 is less than the probability of finding a match for G_q^1, then it is always desirable to search for g_1 and look for G_q^1 only where an occurrence of g_1 is found. Therefore, we select g_1 to be the most selective edge or 2-edge subgraph in the query graph and always search for g_1 around every new edge in the graph. Once we detect subgraphs in G_d that match with g_1, we follow the same approach to search for G_q in their neighborhood. We partition G_q^1 further into two subgraphs: g_2 and G_q^2, where g_2 is another 1-edge or 2-edge subgraph.

DATA STRUCTURES With the SJ-Tree, the partitioning of G_q is done upfront at the query compile time with g_1, g_2 etc becoming the leaves of the tree. The main difference between Lazy Search and that of Algorithm 2 is that we will be searching for g_2 only around the edges in G_d where a match with g_1 is found. Therefore, for every vertex u in G_d, we need to keep track of the g_i-s such that u is present in the matching subgraph for g_i. We use a bitmap structure M_b to maintain this information. Each row in the bitmap

refers to a vertex in G_d and the i-th column refers to g_i, or the i-th leaf in the SJ-Tree. If the search for subgraph g_i is enabled for vertex u in G_d, then $M_b[u][i] = 1$ and zero otherwise. Whenever a matching subgraph g' for g_i is discovered, we turn on the search for g_{i+1} for all vertices in $V(g')$. This is accomplished by setting $M_b[v][i + 1] = 1$ where $v \in V(g')$.

ROBUSTNESS WITH SUBGRAPH ARRIVAL ORDER Consider a SJ-Tree with just two leaves representing query subgraphs g_1 and g_2, with g_1 representing the *more selective* left leaf. The above strategy is not robust to the arrival order of matches. Assume g_1' and g_2' are subgraphs of G_d that are isomorphic to g_1 and g_2 respectively. Together, $g_1' \times g_2'$ is isomorphic to the query graph G_q. Because we are searching for g_1 on every incoming edge, g_1' will be detected as soon as it appears in the data graph. However, we will detect g_2' only if appears in G_d after g_1'. If g_2' appeared in G_d before g_1' we will not find it because we are not searching for g_2 all the time.

We introduce a small change to address this temporal ordering issue. Whenever we enable the search on a node in the data graph, we also perform a subgraph search around the node to find any match that has occurred earlier. Thus, when we find g_1 and enable the search for g_2 on every subsequent edge arrival, we also perform a search in G_d looking for g_1. This ensures that we will find g_2 even if it appeared before g_1.

Algorithm 3 summarizes the entire process. Lines 2-3 loop over all news edges arriving in the graph and update the graph. Next, given a new edge e_s, for each node in the SJ-Tree, we check to see if we should be searching for its corresponding subgraph around e_s (lines 4-8). The DISABLED() function queries the bitmap index and returns *true* if the corresponding search task is disabled. GET-QUERY-SUBGRAPH returns the query subgraph g_{sub}^q corresponding to node n in the SJ-Tree (line 9). Next, we search for g_{sub}^q using a subgraph-isomorphism routine that only searches for matches containing at least one of the end-point vertices of e_s (u and v, mentioned in line 5-6). For each matching subgraph found containing u or v, we enable the search for the query subgraph corresponding the sibling of n in the SJ-Tree. If n was not left-deep most node in the SJ-Tree, then we also query the left sibling to probe for potential join candidates (QUERY-SIBLING-JOIN(), line 16). Any resultant joins are pushed into the parent node and the entire process is recursively repeated at one level higher in the SJ-Tree.

5. SJ-TREE GENERATION

Here we address the topic of automatic generation of the SJ-Tree from a specified query graph. We begin with introducing key definitions, followed by the decomposition algorithm.

DEFINITION **Subgraph Selectivity** Given a large typed, directed graph G, the selectivity of a typed, directed subgraph g with k-edges (denoted as $S(g)$) is the ratio of the number of occurrences of g and the total number of all k-edge subgraphs in G. Instances of g may overlap with each other.

DEFINITION **Selectivity Distribution** The selectivity distribution of a set of subgraphs G_k is a vector containing the selectivity for every subgraph in G_k. The subgraphs are ordered by their frequencies in ascending order.

We present a greedy algorithm (Algorithm 4) for decomposing a query graph into its subgraphs and generating a SJ-Tree. Our choice for the greedy heuristic is motivated by extensive survey of the literature on optimal join order determination in relational databases [10, 14, 21]. A key conclusion of the survey states that *left-deep join plans* (or left deep binary trees in this case) is one

Algorithm 3 LAZY-SEARCH(G_d, T, edges)
```
1: leaf-nodes =GET-LEAF-NODES(T)
2: for all e_s ∈ edges do
3:     UPDATE-GRAPH(G_d, e_s)
4:     for all n ∈ leaf-nodes do
5:         u =src(e_s)
6:         v =dst(e_s)
7:         if DISABLED(u, n) AND DISABLED(v, n) then
8:             continue
9:         g_sub^q =GET-QUERY-SUBGRAPH(T, n)
10:        matches =SUBGRAPH-ISO(G_d, g_sub^q, e)
11:        for all m ∈ matches do
12:            if n = 0 then
13:                ENABLE-SEARCH-SIBLING(n, m)
14:            else
15:                M_j = QUERY-SIBLING-JOIN(n, m)
16:                p = PARENT(n)
17:                for all m_j ∈ M_j do
18:                    UPDATE(p, m_j)
19:                    ENABLE-SEARCH-SIBLING(p, m)
```

