TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data

Kyongmin Kim[†§] In Seo[†§] Wook-Shin Han[†*]
Jeong-Hoon Lee[†] Sungpack Hong[‡] Hassan Chafi[‡]
Hyungyu Shin[†] Geonhwa Jeong[†]
Pohang University of Science and Technology (POSTECH), Korea[†], Oracle Labs, USA[‡]
{kmkim, iseo, wshan, jhlee, hgshin, ghjeong}@dblab.postech.ac.kr[†]{sungpack.hong, hassan.chafi}@oracle.com[‡]

ABSTRACT

A dynamic graph is defined by an initial graph and a graph update *stream* consisting of edge insertions and deletions. Identifying and monitoring critical patterns in the dynamic graph is important in various application domains such as fraud detection, cyber security, and emergency response. Given a dynamic data graph and a query graph, a continuous subgraph matching system reports *positive* matches for an edge insertion and reports *negative* matches for an edge deletion. Previous systems show significantly low throughput due to either repeated subgraph matching for each edge update or expensive overheads in maintaining enormous intermediate results. We present a fast continuous subgraph matching system called TurboFlux which provides high throughput over a fast graph update stream. TurboFlux employs a concise representation of intermediate results, and its execution model allows fast incremental maintenance. Our empirical evaluation shows that TurboFlux significantly outperforms existing competitors by up to *six* orders of magnitude.

ACM Reference Format:
Kyongmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196917

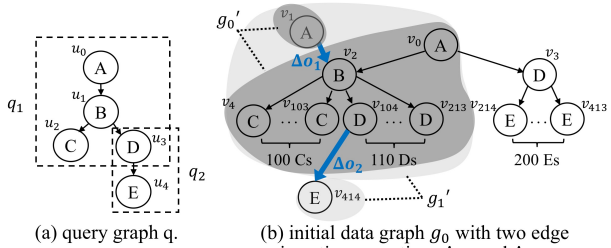
1 INTRODUCTION

A dynamic graph is defined by an initial graph and a graph update stream of edge insertions and deletions. Typical examples of dynamic graphs include social media streams, cyber data sources such as computer network traffic, phone call networks, and financial transaction networks. For example, social media streams consist of vertices such as people, movies, or images and edges, such as friendship, while network traffic streams consist of vertices such as IP addresses and edges such as class of network traffic (e.g., TCP).

Identifying and monitoring critical patterns in a dynamic graph is important in various application domains [13] such as fraud detection, cyber security, and emergency response. In banking applications, fraudsters organize into fraud rings, which can be detected by subgraph matching using a query graph having a ring shape [25]. Note that it is important to find such fraudsters in real-time. Cyber security applications should detect cyber intrusions and attacks in computer network traffic as soon as they appear in the data graph [6]. Various emergent monitoring cases, such as worm spread and virus attack can also be modeled as query graphs [6].

In order to identify and monitor such patterns, we study the continuous subgraph matching problem. Given an initial graph g_0, a graph update stream Δg consisting of edge insertions and deletions $(\Delta o_1, \Delta o_2, \cdots)$ and a query graph q, $g_i = g_{i-1} \oplus \Delta o_i$, where $\oplus$ means that Δo_i is applied to g_{i-1}, and $M(g, q)$ denotes the set of subgraph matching results between data graph g and q. Here, subgraph matching will be formally defined in Section 2. Then the continuous subgraph matching problem is to report $M(g_i, q)$ - $M(g_{i-1}, q)$ (or $M(g_{i-1}, q)$ - $M(g_i, q)$) as *positive (or negative) matches* for each edge insertion (or deletion) operation Δo_i.

Figure 1 shows an example of continuous subgraph matching. When a query graph q is given as in Figure 1a, and an initial graph g_0 having 412 edges is updated with two edge insertion operations Δo_1 (edge insertion between v_1 and v_2) and Δo_2 (edge insertion between v_{104} and v_{414}) as in Figure 1b, we find all positive matches for each operation. Δo_1 reports nothing, since there is no data edge in g_1 that matches the query edge (u_3, u_4). When Δo_2 occurs, it incurs 200 positive matches in g_2 since it matches the query edge (u_3, u_4): The first 100 positive matches are due to the induced graph consisting of v_0 (mapping to u_0), v_2 (mapping to u_1), $v_4 \sim v_{103}$ (mapping to u_2), v_{104} (mapping to u_3), and v_{414} (mapping to u_4). Here, u_2 matches 100 Cs in g_2, and thus we have 100 positive matches. The next 100 positive matches are due to the induced graph consisting of v_1 (mapping to u_0), v_2 (mapping to u_1), $v_4 \sim v_{103}$ (mapping to u_2), v_{104} (mapping to u_3), and v_{414} (mapping to u_4).



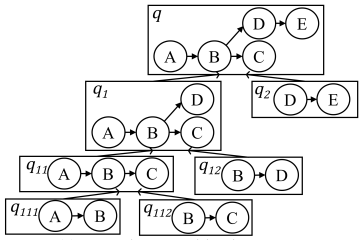
(a) query graph q. (b) initial data graph g_0 with two edge insertion operations Δo_1 and Δo_2.

Figure 1: An example of continuous subgraph matching.

