Mnemonic: A Parallel Subgraph Matching System for Streaming Graphs

Bibek Bhattarai, Howie Huang
George Washington University
{bhattarai_b, howie}@gwu.edu

Abstract—Finding patterns in large highly connected datasets is critical for value discovery in business development and scientific research. This work focuses on the problem of subgraph matching on streaming graphs, which provides utility in a myriad of real-world applications ranging from social network analysis to cybersecurity. Each application poses a different set of control parameters, including the restrictions for a match, type of data stream, and search granularity. The problem-driven design of existing subgraph matching systems makes them challenging to apply for different problem domains. This paper presents Mnemonic, a programmable system that provides a high-level API and democratizes the development of a wide variety of subgraph matching solutions. Importantly, Mnemonic also delivers key data management capabilities and optimizations to support real-time processing on long-running, high-velocity multi-relational graph streams. The experiments demonstrate the versatility of Mnemonic, as it outperforms several state-of-the-art systems by up to two orders of magnitude.

Index Terms—subgraph, graph pattern, matching, isomorphism, streaming

I. INTRODUCTION

Subgraph matching is the task of finding matches of a specific pattern in a data graph. Those patterns are referred to as query graphs and each of their matches in the data graph are known as embeddings. Subgraph matching has several applications such as social media analysis [1], fraud detection [2], and cyber-attack detection [3]. Due to constant data generation in these systems, promptly finding the newly formed embeddings is important in order to get new insights.

However, there are still some major challenges to build a subgraph matching system for streaming graphs. Most of the existing works follow filtering-enumeration technique [4], where per-query metadata, generally referred as an index, is used to store the candidate matches of the query graph and provide better access locality for backtracking processes [5]–[11]. Later, all the embeddings are listed and verified by traversing the index. However, these works are not suitable for streaming graph as updating index for single edge insertion or deletion can take up to $O(|V|)$, where $V$ is the number of nodes on the data graph. Recomputing the index every time the graph gets updated incurs huge redundant computation. Recent works [12], [13] present data-graph centric index structures to improve the updating and incremental subgraph matching. However, there is still a large gap to be filled in order to support a real-time subgraph matching (1) The multiple edges between two endpoints (e.g., NetFlow events at different time) are treated as one entry in their index and thus loses the temporal context of the system behavior. (2) While graphs are streamed, the index is maintained entirely in the memory. Since index size is much bigger than graph itself, it is not suited for long running streams with large data volume. (3) The edge insertions and deletions are processed in strictly sequential manners. While there is room for parallelization even for 1-edge insertion or deletion, it fails to sufficiently utilize the computing power of multi-core machines.

In addition, there is a lack of general framework for subgraph matching that can be tailored to the problem in hand. Most datasets, in addition to edge and node type contain multiple attributes per node/edge and the combination of these features are used to match a data graph edge to a query edge. Similarly, the definition of match itself varies depending upon the data and problem at hand. Some of the commonly used matching techniques include subgraph isomorphism [5], [9], [10], [14], homomorphism [12], [15]–[17], and simulation [18], [19]. Most of the existing works hard-code these specifications in their system. Without expertise in graph analysis, it is very difficult for users to choose appropriate subgraph matching solutions or make necessary changes.

We present Mnemonic, a programmable subgraph matching framework for streaming graphs. It is designed with following objectives: (1) It distinguishes the different instances of edges between the same endpoints, enabling a context-aware subgraph matching. (2) It supports incremental computation on long running, high-velocity event streams using batch processing and disk support whenever necessary, and (3) It exposes a high-level API set to democratize the implementation of subgraph matching solution that fits user requirements.

In order to accomplish these goals, Mnemonic designs the index storage, henceforth known as Data graph Edge centric, Binary Index (DEBI), which uses a bitmap to represent whether a given data graph edge matches the query graph edges. In order to support multi-graph, each edge in the graph is assigned a unique identifier, edgeId, and the index along with other attributes for each data graph edge is accessed using edgeId. The DEBI entry for a given data-query edge pair can be read, written, and updated in constant time($O(1)$). By reclaiming memory from deleted edges and their DEBI...
entries and periodic resets, memory consumption increases at a much smaller rate as the stream progresses. To the best of our knowledge, Mnemonic is the first subgraph matching system that achieves a non-monotonic index size. For events requiring a bigger search context, Mnemonic backups the data edges and their corresponding matches in the disks.

In order to batch the updates, Mnemonic uses a snapshot generator with a set of user-controlled parameters, e.g., stream type, window size, and stride. Each snapshot includes the last instance of the data graph and the changes made since then. Based on the insertions/deletions, the DEBI is updated and new embeddings are enumerated. For each inserted/deleted edge, this process has to explore graph top-down and bottom-up to find all the impacted matches. Since, there is huge amount of redundant graph traversals among the exploration for each edge, Mnemonic constructs a unified traversal frontier where each edge is traversed only once for a batch of insertion or deletion. Since the batch of edges are processed at once, redundant embeddings are generated, which are proactively eliminated using masking.

Finally, to ease the process of implementing subgraph matching solutions, it exposes a set of high-level APIs. For the majority of subgraph matching variants, a user needs to implement two application-specific functions - edgeMatcher() function controls the content of DEBI by defining the matching conditions between a data edge and a query edge based on the vertex and edge attributes, and enumerator() function accesses the DEBI using APIs to construct the embeddings that meet the structural constraints of the query graph. Optimization routines such as finding match order and starting nodes and parallelization are done by Mnemonic. Experienced users can achieve more flexibility by modifying these optimizations routines as well. To the best of our knowledge, Mnemonic is the first programmable subgraph matching framework.

We have implemented several subgraph matching variants using Mnemonic with different embedding definitions and data characteristics. A series of evaluations demonstrate that Mnemonic is able to outperform state-of-the-art systems on various problems, e.g., streaming graph isomorphism Turbo Flux [12] by 7.8x and BigJoin [16] by a orders magnitude in larger queries, re-computation from scratch using static graph solution CECI [9] by 42.1x, and time-constrained isomorphism [20] by 1.8x.

II. BACKGROUND

A. Preliminaries

A graph is formally defined as \( G = (V,E,L^v,L^e) \), where \( V \) is the set of vertices, \( E \) the set of edges, and \( L^v \) and \( L^e \) attribute functions that map a node and an edge to a set of attributes \( \Xi^v \) and \( \Xi^e \) respectively. Most commonly used edge attributes in subgraph matching are labels, which represent the types of nodes or relationships. Figure 1(b), (c) and, (d) show the different snapshots of an example data graph, i.e., the initial graph \( G \) at time \( t \), and two updated snapshots \( G1 = G \oplus \Delta G1 \) and \( G2 = G1 \oplus \Delta G2 \) at time \( t1 \) and \( t2 \) respectively. We use directed graphs in the examples and experiments, but our techniques work on undirected graphs as long as both directions are recorded. To facilitate the easy update and sequential accesses, Mnemonic uses the adjacency list format to store the graphs, where each vertex has a list that stores all its outgoing and incoming edges. To make sure the edges in multigraph are differentiated, each edge is assigned an id. The vertex and edge attributes are stored in another data structure indexed by their id.