Algorithm 4 BUILD-SJ-TREE(G_q, M)
```
1: frontier = ∅
2: while |V(G_q)| > 0 do
3:     g_sub = ∅
4:     for all g_M ∈ M do
5:         if frontier ≠ ∅ then
6:             for all v ∈ frontier do
7:                 g_sub =SUBGRAPH-ISO(G_q, v, g_M)
8:                 break
9:         else
10:            g_sub =SUBGRAPH-ISO(G_q, g_M)
11:        if g_sub ≠ ∅ then
12:            frontier = frontier ∪ V(g_sub)
13:            G_q =REMOVE-SUBGRAPH(G_q, g_sub)
```

of the best performing heuristics. The above mentioned studies point to a large body of research using techniques such as dynamic programming and genetic algorithms to find the optimal join order. Nonetheless, finding the lowest cost join order or using a cost-driven join order determination remains an interesting problem in graph databases, and the approaches based on minimum spanning trees or approximate vertex cover can provide an initial path forward.

Inputs to Algorithm 4 are the query graph G_q and an ordered set of primitives M. Our goal is to decompose G_q into a collection of (possibly repeated) subgraphs chosen from M. Entries of M are sorted in ascending order of their subgraph selectivity. Given a query graph G_q, the algorithm begins with finding the subgraph with the lowest selectivity in M. This subgraph is next removed from the query graph and the nodes of the removed subgraph are pushed into a "frontier" set. We proceed by searching for the next selective subgraph that includes at least one node from the frontier set. We continue this process until the query graph is empty. SUBGRAPH-ISO performs a subgraph isomorphism operation to find an instance of g_M in G_q. Algorithm 4 uses two versions of SUBGRAPH-ISO. The first version uses three arguments, where the second argument is a vertex id v. This version of SUBGRAPH-ISO searches G_q for instances of g_M by only searching in the neighborhood of v. The other version accepting two arguments searches entire G_q for an instance of g_M. REMOVE-SUBGRAPH accepts two graphs as argument, where the second argument (g_{sub}) is a subgraph of the first graph (G_q). It removes all edges in G_q that belong to g_{sub}. A vertex is removed from G_q only when the edge removal results in a disconnected vertex.

5.1 Selectivity Estimation of Primitives

We propose computing the selectivity distribution of primitives by processing an initial set of edges from the graph stream. For experimentation purposes we assume that the selectivity order remains the same for the dynamic graph when we perform the query processing. This work does not focus on modeling the accuracy of this estimation. Modeling the impact on performance when the actual selectivity order deviates from the estimated selectivity order is an area of ongoing work.

Which subgraphs are good candidates as entries of M? Following are two desirable properties for entries in M: 1) the cost

for subgraph isomorphism should be low. 2) Selectivity estimation of these subgraphs should be efficient as we will need to periodically recompute the estimates from a graph stream. Based on these two criteria, we select single edge subgraphs and 2-edge paths as primitives in this study. Computing the selectivity distribution for single-edge subgraphs resolves to computing a histogram of various edge types. The selectivity distribution for 2-edge paths on a graph with V nodes, E vertices and k unique edge types can be done in $O(V(E + k^2))$ time. Algorithm 5 provides a simple algorithm to count all 2-edge paths. In our experiments, computing the path statistics for a network traffic dataset with 800K nodes and nearly 130 million edges takes about 50 seconds without any code optimization.

Algorithm 5 uses a Counter() data structure, which is a hashtable where given a key, the corresponding value indicates the number of times the key occurred in the data. A Counter() is updated via the UPDATE routine, which accepts the counter object, a key value and an integer to increment the corresponding key count. We iterate over all vertices in the input graph (G_d) (line 2). For an given vertex v, we count the number of occurrences of each unique edge type associated with it (accounting for edge directions). Line 8 iterates over all unique edge types associated with v. Next, given an edge type e_1 and its count n_1, we count the number of combinations possible with two edges of same type ($\binom{n}{2}$). Next, we compute the number of 2-edge paths that can be generated with e_1 and any other edge type e_2. We impose the LEXICALLY-GREATER constraint to ensure each edge is factored in only once in the 2-edge path distribution.

Note that we use a $Map()$ function instead of simply using the type associated with every edge. Most of our target applications have significant amount edge attributes in the graphs. As an example, in a network traffic graph we use the protocol information to determine the edge property. Thus, each network flow with the same protocol (e.g. HTTP, ICMP etc.) are mapped to the same edge type. Each flow is accompanied by multiple attributes such as source and destination ports, duration of communication etc.. Therefore, we can provide a hash function to map any user defined edge properties to an integer value. Thus, for queries with constraints on vertex and edge properties, a generic map function factors in both structural and semantic characteristics of the graph stream.

Counting the frequency for larger subgraphs is important. Given a query graph with M edges, ideally we would like to know the frequency of all subgraphs with size $1, 2, .., M − 1$. Collecting the frequency of larger subgraphs, specifically triangles have received a significant attention in the database and data mining community

[19]. Exhaustive enumeration of all the triangles can be expensive, specially in the presence of high degree vertices in the data. Approximate triangle counting via sampling for streaming and semi-streaming has been extensively studied in the recent years [11]. We foresee incorporation of such algorithms to support better query optimization capabilities for queries with triangles.