[§]The first two authors contributed equally to this work
[*]corresponding author

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

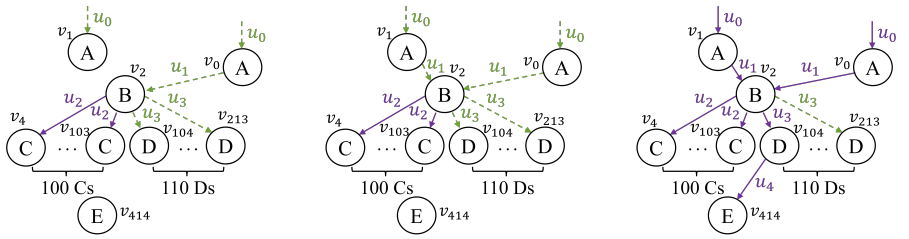
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196917



(a) query decomposition in SJ-Tree.

data graph	q	q_1	q_2	q_{11}	q_{12}	q_{111}	q_{112}
g_0	0	11,000	0	100	110	1	100
g_1 ($= g_0 \oplus \Delta o_1$)	0	22,000	0	200	110	2	100
g_2 ($= g_1 \oplus \Delta o_2$)	200	22,000	1	200	110	2	100

(b) number of partial solutions at each node in SJ-Tree for g_0, g_1, and g_2.



(c) DCG for initial graph g_0 (d) DCG for g_1 (e) DCG for g_2

Figure 2: An example of SJ-Tree and DCG for Figure 1.

A naive approach to solve this problem is to re-compute subgraph matching for each update operation. However, since popular subgraph matching problems such as subgraph isomorphism and graph homomorphism are expensive [14, 17], recomputing from scratch whenever the data graph is updated is practically infeasible [10].

Efforts to support incremental graph pattern matching seemed to enjoy some success. In [10], IncIsoMat identifies and extracts a subgraph g'_{i-1} from g_{i-1} which can be affected by Δo_i. IncIsoMat executes any subgraph matching method over the subgraph to obtain the changes to the original matches of g'_{i-1} by performing the set difference between the original matches and the new matches. Graphflow [16] applies a worst-case optimal join algorithm called *Generic Join* [22] to incrementally evaluate subgraph matching for each update. SJ-Tree [7] uses a left-deep tree, where an internal node in SJ-Tree corresponds to a subgraph containing more than two connected query vertices, and a leaf node corresponds to a subgraph containing two adjacent query vertices. Figure 2a shows how SJ-Tree constructs the left-deep tree for the query graph q in Figure 1a. SJ-Tree stores materialized results for each subgraph of the query. Whenever there are any new insertions in a node n in SJ-Tree, those new insertions are joined with the materialized results stored in the sibling node of n. SJ-Tree then inserts those new join results to the parent node of n, which triggers a series of joins in a bottom-up fashion. On edge insertion, SJ-Tree first finds leaf nodes that match the edge inserted. For each leaf node n' matching the edge inserted, SJ-Tree inserts the edge to n' and start join the edge with the materialized results stored in the sibling node of n'. We will explain these methods in more detail in Section 2.2.

However, all of these methods could incur serious performance problems. Since IncIsoMat does not maintain partial solutions, there is no maintenance overhead. Although IncIsoMat executes subgraph matching on g'_{i-1} rather than g_{i-1}, this repeated-search based method could incur significant overhead in extracting g'_{i-1}, performing subgraph matching on g'_{i-1} and $g'_{i-1} \oplus \Delta o_i$, and computing the set difference for each Δo_i. Moreover, g'_{i-1} could include most of the vertices and edges in g_{i-1}, since the average distance between two vertices in real world graphs is extremely small [28]. Although Graphflow can find positive/negative matches without computing the set difference, it needs to compute the join from scratch since it does not *maintain* any intermediate results. Moreover, Graphflow needs to evaluate subgraph matching for each Δo_i, even if Δo_i does not generate any positive/negative match. Unlike these two methods, SJ-Tree exploits materialized results. However, due to the expensive maintenance cost of materialized results, SJ-Tree has a serious performance problem. Figure 3 compares the performance and storage costs of existing methods with our method (i.e. TurboFlux). The goal of this paper is to find the sweet spot between the performance and storage costs. A more detailed empirical comparison will be presented in Section 5.

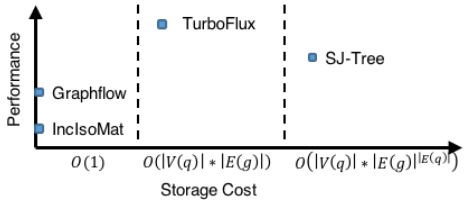


Figure 3: Trade-off between performance and storage costs.

These problems of existing methods motivated us to develop the novel concept of the *data-centric* graph (DCG) for intermediate results. The DCG maintains candidate query vertices for each data vertex using a graph structure such that a data vertex can appear at most once. The salient features of the DCG are summarized as follows: 1) Since the DCG itself is a graph, whenever we need to update the DCG, we can directly access corresponding vertices in the DCG. 2) The worst storage complexity for the DCG is $O(|V(q)| * |E(g)|)$. 3) Unlike SJ-Tree which finds solutions by joining partial solutions, TurboFlux finds solutions by traversing the DCG.

Our contributions are as follows: 1) We propose an efficient continuous subgraph matching system, TurboFlux, to resolve the problems of existing methods. 2) We propose the DCG to represent partial solutions in a compact form. 3) We propose a formal model, the edge transition model, for TurboFlux algorithms to efficiently maintain intermediate results and to report positive/negative matches without computing the set difference. 4) Extensive experiments using both synthetic and real datasets show that TurboFlux outperforms existing competitors by up to orders of magnitude.