A subgraph of \( G \) is a graph \( G_S \) whose vertex set \( V_S \) is a subset of \( V \) (\( V_S \subseteq V \)), and whose edge set \( E_S \) is a subset of \( E \) (\( E_S \subseteq E \)). A query graph \( G_Q = (V_Q,E_Q,L^v_Q,L^e_Q) \) as shown in Figure 1(e) is usually a smaller graph which represents the pattern of interest. A root query node is normally the most selective node in the query graph, which is the starting point of the matching process, that is, \( u_0 \) in Figure 1(e). A query tree represents a variant of the spanning tree of the query graph, e.g., a Breadth-First Search (BFS) tree shown in Figure 1(f). Here, all the tree edges \( (u_j,u_k) \) connect a parent node \( u_j \) to a node \( u_k \). Note that the parent child relationship is not dependent on the direction of edges, e.g., \( u_j \) is the parent of \( u_k \) even though the edge is directed from \( u_k \) to \( u_j \). The edges not in the query tree are referred to as non-tree edges, e.g., \( (u_2,u_0) \). A matching order is the sequence of query nodes and consequently, query tree edges to follow when finding an embedding. The non-tree edges are manually verified as we follow the matching order.

B. Problem Definition

Subgraph matching is the task of listing all the subgraphs of \( G \) that match the query graph \( G_Q \), i.e., a set \( S(G,G_Q) \). Real-time subgraph matching requires continuously computing all the subgraphs that can be found up to the given point in time \( t \). Specifically, given the data graph \( G \) and query graph \( G_Q \) at time \( t_0 \), if a batch of update \( \Delta G = \{e_1,e_2,e_3,...,e_n\} \) is made between \( t_0 \) and \( t_1(= t_0 + \delta t) \), the system needs to compute a result set \( \Delta S \) at \( t_1 \), such that \( S(G \oplus \Delta G,G_Q) = S(G,G_Q) \oplus \Delta S \). Here \( \Delta S \) represents all the embeddings newly formed (or removed) due to the batch of insertions (or deletions) represented by \( \Delta G \).
Whether or not a subgraph $G_S$ is a match to query $G_Q$ depends on the underlying matching rule. For example, isomorphism [5], [8], [9] requires the embeddings to have a bijective mapping with the query graph. Formally, a subgraph $G_S$ of a data graph $G$ is isomorphic to the query graph $G_Q$ if and only if there exists a bijective function $f : V_Q \rightarrow V_S$ such that $\forall u \in V_Q$. $L_Q(u) \subseteq L_S(f(u))$, $\forall (u_i, u_j) \in E_Q$, $(f(u_i), f(u_j)) \in E_S$ and $L_Q(u_i, u_j) = L_S(f(u_i), f(u_j))$. In Figure 1(b), the graph snapshot $G$ contains two isomorphic embedding of the query graph in Figure 1(a), where the query edges $\{(u_0, u_1), (u_2, u_0), (u_0, u_3), (u_1, u_3), (u_1, u_4), (u_2, u_6), (u_2, u_3), (v_1, v_3), (v_1, v_6), (v_3, v_7), (v_4, v_8), (v_4, v_5)\}$ respectively on the first embedding while the next embedding matches query edge $(u_2, u_0)$ to data edge $(v_4, v_0)$. Isomorphism is useful in applications that require strict matching such as in chemical compound or protein-network.

Some works utilize looser restrictions on matching such as homomorphism [12] and simulation [18], [19]. Other variants include the topology of the edges [20], automorphism removal [9], [21], and approximate matching [22]. All of these solutions first find matches of individual edges(nodes) and combine them while confirming to the desired level of structural similarity. Using the edgeMatcher() and enumerator() functions in API, a user can implement all these variants and many more without having to redesign the whole system stack as we demonstrate in our evaluation.

C. Related Works

The gather-scatter models used by graph computing systems [23] or even the coarse-grained filter-expand models used by graph mining-systems [24] are not effective for subgraph matching. Therefore, subgraph matching has been subject of active research for the last 2 decades. These works normally only support static graphs and focus on developing number of techniques to optimize the matching order [8], [25], prune the false candidates [25], [26], optimize index structure [5], [6], reduce the data redundancy [7], search for the multiple queries together [27], and parallelization [9], [28]. The way these systems could work for real-time subgraph matching is getting a graph snapshot at a point in time, and running subgraph matching on that snapshot. However, when the interval between two snapshots gets smaller, the redundancy plagues their performance.

TurboFlux [12] has made a breakthrough in incremental subgraph matching using a data-graph centric index structure. While this improves the index update costs, its inability to represent event context, strictly sequential processing, and in-memory index storage makes its utility limited. A recent work [20] designs a sliding window, time-constrained subgraph isomorphism, which handles the constant inflow and outflow of edges from the search space in batches. However, it stores partially materialized embeddings in the tree, which comes with a very high memory cost and embedding update costs. Mnemonic is closest to these works as it provides an incremental approach to filtering-enumeration subgraph matching in streaming graphs. In contrast to the problem-driven approach in these works, it designs a wide-view system that can be customized to different embedding definitions, while providing the system level capabilities to support subgraph matching on large data and query graphs over long run.

Join based approaches treat graph query as relational multi-way join, where the result set is generally expanded one query edge at a time [13], [29] or one query node at a time [16], [30] using different join plan [31]. While they usually are better suited for streaming graphs, provide superior join plan, and work well with parallelization frameworks, they struggle to work for larger query graphs because they perform aggressive expansion of partial matches using node label and/or edge label filters. In contrast, Mnemonic utilizes the topology of a query graph to perform much better filtering and pruning before expanding the partial embeddings.

III. MNEMONIC

Figure 2 illustrates the main components of the Mnemonic system. Given a data stream, a user provides four inputs (ovals in Figure 2) to the system - query graph, stream configurations that are used to customize the snapshot generation, a function edgeMatcher() that describes the matching conditions between a data and query edge pair, and enumerator() that specifies restrictions on the structural properties of an embedding. Mnemonic follows a filtering and enumeration paradigm, where edgeMatcher() and enumerator() helps to customize the two steps respectively.