Algorithm 5 COUNT-2-EDGE-PATHS(G_d)

1: $P = Counter()$
2: **for all** $v \in V(G_d)$ **do**
3:     $C_v = Counter()$
4:     **for all** $e \in Neighbors(G_d, v)$ **do**
5:         $e_t = Map(e)$
6:         $Update(C_v, e_t, 1)$
7:     $E_t = Keys(C_v)$
8:     **for all** $e_1 \in E_t$ **do**
9:         $n_1 = Count(C_v, e_1)$
10:        $key = (e_1, e_1)$
11:        $Update(P, key, n_1(n_1 - 1)/2)$
12:        **for all** $e_2 \in$ LEXICALLY-GREATER(E_t, e_1) **do**
13:           $n_2 = Count(C_v, e_2)$
14:           $key = (e_1, e_2)$
15:           $Update(P, key, n_1 n_2)$

5.2 Query Decomposition Strategies

Algorithm 4 shows that we can generate multiple SJ-Trees for the same G_q by selecting different primitive sets for M. We can initiate M with only 1-edge subgraphs, only 2-edge subgraphs or a mix of both. As an example, for a 4-edge query graph, the removal of the first 2-edge subgraph can leave us with 2 isolated edges in G_q. At that stage, we will create two leaf nodes in the SJ-Tree with 1-edge subgraphs. For brevity we refer to both the second and third choice as 2-edge decomposition in the remaining discussions. Clearly, these 1 or 2-edge based decomposition strategies has different performance implications. Searching for 1-edge subgraphs is extremely fast. However, we stand to pay the price with memory usage if these 1-edge subgraphs are highly frequent. On the contrary, we expect 2-edge subgraphs to be more discriminative. Thus, we will trade off lowering the memory usage by spending more time searching for larger, discriminative subgraphs on every incoming edge.

DEFINITION **Expected Selectivity** We introduce a metric called *Expected Selectivity*, denoted as $S(\hat{T}_k)$. Given a SJ-Tree T_k, the Expected Selectivity is defined as the product of the selectivities of the leaf-level query subgraphs.

$leaves(T_k)$ returns the set of leaves in a SJ-Tree T_k. Given a node n, $V_{SG}(T, n)$ returns the subgraph corresponding to node n in SJ-Tree T. Finally, $S(g)$ is the selectivity of the subgraph g as defined earlier.

$$S(\hat{T}_k) = \prod_{n \in leaves(T_k)} S(V_{SG}(T_k, n)) \qquad (1)$$

DEFINITION **Relative Selectivity** We introduce a metric called *Relative Selectivity*, denoted as $\xi(T_k, T_1)$. Given a 1-edge decomposition T_1 and another decomposition T_k, we define $\xi(T_k, T_1)$ as follows.

$$\xi(T_k, T_1) = \frac{S(\hat{T}_k)}{S(\hat{T}_1)} \qquad (2)$$

We conclude the section with discussion on two desirable properties of a greedy SJ-Tree generation strategy.

THEOREM 1 Given the data graph G_d at any time t, assume that the query graph G_q is not guaranteed to be present in G_d. Then initiating the search for G_q by searching for g_{rare} where $g_{rare} \subset G_q$ and $\forall g \subset G_q || E(g)| = |E(g_{rare})|, frequency(g) > frequency(g_{rare})$ is in optimal strategy.

PROOF The time complexity for searching for a $O(1)$ for a 1-edge subgraph and $O(\bar{d}_v)$ for a 2-edge subgraph. Therefore, the runtime cost to search for g_{rare} is same as any other subgraph of G_q with the same number of edges. However, searching for g_{rare} will require minimum space because it has the minimum frequency amidst all subgraphs with same size. Therefore, searching for g_{rare} is an optimal strategy.
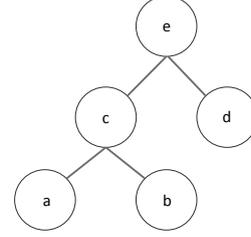
Figure 4: Example SJ-Tree used in proof of theorem 2.

THEOREM 2 Given a set of identical size subgraphs $\{g_k\}$ such that $\cup_k^n g_k = G_q$, a SJ-Tree with ordered leaves $g_k \prec g_{k+1} \prec g_{k+2}$ requires minimal space when $frequency(g_k \bowtie g_{k+1}) < frequency(g_{k+2})$.

PROOF By induction. Assume a SJ-Tree with three leaves as shown in Figure 4. Following the definitions of SJ-Tree, this is a left-deep binary tree with 3 leaves. Therefore, $frequency(c)$ denoted in shorthand as $f(c)$ $f(c) = min(f(a), f(b))$. Substituting for the frequency of c, space requirement for this tree $S(T) = f(a) + f(b) + f(d) + min(f(a), f(b))$. Thus, the space requirement for this tree is minimum if $f(a) < f(b) < f(c)$.

Now we can consider any arbitrary tree where T_n refers to a tree with a left subtree T_{n_1} and a right child l_{n+2}. Above shows that T_1 constructed as above will have minimum space requirement, and so will T_2 if $f(a) < f(b) < f(c) < f(d)$.

OBSERVATION 3 Given g_k, a subgraph of query graph G_q, it is efficient to decompose g_k if there is a subgraph $g \subset g_k$, such that $frequency(g) > \left(\frac{frequency(g_k)}{\bar{d}|V(g_k)|} \right)$, where $\bar{d}$ is the average vertex degree of the data graph and $|V(g_k)|$ is the number of vertices in g_k.

PROOF Given a graph g, the average cost for searching for another graph that is larger by a single edge is $\bar{d}$ multiplied by the number of vertices in g_k, and the proof follows.