The rest of the paper is organized as follows. Section 2 introduces fundamental graph concepts, formally defines the continuous subgraph matching problem, and reviews related work. Section 3 introduces a novel representation of intermediate results called the DCG and proposes the edge transition model. Section 4 explains the algorithms of TurboFlux in detail. Section 5 presents experimental results. Section 6 gives our conclusion.

2 BACKGROUND

2.1 Problem Definition

A labeled graph is defined as (V, E, L) such that V is a set of vertices, E is a set of edges, and L is a label function that maps a vertex to a set of labels.

Definition 1. [17] *A graph $q = (V, E, L)$ is homomorphic to a subgraph of a data graph $g = (V', E', L')$ if there is a mapping (or a match) $m : V \rightarrow V'$ such that 1) $\forall u \in V$, $L(u) \subseteq L'(m(u))$, and 2) $\forall (u_i, u_j) \in E$, $(m(u_i), m(u_j)) \in E'$.*

Since subgraph isomorphism can be obtained by just checking the injective constraint [17], we use the graph homomorphism as our default matching semantics. We also report experimental results using subgraph isomorphism as matching semantics in Appendix B.1. Note that we omit edge labels for ease of explanation, while the actual implementation of TurboFlux and our experiments support edge labels.

Definition 2. *A graph update stream Δg is a sequence of update operations $(\Delta o_1, \Delta o_2, \cdots)$, where Δo_i is a triple (op, v, v') such that op is the type of the update operation such as edge insertion/deletion of an edge (v, v').*

Now, we define some notation used throughout this paper. Each labeled edge is of the form (v, l, v'), where v and v' are vertices, and l is an edge label. u is used for a query vertex, while v is used for a data vertex. u_s is used for denoting the starting query vertex, and v_s is used for denoting a starting data vertex matched with u_s. $deg(v)$ denotes the degree of a vertex v. Given a vertex u, $ID(u)$ denotes the ID of u. Given a tree, $P(u)$ is the parent of u, and $Children(u)$ is the set of children of u. Given a graph, $Adj(v)$ denotes the set of adjacent vertices of v. If there is a path from v to v', we denote it as $v \rightarrowtail v'$. Note that $v \rightarrowtail v$ denotes a path from v to v with path length 0. Also, a path $\langle v, \ldots, v_m, v' \rangle$ can be denoted as $v \rightarrowtail v_m.v'$.

Definition 3 (Problem Statement). *Given an initial data graph g_0, a graph update stream Δg, and a query graph q, the* continuous subgraph matching *problem identifies all positive/negative matches for each update operation in Δg.*

2.2 Related Work

Subgraph Isomorphism. Subgraph isomorphism has been studied extensively since 1976. Ullmann [29] proposed a seminal work for subgraph isomorphism, which is a backtracking based algorithm. The representative algorithms in subgraph isomorphism are VF2 [8], QuickSI [26], GraphQL [15], SPATH [32], TurboISO [14], TurboISOBoosted [24], and CFL [5]. These algorithms exploit good matching orders and effective pruning rules in order to boost subgraph matching. All of these algorithms assume that the input graph is static, while our work focuses on the dynamic graph.

Graph Homomorphism. RDF query processing follows graph homomorphism. The representative RDF systems are RDF/3X [21], gStore [33], Trinity.RDF [31], and TurboHom++ [17]. RDF/3X executes graph homomorphism by using merge-based join. gStore [33] supports graph homomorphism using the filter-and-refinement strategy with VS*-index. Trinity. RDF is a distributed RDF processing system which uses a distributed key-value store. All of these systems assume a static data graph (or a specific snapshot).

Relaxed pattern matching. Graph simulation uses relaxed pattern matching semantics. Note that different matching semantics have their own important applications. An incremental pattern detection using graph simulation is studied in [11]. There are variants of graph simulation such as resource-bounded query answering [12] and strong simulation [20]. For a series of edge updates over time, [27] incrementally builds connected components for a query graph according to graph simulation semantics. Due to the different semantics between graph homomorphism (or subgraph isomorphism) and graph simulation, [27] would generate many false alarms which must be discarded if we want to support graph homomorphism. Furthermore, in order to support graph homomorphism (or subgraph isomorphism), we have to repeatedly generate intermediate results for each candidate connected component to derive a good matching order, as in [5, 14]. Instead, TurboFlux *incrementally* maintains intermediate results for each update and derives a matching order directly from the intermediate results maintained.

Frequent subgraph mining. Unlike continuous subgraph matching where a query graph is given, in frequent subgraph mining, a set of frequent patterns in a data graph are mined. [1] proposes IncGM+, an incremental frequent subgraph mining algorithm for a dynamic graph. IncGM+ uses a concept called *fringe* which is a set of patterns on the border between frequent and infrequent ones. When an edge is inserted or deleted, IncGM+ finds positive/negative matches for patterns in *fringe* to find patterns that are changed from frequent to infrequent or vice versa. IncGM+ uses a simple method of continuous subgraph matching that finds positive matches by applying subgraph isomorphism starting from the inserted edge, since the continuous subgraph isomorphism method itself is not the focus of the paper. It would be interesting future work to apply our technique to this problem to speed up incremental frequent subgraph mining.