Algorithm 1: Subgraph Matching using Mnemonic

1. input: queryGraph $G_Q$, dataStream $G$, config
2. output: set $S(G, G_Q) = \{\}$ of embedding

// System Initialization
3. stream_handle_t streamHandle = InitializeStream(dataStream, config);

4. index_handle_t indexHandle = InitializeIndex(queryGraph, config);

// Continuous Enumeration of Embeddings
5. snapshot_t s;
6. while (s = getSnapshot()) do
7. if s.insertBatch.size() > 0 then
8. batchInserts(s.insert_list, edgeMatcher(), enumerator());
9. if s.deleteBatch.size() > 0 then
10. batchDeletes(s.delete_list, edgeMatcher(), enumerator());

Under the hood, the system is facilitated by two major data structures and three major components. As specified, the graph is stored as an adjacency list, where each node has its own list of incoming and outgoing edges. The attribute store indexed

---

1 Due to lack of space, we refrain from their definition on this paper but we encourage interested readers to refer to the cited works.
by node and edge ids store the vertex and edge attributes. The index structure DEBI is stored alongside vertex attributes. The snapshot generator generates the new batch of inputs to be processed at a given time. The index management unit incrementally updates the DEBI for a given snapshot (Section V). The embedding enumeration unit prepares the current batch of updates for enumeration, computes matching order and ensures the correctness of result (Section VI).

Mnemonic implements a subgraph matching system as outlined in Algorithm 1. First, using data and query based heuristics, it sets up hyper-parameters. This step is normally referred to as preprocessing in existing works [4], which involves tasks such as finding the root query node, generating the query tree, and determining the matching order. The default heuristics for these tasks are chosen in Mnemonic based on common practices. For example, the most selective node is chosen as the root node, the BFS tree rooted at that node is the spanning tree, and BFS traversal order is the matching order. These tasks can be accomplished by invoking InitializeIndex. InitializeComponent sets up the snapshot generator, where a few customizing parameters such as window size, stride or batch size are specified by the user. In each iteration, the last snapshot of the graph and changes made since then are provided with two lists—insertion and deletion list. For example, at time \( t_2 \) in Figure 3(d), the insertion list \((1, 2, 0, 0)\), and deletion list \((3, 7, 1, 1, 5, 0)\) are generated on top of snapshot \( G_1 \). Once the hyper-parameters are determined, it fetches the new snapshot, makes changes to the snapshot \( G \) and configure index handle respectively. The function batchInserts() initializes the graph stream and configures index handle respectively. The function getNextSnapshot() provides access to the latest stable snapshot of the data graph and changes made since then. batchInserts() updates the graph and DEBI with insertions and finds newly formed embeddings, while batchDeletes() does the same for deletions. Function getCandidates() enables the enumerator() to fetch the candidates of a given query edge from DEBI. The function verifyNte() is used to verify the presence of non-tree connections for the current data-query node pair on partially materialized embedding.

Let us implement edgeMatcher() and enumerator() for subgraph isomorphism using Mnemonic APIs. Given a data-query edge pair, edgeMatcher() returns True if the edge labels and respective node labels of both endpoints are identical as shown in Figure 4 (line 3-5). The enumerator() for isomorphism (line 11-31) implements a backtracking algorithm that joins matches of individual query edges to build the embeddings. It takes an incomplete embedding \( e \) and edge iterator \( \text{itr} \) as input. For each edge in the query tree, it retrieves the candidate matches by calling getCandidates(). The corresponding entry of the embedding \( e \) is then set to each of the candidate matches one at a time, while the process is repeated recursively for the next query edge. Line 23 makes sure that each node has a distinct match in a given embedding (injective constraint) while line 19 verifies the non-tree edge connections for the pair of \( v \) and \( u \). When the match for all edges is found, the embedding is written to the result file using function saveEmbedding() and the backtracking process returns to the previous edge (line 13-16).

By customizing these functions, a user can implement different variants of subgraph matching solutions. For example, the enumerator() for homomorphism can be obtained by removing the injective constraint verification from isomorphism, i.e., line 23 from Figure 4. In homomorphism, a single data edge can be used as a match for multiple query edges. Dual simulation [18] can simply be obtained by joining candidates of each edge and verifying duality for non-tree edges. Similarly, strong simulation can be obtained by adding locality constraint on the enumerator() for dual simulation. We skipped the implementation detail of these algorithms due to space constraints, but our evaluation demonstrates Mnemonics’ ability to program different variants.

IV. INDEX DESIGN

For the filter and enumerate techniques, naive storage designs such as per-node candidate list or match matrix perform poorly during enumeration as they lose graph topology and have to repeatedly verify edges by searching on the data graph. Recent works [8, 9] adopt an approach where explicit
candidate lists are collected and stored along with the graph topology. Since it stores the matches of each query node, and their connections to the matches of the neighboring query nodes, search space becomes very compact. Figure 5(a) shows the index constructed by CECI [9] for the data graph in Figure 1(b). Each node in the query tree contains a key-value store, where the keys \( v_p \) are the matches of the parent query node \( u_p \) and the values are the list of nodes \( \{v\} \) that match \( u \) and are adjacent to \( v_p \). In addition to reducing search space, they obtain better access throughout since the matches of a node are stored consecutively.

Observation #1: Query-centric index structures are not suited for streaming graphs due to high update cost. In G2, when the edge \((v_3, v_7)\) is deleted, since, it matches the edge \((u_1, u_4)\), one first needs to find the key \( v_3 \) in the key-value store of \( u_4 \), and remove \( v_7 \) from entry \( ceci[u_4][v_3] \). Locating the key \( v_3 \) and removing \( v_7 \) both take up \( \mathcal{O}(|V|) \) time. To reduce the update cost, TurboFlux [12] proposes a data graph centric index structure known as \( \text{dcg} \). They design \( \text{dcg} \) as a complete multigraph where every vertex pair \((v_i, v_j)\) \( \in E \) has \(|V| - 1\) edges. Each edge has a query vertex \( u \) as the edge label, and its state is one of \text{Null}, \text{Implicit}, or \text{Explicit} based on local and neighborhood attributes. When an edge \((v_i, v_j)\) is inserted to the graph, it simply adds edges to the \( \text{dcg} \).

Observation #2: While Turboflux is easier to update and enables the incremental subgraph matching, it fails to preserve event context and even produces false positives. In \( \text{dcg} \), all instances of an edge between two endpoints are considered the same. While they have the option to re-enumerate all embedding involving an edge when it is inserted again, it fails to distinguish between 2 instances of the edges. However, such distinction is very important in many applications. For example, in cyber forensic, a login to the server by a user after compromise is very significant to attack impact analysis compared to one before compromise. Including the benign login in the embedding would produce a false positive. Uniquely identifying such events is necessary in order to generate useful embeddings.

A. \text{DEBI}

Each edge in the data graph has a unique identifier associated with it henceforth referred to as \text{edgeId}. Figure 5(b) shows the \text{DEBI} for graph snapshot \( G \), while Figure 5(c)- (d) shows the updated \text{DEBI} at time \( t1 \) and \( t2 \) respectively after processing two snapshots. \text{DEBI} maintains a bitmap of size \(|V_Q| - 1\) bit for each data graph edge, where each bit represents whether the data edge matches the corresponding query edge or not. The information about whether an edge \((v', v)\) with the id of \text{edgeId} matches \((u_p, u)\) is stored at \text{DEBI}[\text{edgeId}[u]]. In addition, when adding the nodes of the root query node \((u_0)\) are stored in a bit vector of size \(|V|\), known as \text{roots}. The size of \text{DEBI} is equal to \( (|E| \times (|V_Q| - 1) + |V|) \) bits, which is same as that of TurboFlux [12] and smaller than \( \mathcal{O}(|E_Q| \times |E|) \) of CECI [9], \( \mathcal{O}(|E| \times |V_Q|) \) of CFLMatch [8], and \( \mathcal{O}(|E| \times |V_Q|) \) of Turboiso [5].