Space Complexity The space complexity of the SJ-Tree can be measured in terms of the storage required by each leaf in the tree. The storage for any node in the tree is approximated by the product of the corresponding subgraph size (measured as the number of edges) and its frequency. Therefore, the space complexity of the SJ-Tree is $S(T) = \sum_k |E(g_k)| frequency(g_k)$. Given two subgraphs g_{small} and g_{big}, where g_{big} contains g_{small}, the frequency of g_{small} serves as an upper bound for g_{big}, assuming no overlapping edges. Therefore, we can assign each node in the tree to a group, where one node in each group serves to approximate the frequency of rest of the nodes in the group. Suppose $g_r(i)$ is the cardinality of the i-th group. Trivially, $\sum_i g_r(i) = N_T$, where N_T is the number of nodes in the SJ-Tree.

Therefore, given a query graph G_q and a SJ-Tree T expressing one possible query decomposition, we can estimate its space

163

complexity as $S(T) = \sum_i g_r(i)|E(g_i)|frequency(g_i)$. There is clearly a tradeoff between the accuracy of this estimate and the computation required to obtain the necessary measurements. Approximating the space complexity in terms of single edge subgraphs is computationally easiest, although it would be a very loose bound when the frequency of a single edge subgraph is orders of magnitude higher than larger subgraphs containing that single edge subgraph. Realistically, we foresee the groups being composed of unique 1-edge, 2-edge subgraphs and triangles (if it exists in the SJ-Tree) and approximate all larger subgraph in the SJ-Tree assigned to these groups.

5.3 Comparison with selectivity agnostic approaches

Our pattern decomposition approach based on relative selectivity provides an optimal way to look for discriminate patterns compared to existing approaches. For e.g, consider the generic path query graph in 5(a). A DAG based decomposition approach [7] may look either for complete path query or decompose it randomly as shown in 5(b). As the source vertex(s1) in such a pattern may be lot more frequent than sink $v4$, our selectivity based approach will clearly identify the s2->s3->s4 pattern as being more selective and start processing search from there, clearly this is more optimal than searching for every pattern starting at s1->s2.
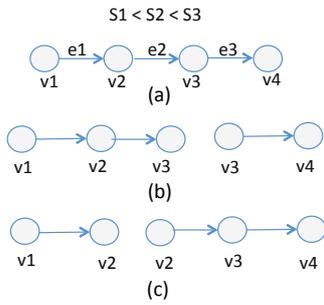
Figure 5: (a) Example path query. S_i **indicates the selectivity of edge** e_i**. (b) A selectivity agnostic decomposition. (c) Decomposition using our selectivity based approach.**

6. EXPERIMENTAL STUDIES

We perform experimental analysis on two real-world datasets (New York Times [1] (Internet Backbone Traffic data [1]) and a synthetic streaming RDF benchmark. In interest of space, we include result for CAIDA dataseti and RDF benchark only, NYTimes performance being similar to CAIDA. The experiments are performed to answer questions in the following categories.

1. STUDYING SELECTIVITY DISTRIBUTION What does the selectivity distribution of 2-edge subgraphs look like in real world datasets? What is the duration of time for which the selectivity distribution or selectivity order of 2-edge subgraphs remains static?

2. COMPARISON BETWEEN SEARCH STRATEGIES In the previous sections, we introduced two different choices for query decomposition (1-edge vs 2-edge path based) and two different choices for query execution (lazy vs non-lazy). How do the strategies compare?

3. AUTOMATED STRATEGY SELECTION Given a dynamic graph and a query graph, can we choose an effective strategy using their statistics?

COMPARISON WITH OTHER APPROACHES Although other continuous subgraph query systems exist ([7, 15], their objectives are different. Both focus on distributed system implementations, and explore aggregate queries or approximate queries. Also, their support for the type of graph is different from ours. Our test datasets drawn from cyber security and social networks involve directed graphs with labeled vertices and edges. We believe that the research contributions complement each other; hence, we compare our implementation with a non-incremental approach that performs subgraph isomorphism for the query graph (using VF2) on every new edge in the dynamic graph. .

6.1 Experimental setup

The experiments were performed on a 32-core Linux system with 2.1 GHz AMD Opteron processors, and with 64 GB memory. The code was compiled with g++ 4.7.2 compiler with -O3 optimization.

Given a pair of data graph and query graph, we perform either of two tasks: 1) query decomposition and 2) query processing.

Query decomposition: Query decomposition involves loading the data graph, collecting 1-edge and 2-edge subgraph statistics and performing query decomposition using the selectivity distribution of the subgraphs. The SJ-Tree generated by the query decomposition algorithm is stored as an ASCII file on disk.

Query processing: The query processing step begins with loading the query graph in memory, followed by initialization of the SJ-Tree structure from the corresponding file generated in the query decomposition step. We initialize the data graph in memory with zero edges. Next, edges parsed from the raw data file are streamed into the data graph. The continuous query algorithm is invoked after each AddEdge() call to the data graph.

6.2 Data source description

Summaries of various datasets used in the experiments are provided in Table 1. We tested each dataset with a set of randomly generated queries. The following describes the individual datasets and test query generation.

Network Traffic The dataset is an internet backbone traffic dataset obtained from www.caida.org. CAIDA (Cooperative Association for Internet Data Analysis) is a collaborative program that provides a wide collection of network traffic data. We used the "CAIDA Internet Anonymized Traces 2013 Dataset" for experimentation. The dataset contains 22 million network traffic flow (subsequently referred to as *netflow*) records collected over a *one minute period*. We excluded the traffic to/from IP addresses matching patterns 10.x.x.x or 192.168.x.x. These address spaces refer to private subnets and a communication from a given IP address from these spaces can actually refer to multiple physical hosts in the real word. As an example, every internet service provider configures the routers or machines inside a home network with IPs selected from the private IP address range. Therefore, if we see a request from 192.168.1.1 to google.com, there is no way to determine the exact origin of this communication. From a graph perspective, allowing private IP address and the subsequent aggregation of communication will result in the creation of vertices with giant neighbor lists, which will surely impact the search performance. A detailed list of use cases describing subgraph queries for cyber traffic monitoring are described in [12].