Continuous matching in a single machine. IncIsoMat [10] identifies a subgraph g'_{i-1} which is affected by each update operation Δo_i, executes subgraph matching for g'_{i-1} and $g'_{i-1} \oplus \Delta o_i$, and computes the set difference between them. Here, g'_{i-1} consists of data vertices/edges within the diameter of q from two vertices of the edge updated by Δo_i, where the diameter of q is defined as the length of the longest of all pairs' shortest paths in q by regarding q as an undirected graph. For example, in Figure 1b, g'_0 and g'_1 are such reduced subgraphs for the update operations Δo_1 and Δo_2. Here, g'_0 contains vertices within the distance 3 from v_2 and edges among them, since the diameter of q is 3. Note that g'_0 does not contain v_3 and $v_{214} \sim v_{413}$, since there is no query edge (u, u') in q such that the labels of u and u' are A and D, respectively.

Graphflow [16] applies a worst-case optimal join algorithm called *Generic Join* [22] to incrementally evaluate subgraph matching for each update without maintaining intermediate results. For each query edge (u, u') that matches an updated edge (v, v'), Graphflow evaluates subgraph matching starting from a partial solution $\{(u, v), (u', v')\}$.

SJ-Tree [7] decomposes q into two smaller subgraphs q_1 and q_2 where q_1 and q_2 share the common query vertex u_3 in Figure 1. It stores the corresponding partial solutions in a hash table for each subgraph. By performing join between these two collections of partial solutions, we can obtain complete solutions (or matches) for q. q_1 is recursively decomposed into smaller subgraphs until all subgraphs in leaf nodes have only two query vertices. Figure 2a shows how SJ-Tree decomposes a query q in Figure 1 into six smaller subgraphs (q_1, q_2, q_{11}, q_{12}, q_{111}, and q_{112}), and performs three joins, $q_{111}\bowtie q_{112}$, $q_{11}\bowtie q_{12}$, and $q_1\bowtie q_2$. Figure 2b shows the number of partial solutions stored at each node for g_0, g_1, and g_2. Observe that q_1 stores 11,000 partial solutions $\{(u_0, v_0), (u_1, v_2), (u_2, v_4), (u_3, v_{104})\}, \cdots, \{(u_0, v_0), (u_1, v_2), (u_2, v_{103}), (u_3, v_{213})\}$ for g_0, even though the number of complete solutions is zero. When Δo_1 (v_1, v_2) is inserted into g_0, the number of partial solutions for q_1 increases twice (=22,000), although the number of complete solutions remains zero. The worst storage complexity for SJ-Tree is $O(|V(q)| * |E(g)|^{|E(q)|})$ when a data graph g has $|E(g)|$ edges and a query graph has $|V(q)|$ vertices. This is because the root node of SJ-Tree can have $|E(g)|^{|E(q)|}$ partial solutions in the worst case, and each partial solution in the root node has $|V(q)|$ vertices.

In addition to the explosive size problem of partial solutions, SJ-Tree requires duplication elimination process before inserting partial solutions to nodes in the tree. In order to avoid inserting duplicate partial solutions to nodes in SJ-Tree, it uses the 'generate-and-discard' strategy. Specifically, when a new partial solution is being inserted to a non-leaf node n in SJ-Tree, it generates candidate partial solutions for the sibling leaf node of n. If a partial solution to be inserted is already in the hash table for the sibling of n, SJ-Tree discards it. For example, when an edge (v_1, v_2) is inserted into q_{111} by Δo_1, SJ-Tree generates 100 partial solutions, $\{(u_1, v_2), (u_2, v_4)\}$, ..., $\{(u_1, v_2), (u_2, v_{103})\}$ for the sibling node q_{112} by accessing the adjacent vertices of v_2 and discards them since they already exist. As the results of join between q_{111} and q_{112}, SJ-Tree obtains 100 new partial solutions for q_{11}, $\{(u_0, v_1), (u_1, v_2), (u_2, v_4)\}$, ..., $\{(u_0, v_1), (u_1, v_2), (u_2, v_{103})\}$. Then, for each new partial solution for q_{11}, SJ-Tree first generates partial solutions for the sibling node q_{12}. Here, for each partial solution among these 100 partial solutions, the same 110 partial solutions, $\{(u_1, v_2), (u_3, v_{104})\}$, ..., $\{(u_1, v_2), (u_3, v_{213})\}$, are generated but discarded, since they are already in q_{12}.

[30] also deals with continuous subgraph matching for evolving graphs. However, this method produces *approximate* results only, while our approach generates *exact* results.

Continuous matching in a distributed environment. [23] regards a query as a view and presents a view maintenance algorithm over a dynamic graph when a set of queries is registered. The main focus of [23] is how to merge several views into a merged view in a distributed environment using a variant of the maximal common subgraph. [13] presents continuous subgraph matching in a Pregel-like graph engine. The main focus is on how to decompose a query so that it can reduce the message transformation in a distributed environment. It also exploits the concept of snapshot isolation [4] since the graph could evolve during subgraph matching. TurboFlux can also easily support such semantics if we adopt multi-version concurrency control (MVCC) [19]. It would be an interesting future work to execute TurboFlux under MVCC.

3 INCREMENTAL MAINTENANCE OF INTERMEDIATE RESULTS

Unlike typical subgraph matching, continuous subgraph matching is triggered by each update operation on data vertices and edges. Thus, rather than maintaining candidate data vertices for each query vertex, we need to maintain candidate query vertices for every data vertex. This motivated us to develop a data-centric representation of intermediate results for continuous subgraph matching.

3.1 Data-centric representation

As in [5, 14, 17], we first convert a query graph q to a query tree q' by removing some non-tree edges if q contains cycles. Thus, we assume that a query graph q is a tree in this section. We explain how to handle non-tree edges in Section 4.