**Updating**: The time for inserting, updating, or reading the entry for any data and query edge pair in \text{DEBI} is constant, i.e., \( \mathcal{O}(1) \). When an edge \((v', v)\) with the id of \text{edgeId} is inserted, the entry \text{DEBI}[\text{edgeId}] is initialized. For all \((u_p, u)\) in the query tree that matches \((v', v)\), \text{DEBI}[\text{edgeId}[u]] is set to True. If the inserted edge matches multiple query edges, all the entries that need to be changed are stored together in \text{DEBI}. Once the initial edges to change for are identified, the graph is traversed downward and upward to reflect the changes of update in the \text{DEBI} as explained in Section V.

For the graph in Figure 1(b) and index in Figure 5(c), when \((v_0, v_2)\) is inserted, \text{DEBI}[1][v_1] is updated to True. In addition, \text{DEBI}[10][u_3] and \text{DEBI}[4][u_4] are now changed to True. Since \((v_4, v_1)\) matches both \((u_2, v_0)\) and \((u_3, v_7)\), we need to set \text{DEBI}[2][u_2] and \text{DEBI}[2][u_3] to True. In contrast, CECI in Figure 5(a) needs to first find \( v_0 \) in CECI[\(v_1\)] and add \( v_2 \) to it. Then, one needs to repeat the process by searching \( v_2 \) at \( v_3 \) and \( u_4 \), and adding \( v_6 \) and \( v_7 \) to the respective lists, which costs \( \mathcal{O}(|V|) \) each.

**Accessing**: \text{DEBI} achieves an equivalency to CECI [9] in terms of accessing candidates. Given a query edge \((u_p, u)\), \text{getCandidates} function returns all edges \((u_p, v)\) with the id of \text{edgeId} such that \text{DEBI}[\text{edgeId}[u]] is True, where \((*, v_p)\) is already matched with \((*, u_p)\). Since all outgoing/incoming edges of \( v_p \) are stored together in the adjacency list, all the candidates can be fetched in one read. A small penalty is paid to filter out the edges that do not match \((u_p, u)\).

**Memory recycling**: In Mnemonic, the memory space used by deleted edges is recycled for upcoming edges. When an edge is deleted, Mnemonic first locates this edge in the adjacency list of the corresponding vertex and swaps it with the last entry of the adjacency list, and reduces the degree of the vertex. When a new edge for the same vertex is inserted later, the degree is increased and the \text{edgeId} of the previously deleted edge is reused. When no \text{edgeId} is available, a new entry is inserted. In Figure 1(d), when the edge \((v_0, v_9)\) is inserted after \( t2 \), it uses the id of the last deleted edge of \( v_1 \), i.e. \text{edgeId} = 3 of \((v_1, v_5)\). By reclaiming the memory from deleted edges and their corresponding \text{DEBI} entries, Mnemonic is able to keep index size in check.

**External memory support**: Mnemonic provides an option to backup edges and their attributes to the external-memory storage when search space is large. It adopts a simple FIFO retention policy, where the newly generated events are kept in the memory. The amount of edges to be kept in memory is specified by a user-controlled variable called in-memory window. All the edges \((v_p, v)\) with an id of \text{edgeId} outside of
this window are added to a buffer along with DEBI[edgeId]. When the buffer is full, it is dumped into the disks. The vertex information is maintained entirely in memory, while the edges on the disk are stored using transactional edge logs [32], so that the adjacency list of a given node can be fetched in a single transaction.

V. INDEX MANAGEMENT AND UPDATE

When an edge is inserted to or deleted from the data graph, DEBI needs to be updated. Given a data edge \((v_p, v)\) with an id of edgeId, and a query tree edge \((u_p, u)\), the entry \(\text{DEBI}[\text{edgeId}][u]\) is set True when the following two conditions are met: (1) (edge match) The data edge \((v_p, v)\) matches \((u_p, u)\) as defined by the edgeMatcher(), and (2) (neighborhood match) For each neighbor \(u'\) of \(u\), there is at least one edge \((v, v')\) that matches the query edge \((u, u')\). Similarly, for each neighbor \(u'_p\) of \(u_p\), the edge \((u_p, u'_p)\) has at least one edge \((v_p, v'_p)\) that is a match.

For each edge insertion or deletion, a DEBI entry needs to be initialized or cleared. In addition, each update in the data graph may change the to be initialized or cleared. In addition, each update in the data graph may change the adjacency list of a given node can be fetched in a single transaction.

For a query edge \((u_p, u)\), the frontier consists of all the edges from current batch that matches it, and all the edges adjacent to \(v_p\), where \((*, v_p)\) matches \((*, v_p)\) (F1). For each edge \((v_p, v)\) in the frontier with id edgeId, if \(\text{DEBI}[\text{edgeId}][u]\) is True or if there is another edge \((v'_p, v')\) (id of edgeId') such that \(\text{DEBI}[\text{edgeId'}][u]\) is True, the \(\text{DEBI}[\text{edgeId}][u]\) is set True and the edge is removed from frontier. If edge \((v_p, v)\) and \((u_p, u)\) do not meet conditions specified in F2 or F3 it is removed from frontier. Otherwise, \(\text{DEBI}[\text{edgeId}][u]\) is set True. In snapshot G1, since \((v_0, v_2)\) satisfies F2 and F3 (for \((u_0, u_1), \text{DEBI}[11][u_1]\) is set True. Later, since edges \((v_3, v_6)\) (and \((v_3, v_7)\) are already matches of \((u_1, u_3)\) and \((u_1, u_4)\), \(\text{DEBI}[10][u_3]\) and \(\text{DEBI}[4][u_4]\) are readily set True and are removed from the corresponding frontiers. Here, 10 and 4 are id of \((v_2, v_6)\) and \((v_2, v_7)\) respectively.