Social Media Stream Our final test dataset is a synthetic RDF social media stream available from the Linked Stream Benchmark
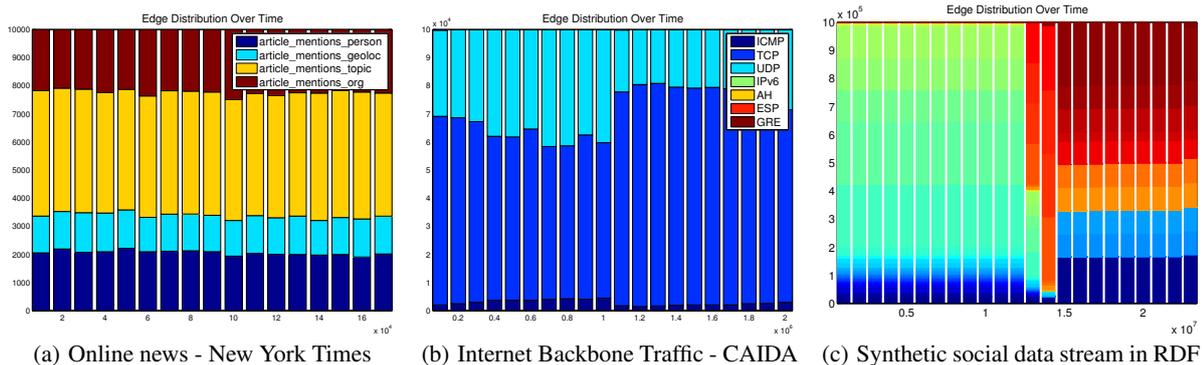
(a) Online news - New York Times   (b) Internet Backbone Traffic - CAIDA   (c) Synthetic social data stream in RDF

Figure 6: Edge type distribution shown with the evolution of the dynamic graph.

Table 1: Summary of test datasets

| Dataset	Type	Vertices	Edges
Internet Backbone Traffic	Network traffic	2,491,915	19,550,863
LSBench/CSPARQL Benchmark	RDF Stream	5,210,099	23,320,426
New York Times	Online News	64,639	157,019

(LSBench) [1]. We generated the dataset using the sibgenerator utility with 1 million users specified as the input parameter. The generated graph has a static and a streaming component. The static component refers to the social network with user profiles and social network relationships. The streaming component includes 3 streams. The *GPS stream* includes user checkins at various locations. The *Post and Comments stream* includes posts and comments by the users, subscriptions by users to forums, and a stream of "likes" and "tags". Finally, the *photo stream* includes information about photos uploaded by users, and "tags" and "likes" as applied to photos.

6.3 Selectivity Distribution

Figure 6 shows the edge distribution plotted over time. X-axis shows the number of cumulative edges in the graph as it is growing. The plotted distribution is not cumulative. The edge distribution is collected after fixed intervals. The interval is 10 thousand, 100 thousand and 1 million respectively. There are 4, 7, and 45 edge types in these datasets. The first half of the RDF dataset contains data for a simulated social network. The second half contains simulated data about the activities in the network such as posts, and checkins at locations The shift in the edge distribution around the mid point reflects these different characteristics. The key observation is that the relative order of different types of edges stays similar even as the graph evolves.

There were 14, 62 and 676 unique 2-edge paths present in the New York Times, netflow and LSBench datasets. Figure 7 shows the 2-edge path distribution for the LSBench dataset. We found a small number of 2-edge subgraphs to dominate the distribution across all the datasets. Other datasets show a similarly skewed distribution, and was omitted for space. The skew is heaviest for the LSBench dataset, which is expected given the higher number of unique edge types and the larger size of the dataset.

The goal of this analysis was to observe the variability in the selectivity distribution over time. The selectivity distribution is expected to vary over time. However, it is the relative order of the unique single edge or 2-edge subgraphs that matters from the query decomposition perspective. For each of the test datasets, we took
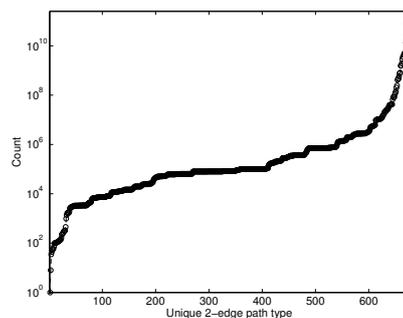
(a) Synthetic social data stream in RDF

Figure 7: 2-edge path distribution in each test data set. Each point on X-axis represents a unique 2-edge path and Y-axis shows its corresponding count.

multiple snapshots of the selectivity order and found it to be stable, except with fluctuations for the very low frequency components (data points on the left end of the distributions in Fig. 7). Significant changes in the selectivity order can adversely impact the performance of the query. Estimating the duration over which the selectivity ordering stays stable for a given data stream, quantification of errors based on shift in the distribution, and adapting the query algorithm to handle such shifts is reserved for future work.