Instead of storing partial solutions for query subgraphs as intermediate results as in SJ-Tree, for each data vertex v', we store the corresponding candidate query vertices as incoming edges to v' in intermediate results, which enables us to minimize and maintain intermediate results efficiently. This intermediate result representation is minimal in the sense that, if we remove any edge from the DCG, we have to execute subgraph matching for the data graph to recover such information.

Now, we provide the formal definition of a data-centric representation called the DCG for representing intermediate results as a graph. The DCG is a complete multigraph such that every vertex pair (v_i, v_j) $(v_i, v_j \in V(g))$ has $|V(q)|$ - 1 edges. Here, each edge has a query vertex ID as edge label, and its state is one of NULL/IMPLICIT/EXPLICIT. The following two definitions introduce concepts of implicit and explicit edges. Note that null edges are hypothetical edges in order to explain the edge transition, which will be detailed in the following section. Thus, we do not store edges whose states are NULL in the DCG.

Definition 4. *An explicit edge (v, u', v') represents the candidate query vertex u' for v' such that 1) there exists a data path $v_s \rightarrow v.v'$ that matches $u_s \rightarrow P(u').u'$, and 2) every subtree of u' matches the corresponding subtree of v'.*

Definition 5. *An implicit edge (v, u', v') represents the candidate query vertex u' for v' such that 1) there exists a data path $v_s \rightarrow v.v'$ that matches $u_s \rightarrow P(u').u'$, and there exists a subtree of u' that does not match any subtree of v'.*

Figure 2c shows the DCG for q and g_0 in Figure 1. Here, a solid arrow represents an explicit edge, while a dashed arrow represents an implicit edge. Since $u_0 \rightarrow u_2$ matches $v_0 \rightarrow v_2.v_4$, and u_2 does not have any subtree, the DCG has an explicit edge (v_2, u_2, v_4). However, since $u_0 \rightarrow u_3$ matches $v_0 \rightarrow v_2.v_{104}$ and v_{104} does not have any subtree that matches the subtree of u_3 (i.e., (u_3, u_4)), the DCG has an implicit edge (v_2, u_3, v_{104}).

Since the starting query vertex u_s has no parent in the query tree, we assume that there exists an artificial vertex v_s^* which is connected to every starting data vertex v_s in the DCG with edge label u_s (i.e., (v_s^*, u_s, v_s)). Since v_s^* is artificial, we do not store it in the DCG. For example, the DCG in Figure 2c has two edges with edge label u_0, and the source vertex v_s^* is omitted.

In Figure 2c~e, the DCG stores 213, 214, and 215 edges only for g_0, g_1, and g_2, while the numbers of partial solutions of SJ-Tree for g_0, g_1, and g_2 are 11311, 22412, and 22613, respectively.

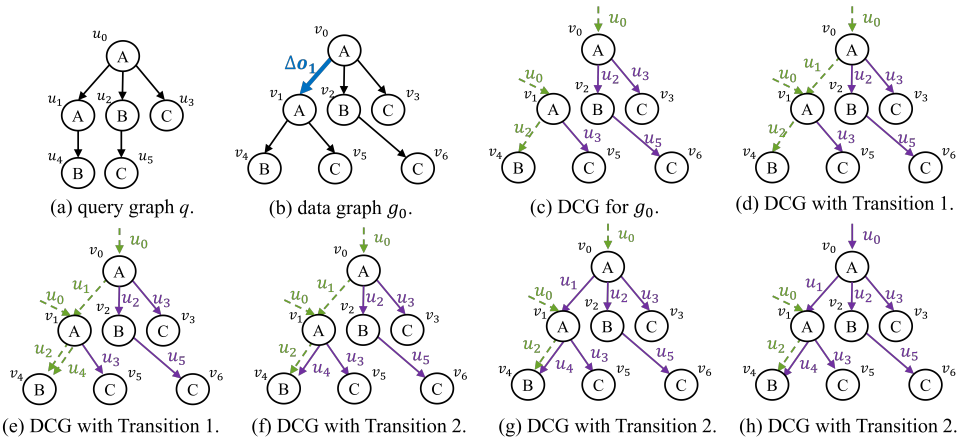


Figure 4: A running example.

Note also that the size of a partial solution for a subquery in SJ-Tree is the number of vertices in the subquery. Moreover, in contrast to the generate-and-discard strategy of SJ-Tree, TurboFlux uses the 'check-and-avoid' approach. It first checks if intermediate results (i.e., edges) to be inserted are already built in the DCG, then we simply avoid building them, which is more efficient than the generate-and-discard strategy. For example, when Δo_1 is applied to g_0, TurboFlux inserts an edge (v_1, u_1, v_2) into the DCG, since (u_0, u_1) matches (v_1, v_2). TurboFlux first checks if every subtree of u_1 is already matched with the corresponding subtree of v_2 in the DCG. Note that these intermediate results are already constructed when we traverse from v_0.

As an optimization for reducing the storage cost, we do not store implicit edges in the DCG. Instead, we use a bitmap for each data vertex v where the i-th bit indicates whether v has any incoming implicit edge whose label is u_i. When the i-th bit is set, we can retrieve the implicit edges to v, $\{(v_p, u_i, v) \mid \text{the } ID(P(u_i))\text{-th bit of } v_p \text{ is set, and } (v_p, v) \text{ matches } (P(u_i), u_i)\}$, by accessing the data graph.