The bottom-up filtering (line 5) routine processes one query edge at a time in reverse BFS traversal order. Given an edge \((v_p, v)\) with the id of edgeId such that \(\text{DEBI}[\text{edgeId}][u]\) is True, the bottom-up filtering rule F4 makes sure that query sub-tree rooted at \(u\) matches a tree in data graph rooted at \(v\). In other word, for each child \(u_c\) of \(u\) in the query tree, there must be an edge \((v, v_c)\) with an id edgeId in data.
graph such that \( \text{DEBI}[\text{edgeId}][u] \) is True. This process is not required for edges with a leaf node as one endpoint. For an edge \((v_p, v)\) with id \text{edgeId} in the frontier of an edge \((u_p, u)\), if \( \text{DEBI}[\text{edgeId}][u] \) is True and \( u \) is not satisfied, \( \text{DEBI}[\text{edgeId}][u] \) is changed to False. Now, all the edges \(*, v_p\) with an id \text{edgeId} such that \( \text{DEBI}[\text{edgeId}][u] \) is True are added to the frontier of parent query edge \(*, u_p\). In snapshot G1, since \((v_2, v_6)\) and \((v_2, v_7)\) are matches of \((u_1, u_3)\) and \((u_1, u_4)\) respectively, \( \text{DEBI}[11][u_1] \) remains True. Similarly, for a node \( v \) in the frontier of \( u_0 \), if \( f_4 \) is not satisfied, roots\( [v] \) is changed to False. Since \( v_0 \) now has matching neighbors for each of \( u_1, u_2 \), and \( u_5 \), roots\( [v_0] \) remains True.

**Batching Deletions:** The process for the batch of deletion is somewhat reverse to the insertion. Starting from the deleted edges, the enumeration is performed on \( \text{DEBI} \) before updating. Later the batch of deleted edges are removed from the graph and the top-down filtering follows the bottom-up in order to update the \( \text{DEBI} \). The frontier for each query node \((u_p, u)\) initially contains the edges \((id = \text{edgeId})\) from the deletion batch such that \( \text{DEBI}[\text{edgeId}][u] \) is True. During bottom-up exploration, for each edge \((v_p, v)\) with id \text{edgeId} in the frontier of the query edge \((u_p, u)\), \( \text{DEBI}[\text{edgeId}][u] \) is set to False. If the pair \( v_p \) and \( u \) does not satisfy \( f_4 \), all edges \(*, v_p\) with an id \text{edgeId} such that \( \text{DEBI}[\text{edgeId}][u] \) is True are added to the frontier of \(*, u_p\). For the root query node, the roots is updated using the same rule. In the snapshot G2, when an edge \((v_3, v_7)\) \( (id = 6) \) is deleted, \( \text{DEBI}[11][u_1] \) is changed to False, i.e., \((v_1, v_3)\) no longer matches \((u_0, u_1)\). Later, when \((v_1, v_5)\) \((id = 3) \) is deleted, the pair \( v_1 \) and \( u_0 \) does not satisfy \( f_4 \) as \( v_0 \) does not have a neighbor matching \( u_5 \). Thus, roots\( [v_1] \) is set to False.

During top-down filtering, for all \(*, v_p\) in the frontier of \(*, u_p\), all edges \((v_p, v)\) with id \text{edgeId} are obtained, and \( \text{DEBI}[\text{edgeId}][u] \) is set False. If there is no other edge \((v_p, v)\) with id \text{edgeId} such that \( \text{DEBI}[\text{edgeId}][u] \) is True, \((v_p, v)\) is added to the frontier of \((u_p, u)\). In snapshot G2, since roots\( [v_1] \) is False, \{\((v_1, v_2), (v_1, v_3)\)\} are initially added to the frontier of \((u_0, u_1)\). But since \((v_2, v_6)\) still matches \((u_0, u_1)\), \((v_1, v_2)\) is removed from the frontier. However, \((v_1, v_3)\) is kept in the frontier as there is no other edge incident on \( v_3 \) that matches \((u_0, u_1)\). The effect propagates all the way down and the whole sub-tree rooted at \( v_1 \) is no longer a match to their counterpart in the query tree.

**VI. Embedding Enumeration**

Embedding enumeration is carried out with the help of a user-defined function \text{enumerator}(). Under the hood, \text{Mnemonic} needs to do a few tasks to improve the performance and ensure correctness of results.

**Work decomposition:** For a batch of updates, \text{Mnemonic} decomposes the incoming batch into fine-grained work units, i.e., data-query edge pair. Since the real-world graphs follow power-law nature, this finer-grained decomposition can obtain better load balance. For each data edge \((u_p, v)\) with an id \text{edgeId} inserted to or deleted from the graph, an initial embedding \{\((v_p, v)\)\} is formed if \( \text{DEBI}[\text{edgeId}][u] \) is True. If the inserted edge \((v_x, v_y)\) matches a non-tree edge \((u_x, u_y)\), then potentially multiple initial embeddings \{\((v', v_y), (v'', v_x)\)\} are generated such that these edges match the tree edges \((v', u_x)\) and \((u_y', u_y)\) respectively. Each of these initial embeddings represent one work unit and are dynamically distributed among workers using a pull based technique. In the snapshot G1, initial embeddings for enumeration are \{\((v_2, v_6)\rightarrow(u_1, u_3)\), \((v_0, v_2)\rightarrow(u_0, u_1)\), and \((v_0, v_5)\rightarrow(u_0, u_5)\)\}.

**Matching order computation:** For static graphs, once the matching order is computed, it is used throughout the process. \text{Mnemonic} strictly follows the BFS order of the query tree during filtering. However, since initial embedding \((v_p, v)\) can match any of the query edges \((u_p, u)\), matching order may start from any of the query graph edges. Thus, a different matching order needs to be computed for enumeration starting at every different edge in the query graph. For a tree edge \((u_p, u)\), the path from the node \( u \) to the root query node is placed first in the matching order and the BFS traversal order is used for the rest of the query tree. For a non-tree edge \((u_x, u_y)\), the matching order is \{\((u'_x, u'_y), (u'_x, v_x)\)\} followed by the path from \( u_x \) to the root and the remaining edges in the BFS order. Here, \( u'_x \) and \( u'_y \) are parent nodes of \( u_x \) and \( u_y \) respectively. Once the initial embeddings \( e \) and their corresponding matching order is obtained, multiple instances of \text{enumerator}() are spawned. In snapshot G1, the inserted edge \((v_2, v_6)\) matches \((u_1, u_3)\), thus the matching order is \{\((u_1, u_3), (u_0, u_1), (u_2, u_0), (u_0, u_5), (u_1, u_4), (u_2, u_0)\)\}. Similarly, for edge \((v_0, v_2)\), the matching order is \{\((u_0, u_1),(u_2, u_0), (u_0, u_5),(u_1, u_3),(u_1, u_4), (u_2, u_0)\)\}.

**Duplicates Removal:** When a batch of insertion is processed, since \text{DEBI} is updated for the whole batch before the enumeration, many duplicates can be generated. Embeddings that use 2 or more edges inserted in the current batch are listed for each new edge. For example, after updating \text{DEBI} to Figure 5(c) at time 1, two new embeddings are obtained, i.e., the query edges \{\((u_1, u_3), (u_0, u_1), (u_2, u_0), (u_0, u_5), (u_1, u_4), (u_2, u_0)\)\} are matched to edges with ids \{\text{10, 11, 0, 12, 4, 7, 9}\} and \{\text{10, 11, 0, 12, 4, 2, 9}\} respectively. Same two embeddings are obtained for each of three edge insertions, i.e., \((v_2, v_6)\), \((v_0, v_2)\), and \((v_0, v_3)\) even though they are formed only after inserting the last edge. Similar problem also arises for a deletion batch when removed embedding contains two or more edges on the current batch.