## 6.4 Query Performance Analysis

This section presents query performance results obtained through query sweeps on the network traffic and social network dataset. We restrict the analysis to these two datasets for their larger size. The analysis on New York Times dataset made available in the Appendix section in the interest of space. For each query, we collect performance from 4 different query execution strategies obtained by 1-edge or 2-edge decomposition of a query graph and the lazy vs. track everything approach adapted by the query algorithm. The following tags are used to describe the plots in the remainder of the

165

paper: a) "*Single*": 1-edge decomposition, search tracks all matching subgraphs in SJ-tree, b) "*SingleLazy*": 1-edge based query decomposition, use "Lazy" approach to search, c) "*Path*": 2-edge decomposition, search tracks all matching subgraphs in SJ-Tree, and d) "*PathLazy*": 2-edge decomposition with "Lazy" search.

6.4.1 Network Traffic and LSBench

We present aggregated results for each query group for LSBench and CAIDA. Both of these datasets are orders of magnitude larger than New York Times and the scale allows us to magnify the differences between multiple strategies.

QUERY GENERATION We generate both path queries and binary tree queries for the netflow data. Figure 8 shows two decompositions of an example query. The vertex labels are fixed to type "ip" and the edge types are randomly chosen from a set of 7 protocols: ICMP, TCP, UDP, IPv6, AH, ESP and GRE. The binary tree queries were generated following the test generation methodology described in [16]. The LSBench dataset is tested with path queries and n-ary trees. A list of valid triples (vertex type, edge type , vertex type) is generated using the LSBench schema. A tree query is generated by randomly selecting an edge from the set of valid triples and then iteratively adding valid new edges from any of the nodes available. All our query graphs are unlabeled. Using netflow data as an example, we do not generate a query that has a label associated with any of the nodes. In practice, we expect users to employ labeled queries such as finding a tree pattern in the network traffic where the root of the tree has a IP address (i.e. label) from a certain subnet. For social data, we may look for paths with specified user ids (node labels) on the source and the destination nodes on the path. Here, our experiments are motivated to study the impact of subgraph distributional statistics on query processing.

(a)

(b)
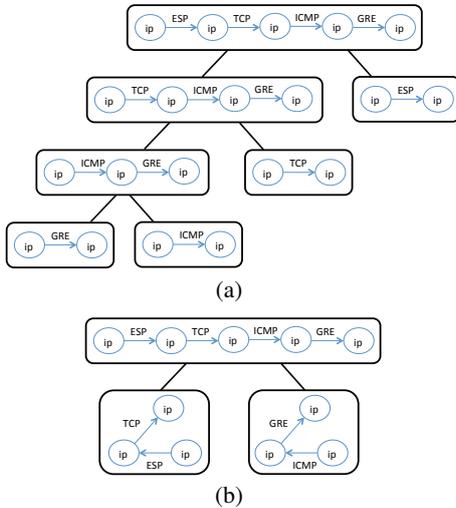
Figure 8: 1 and 2-edge based decompositions of a path query on netflow traffic data.

COMPARISON WITH OTHERS In our previous work [3] we had compared the performance of our implementation with the IncIsoMatch algorithm proposed by Fan et al. [6]. Our IncIsoMatch implementation was based on a variant of the well-known VF2 algorithm [5].

SUMMARIZATION OF RESULTS All queries of the same type (path or tree) and size (3-hop length or 5 nodes) are denoted as a group. We generated 100 queries for each group and then eliminated ones that contained 2-edge paths not seen in the sampled

path distribution. This was done for two reasons; first, inclusion of an unseen 2-edge path combination makes the query artificially discriminative. Our goal is to observe query processing time as a function of varying selectivity, so including unusually discriminative queries bias our studies. Second, when asked to generate a path-based decomposition, our SJ-Tree generator resorts to generating a single-edge based decomposition when a query subgraph contains an unseen 2-edge path. This would bias our comparison between a path-based decomposition and single-edge based decomposition. Finally, for all the "valid" queries we further sampled them by the Expected Selectivity computed using 2-edge path distribution and reduced each group to a smaller set of queries that provide a near uniform sampling of the Expected Selectivity from the larger set. Finally, the reported runtime for a given strategy (e.g. "PathLazy") is obtained by averaging the runtimes from the reduced set of queries,

Figure 9a-d shows the query processing times collected for both datasets. The size of the query processing window was fixed at 8M triples, and the performance statistics were collected at at the middle and at the end of the graph stream. We profiled different components of the query processing such as the time spent in performing subgraph isomorphism and the time spent in updating the SJ-Tree. The latter is largely composed of the time spent in looking up the hash tables in various nodes of the SJ-Tree, performing joins between partial matches and inserting new entries. We found that the subgraph isomorphism operation (for 1 or 2-edge subgraphs) dominates the processing time. Considering both classes of queries with diameter 4 and 5, the subgraph isomorphism operation consumes more than 95% of the total query processing time.

A general observation is that the performance of non-incremental search by VF2 is found to be 10-100x slower. The Y-axis is plotted in log scale, and we can see how the run times of the "Path" and "Single" approaches rise exponentially as the query sizes are increased. Overall, we find the "SingleLazy" and "PathLazy" are the best performing search approaches. As the tree queries show, the growth rate in the query processing time is much slower for the "Lazy" variants. This conclusively demonstrates the effectiveness of restricting the search to where a match is emerging, and growing the match by starting from the most selective sub-query.