Compared to existing query-centric representations where they store either a set of partial solutions or a set of candidate data vertices for each query vertex u, the DCG has the following three fundamental advantages, two technical advantages and one practical engineering advantage. Note also that a data vertex v can match multiple query vertices, and v can be replicated in the query-centric representation as many times as the number of query vertices. First, in order to support continuous subgraph matching, whenever an update occurs on v, we have to find entries containing v stored in the query centric representation. Thus, we need to devise and maintain an additional *duplicate key index* so that, given a data vertex, we can find relevant entries in the query centric representations. In the DCG, however, we can directly access the adjacency list of v in the DCG, since the DCG itself is a graph. Second, when an inserted (or deleted) edge (v, v') matches multiple query edges $\{(u_i, u_i')\}$, we need to access and update multiple candidate entries, which are stored in different cache lines. However, in the DCG, we only need to access the adjacency list of v and update edges (v, u_i', v')s in this list. Third, since the DCG is a graph itself, we just need to maintain a single code path for the data graph and intermediate results. On the other hand, the query-centric representation, such as candidate regions in [14], is not a graph since data vertices can be replicated in many places. Thus, we need to maintain two different code paths, one for data graph and the other for intermediate results. This could cause serious semantic bugs due to any inconsistency in the two code paths. Thus, having different code paths for the same functionality should be avoided in a single system. In a similar vein, in relational databases, materialized views are maintained as tables.

Now, we compare the candidate subregions in [14], a representative, *static* query representation, with the explicit edge in DCG. Note that there are two fundamental difference between the candidate subregions and the explicit edge. 1) The candidate subregion of Definition 2 in [14] is recursively defined in a top-down fashion, i.e., from the root to leaf. Thus, the candidate subregions are used for storing *parts of complete solutions* only, not for storing partial solutions. On the other hand, the explicit edges in the DCG are also used a) for storing partial solutions and b) for *efficiently* checking whether partial solutions corresponding to subgraphs connected from an explicit edge contribute to complete solutions on edge insertion. Otherwise, those subgraphs need to be traversed on edge insertion, which must be avoided. 2) TurboISO builds a candidate subregion $\text{CR}(u', v)$ for each path $v_s \rightarrow v$. Thus, the candidate subregions are replicated as many times as the number of paths $v_s \rightarrow v$. In the worst case, there exist $|V(g)|^{k-1}$ paths $v_s \rightarrow v$ where k is the number of vertices in $v_s \rightarrow v$. On the other hand, in the DCG, even if there exist a large number of paths $v_s \rightarrow v$ to a subgraph of the DCG, regardless of whether the subgraph corresponds to complete solutions or partial solutions, the subgraph is shared.

The advantages of the DCG are summarized as follows: It is a concise representation; every data vertex appears at most once in the DCG. The DCG stores at most $|V(q)|$ edges for each data edge in g and thus, its worst storage complexity is $O(|V(q)| * |E(g)|)$, which is much more efficient than that of SJ-Tree $|V(q) * E(g)^{E(q)}|$. Since the DCG itself is a graph, whenever we need to update intermediate results, we can directly access corresponding vertices in the DCG.

3.2 Edge Transition Model

The edge transition model uses the DCG to efficiently identify which update operation can affect the current intermediate results and/or contribute to positive/negative matches. Definition 6 formally defines the edge transition model.

Definition 6. *The edge transition model consists of an initial data graph g_0, a graph update stream Δg, a query graph q, a DCG g_{DCG}*

for g_0, six transition rules (Transitions 0 ~ 5), and a rule evaluation algorithm EL. For each update operation Δo_i in Δg, EL transits the state of every relevant edge in g_{DCG} from one state to another according to the transition rules.

Algorithm 1 is called for each update operation in Δg in order to incrementally maintain the DCG. EL repeatedly applies Transition rules 0 ~ 5 until there is no state change in the DCG. These transition rules will be explained below.

Algorithm 1. EL (g_{i-1}, Δo_i, g_{DCG})

```
repeat
    apply Transition rules 0 ~ 5;
until (there is no state transition in g_DCG);
```

Figure 5 shows the state transition diagram, consisting of three states and six transition rules (Transition 0 ~ 5), which demonstrates how one state is transited to another. Here, Transition 1 and 2 are triggered by edge insertion, and Transition 3, 4, and 5 are triggered by edge deletion. Although Transition 0 could be triggered by both edge insertion and deletion, it does not actually transit the states of edges. We detail each transition rule below using the DCG snapshots of the running example (Figure 4).

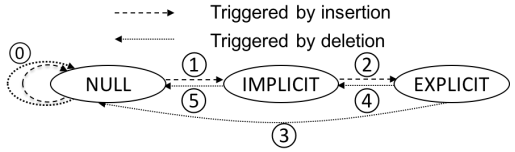


Figure 5: Edge transition diagram.

Note that we can also construct a DCG g_{DCG} for the initial data graph g_0 by using EL. For this purpose, building the DCG for g_0 is regarded as a special type of edge insertion (v_s^*, v_s) for every v_s in $V(g_0)$ where $L(u_s) \subseteq L(v_s)$ and v_s^* is an artificial vertex.

Transition 0 (from NULL to NULL).

Case 1: Suppose that an edge (v, v') is inserted into or deleted from g_{i-1} and fails to match any (u, u') in q. If the state of (v, u', v') in the DCG is NULL, then the state of (v, u', v') remains NULL.

Case 2: Suppose that an edge (v, v') is inserted into or deleted from g_{i-1} and matches (u, u') in q. If the state of (v, u', v') in the DCG is NULL, and v in the DCG has no incoming edge whose label is u, then the state of (v, u', v') remains NULL.

Transition 1 (from NULL to IMPLICIT).