\text{Mnemonic} uses a masking technique to eliminate the duplicates. For each edge in query graph, a different masking array of size \(|E_Q|\) bits is used, where the edges with mask bits set cannot use edges in the current batch as their match in embedding. Table I shows a sample mask array for edges in our query graph. In snapshot G1, the edge \((v_2, v_3)\) when starting at \((u_1, u_3)\) cannot use \((v_0, v_2)\) as match for \((u_0, u_1)\) or \((v_0, v_5)\) as match for \((u_0, u_5)\). However, when starting enumeration from \((u_0, u_1)\) for new edge \((v_0, v_2)\), the masks for neither

| Table I. Sample masking table for a query graph. |
|-----------------------------------------------|
| \(u_0, u_1\) | \(u_2, u_4\) | \(u_0, u_3\) | \(u_3, u_1\) | \(u_1, u_4\) | \(u_2, u_0\) |
| \(000000\) | \(010111\) | \(001011\) | \(000111\) | \(000110\) | \(000000\) |
of \((u_0, u_5)\) and \((u_1, u_3)\) are set. Thus we can use \((v_0, v_5)\) and \((v_2, v_3)\) as matches for \((u_0, u_5)\) and \((u_1, u_3)\) respectively.

VII. Evaluation

Mnemonic is implemented using C++ and OpenMP on Linux, and evaluated on a server with Intel Xeon Gold 6126 24-core 2.60GHz CPU with 512 GB of memory. We demonstrate its ability to support different stream types (Section VII-A), search granularity (Section VII-B), and embedding types (Section VII-C). In addition, we evaluate its memory consumption (Section VII-D), and parallelization (Section VII-E). We compare against four state-of-the-art solutions: (1) TurboFlux [12] and (2) BigJoin [16], two systems for subgraph matching on a streaming graph; (3) CECI [9], a system for subgraph isomorphism on the static graph; and (4) Li et al. [20] for time-constrained isomorphism on the streaming graph.

We use 3 datasets for experiments. (1) NetFlow contains anonymized passive traffic traces collected from high-speed internet backbone links, with 18,520,759 triplets in total. Each triplet represents the source, destination, and transport layer protocol of a flow event. There is no deletion of edges or vertices on this graph, i.e., an insert-only stream. (2) LSBench consists of simulated RDF social network activities for 100K users generated using the Linked Stream Benchmark data generator [34]. Each event represents one of many activities such as posts, comments, and locations. The stream contains 23,320,426 triplets, out of which the first 20.9M are insertions and 10% of the remaining 2.3M updates are deletions. The deletions are generated by randomly picking edges from the first 20.9M edges. The deleted edges are indicated on stream by negating the both endpoints, i.e., \((-1, -3, l)\) if edge \((1, 3, l)\) is deleted. (3) LANL contains the stream of NetFlow records collected from network devices inside Los Alamos National Lab [35]. The events recorded during the first 3 days are used, 540M events in total. These graphs have respectively 1, 1, and 6 node types and 8, 45, and 3 edge types.

Two types of queries of different sizes are used, i.e., acyclic (tree) and cyclic (graph) queries. We followed the technique adopted by TurboFlux [12] for query generation in NetFlow and LSBench datasets. In total, 100 tree queries each of size 3, 6, 9, and 12 (denoted as \(T_3, T_6, T_9, \) and \(T_12\)), and 100 graph queries each of size 6, 9, and 12 (denoted as \(G_6, G_9, \) and \(G_12\)) are generated. The average runtime for 100 queries is reported unless specified otherwise. For LANL, 100 tree and graph queries are generated where the query edges have timestamps from the corresponding edge in the data graph.

These queries are used to evaluate the performance of temporal subgraph matching in Section VII-C.

A. Varying Stream Types

Insertion only stream: The comparison of runtime between TurboFlux and Mnemonic for isomorphism search on NetFlow is presented in Figure 6. In order to assess different amounts of updates, three stream sizes of 0.2M, 2M, and 10M are chosen. For each stream size, the remaining edges from the total of 18.5M edges are loaded in the initial graph (or the first snapshot for Mnemonic). Both systems are set to timeout after two hours. Mnemonic uses a batch size of 16K.

Figure 6(a) shows the average runtime for processing 0.2M edge insertions. Mnemonic comes on top with an overall average speedup of \(7.6 \times\) for all cases (maximum \(18.9 \times\) on \(T_{12}\), minimum \(1.6 \times\) on \(T_3\)). The main advantage comes from batching of edges (Section VII-B), and finer grained enumeration parallelization. In 2M and 10M stream sizes, the average speedup of \(5.9 \times\) and \(3.2 \times\) are obtained. At first glance it seems like speedup decreases with the increase in stream size. This is due to the skewed statistics from excluding the timed-out queries. For 2M edge updates, eight queries timed out in Mnemonic, whereas in addition to these eight, another 29 queries timed out for TurboFlux. The timed out queries in Mnemonic are corner cases with a very high number of matches. For 10M edge insertions, none of the graph queries were finished in TurboFlux within 2 hours while 11, 15, and 23 of \(T_3, T_6, T_9, \) and \(T_{12}\) respectively timed out. In contrast, all tree queries finished in Mnemonic, while 7, 13, and 16 queries from \(G_6, G_9, \) and \(G_{12}\) timed out respectively. If only the runtime of queries finished by TurboFlux were included, the average speedup would be \((9.5 \times \text{and } 10.2 \times\) for 2M and 10M insertions respectively. The runtime is fairly consistent for most queries, and standard deviation error is mostly due to a few long-running edge cases.

![Figure 6: Runtime of TurboFlux and Mnemonic on NetFlow](image)

Figure 7 shows the CPU usage per core over the computation lifetime for query \(T_9 \rightarrow Q_1\) for Mnemonic and TurboFlux.

Figure 8 compares the average number of edges traversed for reflecting the effect of a single edge update on DEBI. Since the edges share the traversal within the insertion/deletion batch, the average number of traversals per insert/delete decreases as we increase the batch size. With the increase in the query size number of edges traversed tends to increase as traversal diameter increases. But, this increment follows query size sub-linearly because larger query graphs have...
more selective nodes and that can eliminate more edges from traversal path.