6.5 Analysis via Relative Selectivity

Figure 10 shows the distribution of relative selectivity for queries with 4 edges across all three datasets. We picked query graphs with 4 edges to find a common basis for comparing different type of queries (k-partite vs. path queries) across multiple datasets, and the discussion is equally applicable to larger or different query class combinations. The top subplot shows the relative selectivity of 10 k-partite queries from the New York Times data. For netflow and LSBench, we randomly sampled 25 queries from the randomly generated path query collection. As can be seen, the relative selectivity is very low for the netflow dataset. Following the definition of relative selectivity, its value is lowered when the path distribution based selectivity is low. In other words, there are some paths in the query which have very low probability of occurrence. Therefore, the "PathLazy" approach is superior for such queries. Empirical observation on larger path queries and other tree queries seem to suggest two prominent clusters of relative selectivity values. The first one typically ranges from 0.001 and above, and the second one contains values that are smaller by multiple orders of magnitude. This suggests a heuristic that "PathLazy" strategy could be employed for queries with relative selectivity below 0.001, and "SingleLazy" be employed for queries above 0.001.
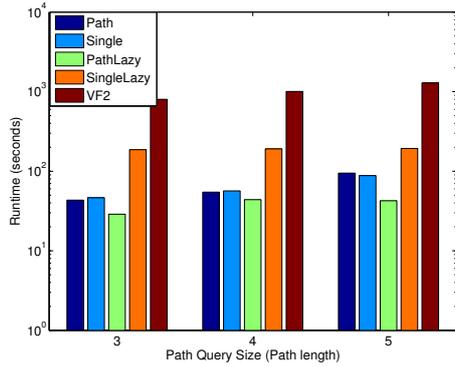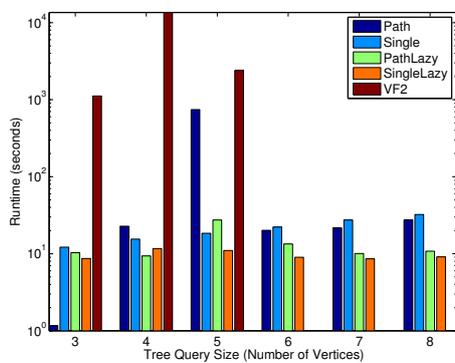
(a) Runtime for Path Queries on Netflow data.
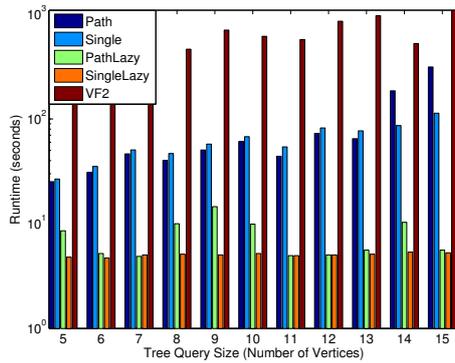
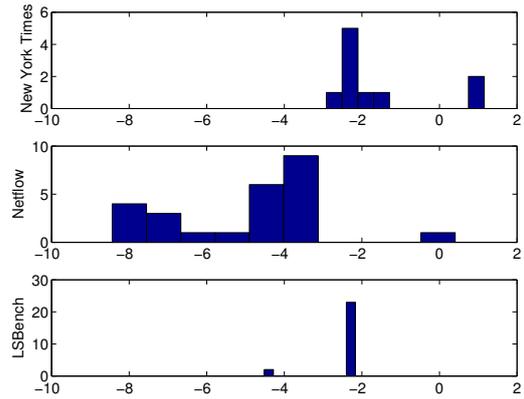(b) Runtime for Tree Queries on Netflow data.
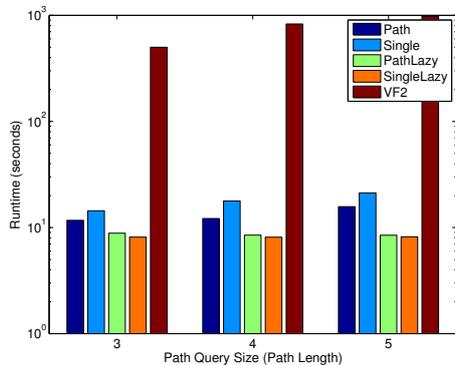
(c) Runtime for Path Queries on LSBench data.

(d) Runtime for Tree Queries on LSBench data.

Figure 9: Runtimes from Path and Tree Queries on Netflow and LSBench.

Figure 10: Distribution of Relative Selectivity across queries with 4 edges in 3 datasets. Relative selectivity is shown on X-axis in log scale.

7. CONCLUSION AND FUTURE WORK

We present a new subgraph isomorphism algorithm for dynamic graph search. We analyzed multiple real-world datasets and discovered that the distribution of 2-edge subgraphs are heavily skewed. We further demonstrated with a "Lazy" search algorithm that a query decomposition strategy exploiting this skew will be consistently efficient. Finally, we concluded with a Relative Selectivity based rule for selecting a search strategy.

The problem of continuous pattern detection is an emerging area, and there is an open field to explore. While our 2-edge subgraph based approach provides an initial foundation, deeper investigations are warranted for more accurate selectivity estimation. Subsequent research can leverage on the significant body of work on counting larger subgraphs such as triangles in streaming or semi-streaming scenarios to obtain quantitative estimates of space complexity of a given query decomposition. Adaptive query processing is an important follow-up problem as well. A long standing database query needs to be robust against shift in the data characteristics. While we propose a fast algorithm for periodic recomputation of the primitive distribution, we do not address the issues of modeling the inefficiency from operating under a different selectivity order and migrating existing partial matches from one SJ-Tree to another.

8. REFERENCES

[1] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1), 2010.

[2] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *IEEE Trans. on Knowl. and Data Eng.*, 22(8):1093–1109, Aug. 2010.