Case 1: Suppose that an edge (v, v') is inserted into g_{i-1}. We check if, for each (u, u') in q such that v in the DCG has an incoming implicit or explicit edge whose edge label is u, (v, v') matches (u, u'). If so, transit the state of (v, u', v') in the DCG from NULL to IMPLICIT.

For example, when (v_0, v_1) is inserted into g_0 in Figure 4b, we transit the state of (v_0, u_1, v_1) in the DCG from NULL to IMPLICIT since the inserted edge matches (u_0, u_1) and v_0 has an incoming explicit edge with label u_0. Figure 4d shows the updated DCG.

Case 2: Suppose that the state of (v, u', v') in the DCG is transited from NULL to IMPLICIT. For each v'' in $Adj(v')$, if (v', v'') in g matches (u', u'') for some u'' in $Children(u')$, then transit the state of (v', u'', v'') in the DCG from NULL to IMPLICIT.

For example, since the state of (v_0, u_1, v_1) is transited from NULL to IMPLICIT in Figure 4d, we check if (v_1, v_4) matches (u_1, u_4). Since it succeeds to match, the state of (v_1, u_4, v_4) is transited from NULL to IMPLICIT. The resulting DCG becomes Figure 4e.

Transition 2 (from IMPLICIT to EXPLICIT).

Case 1: Suppose that the state of (v, u', v') in the DCG is transited from NULL to IMPLICIT. If u' is a leaf node in q' or v' has an outgoing explicit edge whose label is u'' for every u'' in $Children(u')$, then transit the state of (v, u', v') in the DCG from IMPLICIT to EXPLICIT.

For example, since the state of (v_1, u_4, v_4) is transited from NULL to IMPLICIT in Figure 4e, and u_4 is a leaf node in q, we transit its state to EXPLICIT as in Figure 4f.

Case 2: Suppose that the state of (v, u', v') in the DCG is transited from IMPLICIT to EXPLICIT. If v has an outgoing explicit edge in the DCG whose label is u'' for every u'' in $Children(P(u'))$, then transit the state of every IMPLICIT incoming edge of v in the DCG whose label is $P(u')$ from IMPLICIT to EXPLICIT.

For example, since the state of (v_1, u_4, v_4) is transited from IMPLICIT to EXPLICIT in Figure 4f, v_1 now has an outgoing explicit edge for each child of u_1 (i.e., edge whose label is u_4). Thus the state of (v_0, u_1, v_1) is transited to EXPLICIT as in Figure 4g. Here, the state of (v_s^*, u_0, v_0) is also transited from IMPLICIT to EXPLICIT as in Figure 4h since v_0 now has an outgoing explicit edge for each child of u_0 (i.e., edges whose labels are u_1, u_2, and u_3).

Transition 3 (from EXPLICIT to NULL).

Case 1: Suppose that an edge (v, v') is deleted from g. For each (u, u') in q such that v in the DCG has an incoming implicit or explicit edge whose edge label is u, if (v, v') matches (u, u') and the state of (v, u', v') in the DCG is EXPLICIT, then transit the state of (v, u', v') in the DCG from EXPLICIT to NULL.

Case 2: Suppose that the state of (v, u', v') in the DCG is transited from IMPLICIT or EXPLICIT to NULL and v' in the DCG no longer has any incoming edge whose label is u'. For each u'' in $Children(u')$, transit the state of every outgoing explicit edge of v' in the DCG whose label is u'' from EXPLICIT to NULL.

Transition 4 (from EXPLICIT to IMPLICIT).

Case 1: Suppose that the state of (v, u', v') in the DCG is transited from EXPLICIT to IMPLICIT or NULL. If v in the DCG no longer has any outgoing EXPLICIT edge whose label is u', then transit the state of every incoming EXPLICIT edge of v in the DCG whose label is $P(u')$ from EXPLICIT to IMPLICIT.

Transition 5 (from IMPLICIT to NULL).

Case 1: Suppose that an edge (v, v') is deleted from g. For each (u, u') in q such that v in the DCG has an incoming implicit or explicit edge whose edge label is u, if (v, v') matches (u, u') and the state of (v, u', v') in the DCG is IMPLICIT, then transit the state of $(v, u'. v')$ in the DCG from IMPLICIT to NULL.

Case 2: Suppose that the state of (v, u', v') in the DCG is transited from IMPLICIT or EXPLICIT to NULL, and that v' in the DCG no longer has any incoming edge whose label is u'. Then, for each u'' in $Children(u')$, transit the state of every outgoing implicit edge of v' in the DCG whose label is u'' from IMPLICIT to NULL.

4 TURBOFLUX ALGORITHMS

In this section, we present detailed algorithms for TurboFlux. Although EL in the edge transition model is a simple and formal algorithm to understand, a naive implementation of EL would lead to bad performance. While we traverse subgraphs of the DCG, we can

apply necessary transition rules only. This motivated us to develop an enhanced version of the maintenance algorithm for the DCG. Furthermore, since we need to report positive/negative matches while maintaining intermediate results, we present several functions including two major functions InsertEdgeAndEval and DeleteEdgeAndEval, where transitions rules are selectively applied.

4.1 Overview of Solution

This section introduces the overall procedure of TurboFlux. The main algorithm of TurboFlux (Algorithm 2) has three major parts; 1) obtaining a query tree q' and a starting query vertex u_s from q; 2) building an initial DCG g_{DCG} for g_0, obtaining a matching order mo, and reporting solutions for g_0; 3) handling continuous subgraph matching for each update operation.