While the number of traversals directly translates to the time for DEBI update time, the runtime for embedding enumeration is dependent on the number of embeddings. Since enumerating each embedding is an embarrassingly parallel problem, the batching combined with fine grained work decomposition in Mnemonic (Section VI) keeps all the workers busy during the enumeration phase. Figure 8 shows the cpu usage per core over the runtime for searching a query from $T_{9}$ on NetFlow graph for Mnemonic and TurboFlux. As expected, Mnemonic has consistently better cpu usage over the program's life compared to Turboflux. The two extreme drops are because of the long tails formed where single edge update produces many matches.

![Figure 8: Traversals per edge update (insertion/deletion) for different batch size. It is the sum of number of edges traversed during top-down and bottom-up filtering.](image)

BigJoin [16] is faster than Mnemonic and TurboFlux for smaller queries. We computed its runtime for homomorphism of some common queries as listed in Table II. BigJoin performed worse on rectangle and dual-triangle than 5-cliques because it cannot take as much advantage of intersection to minimize match set in sparse queries. As the query gets bigger, the performance worsens for BigJoin as predicted by [11]. It takes an average of 6.75 hours to finish $T_{9}$ on NetFlow (stream size = 2M) and longer than 12 hours for $T_{12}$, $G_{9}$, and $G_{12}$, i.e., an order magnitude longer than Mnemonic.

![Table II. Query Runtime on NetFlow (Seconds)](image)

**Insertion-deletion stream:** The runtime comparison for LS-Bench is presented in Figure 9. Edges are streamed in the batch of 16K. Both positive and negative embeddings are listed. No negative embeddings are the matches of a query graph that should be removed after an edge deletion. Mnemonic is able to outperform TurboFlux by $3.27 \times$ on average. The speedup is lower than for NetFlow because it has fewer parallel edges and thus index structures become somewhat similar. Nonetheless, the advantage of Mnemonic is clear. Note that the runtime is more consistent here, likely because LSBench is a random graph whereas NetFlow follows power law.

![Figure 9: Runtime of TurboFlux and Mnemonic on LSBench](image)

**Sliding-window stream:** We only present the runtime of Mnemonic here as none of the existing systems were able to work out of the box under this scenario. The window and stride sizes are set to 24 hours and 10 minutes respectively. The edges are repeatedly dropped from the tail of windows. The runtime for the isomorphic search of different query sizes is reported in Figure 10. The runtime increases almost linearly to the size of queries. For three days of LANL data, even the slowest queries are completed within two hours since search space is confined within the window.

![Figure 10: Isomorphism on LANL with sliding window](image)

**Static Graph Solution:** Figure 11 compares the average runtime per snapshot between Mnemonic and CECI [9] on LANL. While, we perform incremental computation, CECI performs subgraph matching from scratch on each snapshot. Using a 24-hour window and a 15-minute stride, 96 snapshots are generated starting at the beginning of day 2. Isomorphic search for all the queries is performed on every snapshot and the average runtime over all the snapshots is reported. As expected, Mnemonic easily outperforms CECI by $42 \times$ per snapshot on average. However, on the first snapshot only CECI was slightly better ($\times1.08$) compared to Mnemonic. This shows that while dense index structures like CECI are better for enumeration (due to coalesced memory access), they are not efficient for streaming graphs.

![Figure 11: Comparison of runtime per snapshot against CECI](image)

**B. Varying Search Granularity**

Figure 12 shows the average speedup for $G_{6}$ and $T_{6}$ as we increase the batch size in NetFlow (stream size = 2M). Tree queries $T_{3}$, $T_{6}$, $T_{9}$, and $T_{12}$ achieve the maximum speedup of $4.2 \times$, $9.7 \times$, $10.6 \times$, and $8.8 \times$ respectively. Similarly, $G_{6}$, $G_{9}$ for $G_{12}$ obtain the maximum speedup of $9.7 \times$, $8.9 \times$, and $10.2 \times$ respectively. Note that we use the same number (1) of thread for all the batch sizes, to show that the benefit comes from shared traversal.

![Figure 12: Scalability over batch size.](image)

The frontier computation, top-down filtering, bottom-up filtering, and enumeration are all parallelized using OpenMP. Since DEBI is indexed with edgedl, both read and write are thread-safe, as two threads never process the same edge.
concurrently. The speedup with increasing thread count for \( T_6 \) and \( G_6 \) is shown in Figure 13 where the batch size is fixed to 16K in NetFlow (stream size = 2M). The average speedup among all queries for 24 threads is 5.22×.

C. Varying Embedding Types

Figure 14 compares the runtime between Mnemonic and TurboFlux for homomorphic enumeration on the NetFlow with 2M insertions. Since the system does not have to perform an injection check for each embedding, the process is faster compared to isomorphism and none of the queries timed out. Mnemonic is on average 4.2× faster than TurboFlux.

D. Memory Consumption

We measure the memory consumption trend over 90 snapshots starting from the beginning of day 2 in the LANL data using the window size of 24 hours and stride of 10 minutes. Figure 17 shows the effect of memory reclaiming. Even though the number of events in the 24 hours window remains similar, the total number of edge entries (thus a placeholder in memory) increases rapidly. By reusing the memory of deleted edges, the growth in data size reduces from 67% to 23% over 90 snapshots. In addition, Mnemonic allows the periodic resets, where we can discard the cumulative index at any point in time (e.g., the 91st snapshot) and rebuild the DEBI starting from that snapshot.

For queries requiring a longer context, Mnemonic backs up older edges and DEBI to the disks. We utilize the disk storage to run queries with a 3-day search window in LANL where we only maintain events from last 24-hour in memory. Table III shows the maximum space consumed in memory and disk as well as the overhead involved in maintaining DEBI in disk and fetching candidates on enumerators(). While runtime suffers from higher candidate access latency, it allows to perform subgraph matching in large search window.

The space requirement of DEBI is \( O(|E| \times |V_q|) \) bits. Here \(|E|\) is number of data edges, and \(|V_q|\) is the number of query nodes. This is a sparse data structure, i.e., it needs to store the information between data and query edge pairs even if they do not match each other. For a data graph with 1 Billion edges and a query graph with 10 nodes, 10B bits i.e., 1.25 GB is needed for DEBI. In a dense representation of graph like adjacency list or compressed sparse row, it takes \( 4 \times 10^9 \) Bytes \( \approx 4 \) GB for storing the graph, where 4 Bytes is used to represent a node id. With additional edge and node attributes, graph size is a much bigger space concern compared to DEBI. Fortunately, with a vast pool of research focusing on external memory support for graphs, we leveraged some of those systems [22] to extend the Mnemonic beyond memory capability.

| Query | Storage size required(GB) | Overhead(%) |
|-------|--------------------------|-------------|
| Memory | Disk | DEBI mgmt. | enumeration |
| \( T_3 \) | 14.575 | 26.98 | 5.8% | 3.5% |
| \( T_6 \) | 26.1875 | 69.34 | 3.1% | 7.1% |
| \( T_9 \) | 39.8375 | 102.13 | 4.9% | 7.9% |
| \( T_{12} \) | 52.7 | 102.13 | 8.2% | 9.2% |
| \( G_6 \) | 34.375 | 67.65 | 5.6% | 7.6% |
| \( G_9 \) | 53.375 | 106.87 | 6.3% | 7.8% |
| \( G_{12} \) | 73.7 | 132.27 | 7.3% | 9.3% |
Dense candidate stores [8], [9], which have similar worst space complexity, only store the data nodes that are potential matches of a given query node. Thus, the space advantage of dense stores is evident when the queries are very selective. They also provide a coalesced memory access while fetching the candidate matches of a query node in contrast to stride access pattern in DEBI. This provides the dense storage approach an advantage during enumeration. However, updating the dense storage requires searching through multiple key-value stores and each can take $O(|V|)$. In DEBI, we can add, remove and find the entries to be updated in constant time.