[3] S. Choudhury, L. Holder, G. Chin, and J. Feo. Fast search for multi-relational graphs. *ACM SIGMOD Workshop on Dynamic Network Management and Mining*, 2013.

[4] 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.

[5] 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.

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

[7] J. Gao, C. Zhou, J. Zhou, and J. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, 2014.

[8] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. SIGMOD '13.

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

[10] J. M. Hellerstein and M. Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.

[11] M. Jha, C. Seshadhri, and A. Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *SIGKDD*. ACM, 2013.

[12] C. Joslyn, S. Choudhury, D. Haglin, B. Howe, B. Nickless, and B. Olsen. Massive scale cyber traffic analysis: a driver for graph database research. In *1st ACM SIGMOD Workshop on Graph Data Management Experiences and Systems*, 2013.

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

[14] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, volume 86, pages 128–137. Citeseer, 1986.

[15] J. Mondal and A. Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD 2014*.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[17] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *ICDE '08*.

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

[19] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *SIGKDD*, 2009.

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

[21] Y. Wu, J. M. Patel, and H. Jagadish. Structural join order selection for xml query optimization. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 443–454. IEEE, 2003.

[22] P. Zhao, C. C. Aggarwal, and M. Wang. gsketch: on query estimation in graph streams. *PVLDB*, 5(3), 2011.

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

APPENDIX

A. ANALYSIS OF DYNAMIC GRAPH SEARCH ALGORITHM

At this point, it is probably obvious that different SJ-Tree structures can be generated from the same query graph (Figure 8). While multiple factors can lead to generation of different SJ-Trees, one primary factor is our choice for granularity of decomposition, the size and the structure of the subgraphs we decompose the query to.

Henceforth, we often refer to these set of small subgraphs as *search primitives* or simply *primitives*. As a first step to understand the speed-memory tradeoff associated with different choices for primitives, we begin with the complexity analysis of the dynamic graph search described in Algorithm 1 and 2. A key operation in Algorithm 1 is the process of subgraph isomorphism around every new edge in the graph. Therefore, we exclusively focus on the complexity analysis in terms of 1-3 edge subgraphs as candidates for search primitives.

SINGLE EDGE SUBGRAPHS When the query graph (g_{sub}^q in Algorithm 1, line 5) contains a single edge, checking if an edge from the data graph (e_s) matches the query edge require comparing the types and potentially other attributes of the edges. Depending on the query constraint, we may need to look up the node label to perform a string comparison or evaluate a regular expression. The node labels or any other node-specific properties are stored in an array leading to constant time access to node labels. Therefore, a single-edge query can be matched in $O(1)$ time.

TRIADS Assume that the query graph is a triad with three vertices v_1, v_2 and v_3, and edges ordered as $e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_1)$. For any edge e in the data graph, we can detect a match with e_1 in constant time. If e is matched, we search the neighborhood of the vertex that matches with v_2 to search for e_2. Denoting this vertex as v_2', the cost of this second level of search is $O(degree(v_2'))$. In case of a 3-edge subgraph, each of the successful second level searches proceed to find a match for the third edge. Thus, the cost of a 2-edge subgraph is $O(degree(v_2'))$ and a 3-edge subgraph is $O(degree(v_2') * degree(v_3'))$. We can refine these estimates to obtain an average cost of the search as $O(\bar{d}_2)$ for a 2-edge subgraph and $O(\bar{d}_2 \bar{d}_3)$ for a 3-edge subgraph, where $\bar{d}_2$ and $\bar{d}_3$ are the average degree of the vertices in the graph for the types of v_2 and v_3.

The next step is to estimate a cost for the SJ-Tree update operation (Algorithm 2). We begin with the hash-join operation (Algorithm 2, line 7). Assume the frequency of a graph g_q^i is n_i, where the frequency of a subgraph is defined as the count of its instances over an edge stream of length N. Therefore, over N edges, we can expect $O(n_1)$ matches for g_q^1 and $O(n_2)$ matches for g_q^2. Therefore, H_2 (hash table associated with the SJ-Tree node representing g_q^2) will be probed for a match $O(n_1)$ times over N edges and H_1 (associated with the SJ-Tree node representing g_q^1) will be probed $O(n_2)$ times within the same period.

If we knew the frequency of G_q, henceforth referred as $f_S(G_q)$, then we can also estimate the number of new subgraphs that will be produced as the result of the hash-joins. Given that the frequency of the larger subgraph can not exceed that of the more selective component we can approximate $O(n(G^q)) \simeq min(O(n_1), O(n_2))$. Therefore, the average work for every incoming edge in the graph can be expressed as,

$$\left(f_S(g_q^1) + f_S(g_q^2) + O(n_1) + O(n_2) + min(O(n_1), O(n_2)) \right) / N.$$

The Hash-Join combined with leaf level searches provides the simplest example of a SJ-Tree, a binary tree with height 1. In this section, we analyze the time complexity of the query processing as it happens in a multi-level SJ-Tree. Given any non-leaf node n, we can obtain the expression for average work by adapting the complexity expression shown above. Note that if a child of n, denoted by n_c, is not a leaf level node but an internal node, then the term corresponding to the search cost ($f_S(g)$) disappears. Additionally, we can replace the search cost with the cost corresponding to the average work incurred by the subtree rooted by n_c. Therefore, given a SJ-Tree (T_{sj}) the average work ($C(T_{sj})$) can be obtained by recursive computation from the root. $C(T_{sj}) = C(root(T_{SJ}))$