VIII. CONCLUSION

Mnemonic presents a programmable subgraph matching system, that allows users to tailor the underlying components according to the nature of data and problem characteristics. Its utility on various types of data streams and different embeddings have been evaluated and compared against state of the art systems. We plan to open-source the system as soon as possible, as well as provide capability for distributed computation and querying multiple patterns concurrently as next additions. We strongly believe that it can democratize the process of implementing a subgraph matching system outside the small pool of researchers.

REFERENCES

[1] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, “The ldbc social network benchmark: Interactive workload,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 619–630.

[2] G. Sadowski and P. Rathie, “Fraud detection: Discovering connections with graph databases,” Tech. Rep., 2017.

[3] Verizon Inc., “2018 data breach investigations report,” Tech. Rep., 2018.

[4] S. Sun and Q. Luo, “In-memory subgraph matching: An in-depth study,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD ’2015, pp. 405–418.

[5] W.-S. Han, J. Lee, and J.-H. Lee, “Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases,” in International Conference on Management of Data, SIGMOD, 2013, pp. 337–348.

[6] H. He and A. K. Singh, “Graphs-at-a-time: Query language and access methods for graph databases,” in International Conference on Management of Data, SIGMOD, 2006.

[7] X. Ren and J. Wang, “Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs,” Proceedings of the VLDB Endowment, vol. 8, no. 5, pp. 617–628, Jan. 2015.

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, “Efficient subgraph matching by postponing cartesian products,” in International Conference on Management of Data, ACM, 2016, pp. 1199–1214.

[9] B. Bhattacharj, H. Liu, and H. H. Huang, “Ceci: Compact embedding cluster index for scalable subgraph matching,” in Proceedings of the 2019 International Conference on Management of Data, ser. SIGMOD ’19, 2019.

[10] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, “Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together,” in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1429–1446.

[11] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, “Rapidmatch: A holistic approach to subgraph query processing,” Proc. VLDB Endow., vol. 14, no. 2, p. 176–188, Oct. 2020.

[12] K. Kim, J. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, “Turboflux: A fast continuous subgraph matching system for streaming graph data,” in Proceedings of the 2018 International Conference on Management of Data, ACM, 2018, pp. 411–426.

[13] S. Choudhury, L. B. Holder, G. Chin, K. Agarwal, and J. Feo, “A selectivity based approach to continuous pattern detection in streaming graphs,” in Proceedings of 15th International Conference on Extending Database Technology, 2015.

[14] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrahi, “Dualsim: Parallel subgraph enumeration in a massive graph on a single machine,” in Proceedings of the 2016 International Conference on Management of Data. ACM, 2016, pp. 1231–1245.

[15] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, “Taming subgraph isomorphism for rdf query processing,” Proceedings of the VLDB Endowment, vol. 8, no. 11, pp. 1238–1249, 2015.

[16] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, “Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows,” Proceedings of the VLDB Endowment, vol. 11, no. 6, pp. 691–704, 2018.

[17] C. Kankanange, S. Sahu, A. Mhedbibi, J. Chen, and S. Salihoglu, “Graphflow: An active graph database,” in Proceedings of the 2017 ACM International Conference on Management of Data, ser. SIGMOD ’17, 2017, p. 1695–1698.

[18] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, “Capturing topology in graph pattern matching,” Proceedings of the VLDB Endowment, vol. 5, no. 4, pp. 310–321, 2011.

[19] W. Fan, X. Wang, and Y. Wu, “Incremental graph pattern matching,” ACM Transactions on Database Systems (TODS), vol. 38, no. 3, p. 18, 2013.

[20] Y. Li, L. Zou, M. T. Özs, and D. Zhao, “Time constrained continuous subgraph search over streaming graphs,” in 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 1082–1093.

[21] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, “Parallel subgraph listing in a large-scale graph,” in International Conference on Management of Data, SIGMOD, 2014, pp. 625–636.

[22] Y. Tian and J. M. Patel, “Tale: A tool for approximate large graph matching,” in International Conference on Data Engineering, 2008.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. ACM, 2010, pp. 155–146.

[24] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, “Arabesque: a system for distributed graph mining,” in Symposium on Operating Systems Principles. ACM, 2015, pp. 425–440.

[25] P. Zhao and J. Han, “On graph query optimization in large networks,” Proceedings of the VLDB Endowment, vol. 3, no. 1–2, pp. 340–351, 2010.

[26] M. Shang, Y. Zhang, X. Lin, and J. X. Yu, “Taming verification hardness: an efficient algorithm for testing subgraph isomorphism,” Proceedings of the VLDB Endowment, vol. 1, no. 1, pp. 364–375, 2008.

[27] X. Ren and J. Wang, “Multi-query optimization for subgraph isomorphism search,” Proceedings of the VLDB Endowment, vol. 10, no. 3, pp. 121–132, 2016.

[28] L. Lai, L. Qin, X. Lin, and L. Chang, “Scalable subgraph enumeration in mapreduce,” Proceedings of the VLDB Endowment, vol. 8, no. 10, pp. 974–985, 2015.

[29] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, “Efficient subgraph matching on billion node graphs,” Proceedings of the VLDB Endowment, vol. 5, no. 9, pp. 788–799, 2012.

[30] H. Q. Ngo, C. Ré, and A. Rudra, “Skew strikes back: New developments in the theory of join algorithms,” SIGMOD Rec., vol. 42, no. 4, p. 5–16, Feb. 2014.

[31] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang et al., “Distributed subgraph matching on timely dataflow,” Proceedings of the VLDB Endowment, vol. 12, no. 10, pp. 1099–1112, 2019.

[32] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, “Livegraph: A transactional graph storage system with purely sequential adjacency list scans,” arXiv preprint arXiv:1910.05773, 2019.

[33] “Binary for TurboFlux,” 2020, accessed: 2018-08-07. [Online]. Available: https://sites.google.com/a/dhlab.postech.ac.kr/kmkim/turboflux/.

[34] “ibench,” 2020, accessed: 2020-01-14. [Online]. Available: https://code.google.com/archive/p/ibench/.

[35] M. J. Turcotte, A. D. Kent, and C. Hash, “Unified host and network data set,” 2017.