TurboFlux first selects a u_s (Line 2) and transforms q into q' (Line 3). In order to minimize the number of matching data vertices for u_s, ChooseStartQVertex first selects a query edge (u, u') which has the smallest number of matching data edges. Between u and u', ChooseStartQVertex chooses a query vertex which has a smaller number of matching data vertices. Finally, if there is a tie, ChooseStartQVertex chooses a query vertex having a larger degree. In the running example, we choose u_0 as u_s since (u_0, u_1) has the smallest number of matching data edges, and u_0 has a larger degree than u_1. In TransformToTree, the initial tree has the starting query vertex only, and then we enlarge the tree by repeatedly adding a query edge one at a time. In order to find the most selective tree, we use a greedy approach that chooses an edge that is connected with already matched query vertices and minimizes the estimated intermediate result size. When the tree contains all query vertices, non-selected edges are used as non-tree edges.

Next, we build a DCG, g_{DCG} for the initial data graph g_0, which is constructed by traversing from every v_s in $V(g)$ such that $L(u_s) \subseteq L(v_s)$ (Lines 4-5). Although we call BuildDCG for every v_s, since there is a hypothetical edge from v_s^* to every v_s, only one DCG is constructed here.

After constructing the DCG, TurboFlux determines a good matching order as a sequence of query vertices in q' in order to minimize the number of recursive calls for subgraph matching (Line 6). The related algorithm is explained as follows. Given a matching order $<u_{o_1}, u_{o_2}, \cdots, u_{o_n}>$, T_i denotes the query tree formed by $u_{o_1}, u_{o_2}, ..., u_{o_i}$. Then, given a matching order, the number of recursive calls using this order is computed as $\sum_{i=1}^{n} c(T_i)$ where $c(T_i)$ is the total number of partial solutions for T_i. Thus, our goal is to find a matching order which minimizes $\sum_{i=1}^{n} c(T_i)$. Since enumerating all possible subtrees could be very expensive for large query graphs, we use a greedy strategy. We first start with the query tree q' and shrink it by removing one edge at a time. At each edge deletion, we select an edge that minimizes the number of partial solutions for the shrunk tree. Note that, since we have built the DCG, we can accurately estimate $c(T_i)$ based on the number of explicit data paths (ones having explicit edges only) for each query path. For example, consider q in Figure 1a and the g_{DCG} in Figure 2e. We find a matching order $<u_0, u_1, u_3, u_4, u_2>$, since there exist only two partial solutions when we match u_0, u_1, u_3, and u_4.

After the matching order is selected, in order to report the solutions for the initial data graph, we execute SubgraphSearch for each v_s that has an incoming, explicit edge whose label is u_s (Lines 7-11). Here, we follow explicit edges only since implicit edges indicate that their subtrees do not match the corresponding query subtrees yet. In the running example, we do not execute SubgraphSearch since all v_ss, v_0 and v_1, do not have an incoming explicit edge whose label is u_s. Thus, we do not report any initial matches.

Algorithm 2. TurboFlux($q, g, \Delta g$)

```
   Input: A query graph q, a data graph g, and a graph update stream Δg
1: g_DCG ← CreateEmptyDCG();
2: u_s ← ChooseStartQVertex(q, g);
3: q' ← TransformToTree(q, u_s);
   /* build a g_DCG                                                  */
4: foreach (v_s in V(g) such that L(u_s) ⊆ L(v_s)) do
5:     BuildDCG (u_s, (v_s^*, v_s));
6: mo ← DetermineMatchingOrder(q, g_DCG);
   /* report solutions for the initial data graph g_0                */
7: foreach (v_s in V(g) such that L(u_s) ⊆ L(v_s)) do
8:     if (g_DCG.GetState(v_s^*, u_s, v_s) = E) then
           /* m is a graph homomorphism mapping                      */
9:         m(u_s) ← v_s;
           /* report all complete solutions from g_0                 */
           /* + indicates positive solutions                         */
10:        SubgraphSearch (mo, 1, NIL, m, +);
11:        m(u_s) ← NIL;
12: foreach (Δo in Δg) do
13:    switch (Δo.op) do
14:    case (EDGE_INSERTION) do
15:        InsertEdgeToDataGraph(g, Δo.v, Δo.v');
16:        InsertEdgeAndEval(mo, Δo.v, Δo.v');
17:    case (EDGE_DELETION) do
18:        DeleteEdgeAndEval(mo, Δo.v, Δo.v');
19:        DeleteEdgeFromDataGraph(g, Δo.v, Δo.v');
20:    AdjustMatchingOrder (mo, q, g_DCG);
```

Next, we perform incremental subgraph matching for each update operation. Depending on the update type, we update g_{DCG} and incrementally evaluate subgraph matching (Lines 12-20). For example, on edge insertion, we first insert the edge into g (Line 15) and incrementally evaluate the subgraph matching (Line 16), which reports the positive matches. On edge deletion, we find negative matches through incremental subgraph matching by calling DeleteEdgeAndEval (Line 18) and then delete the edge from g (Line 19). Here, the operation order is important.

For example, consider Figure 4b, where an edge (v_0, v_1) is inserted. We first insert the edge into g_0 and then call InsertEdgeAndEval. After all possible transitions are applied, the DCG becomes as Figure 4h. Since the states of (v_0, u_1, v_1), (v_1, u_4, v_4), and (v_s^*, u_0, v_0) become EXPLICIT, we report a positive match, $\{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_6)\}$.

We invoke AdjustMatchingOrder to check whether the current matching order should be adjusted (Line 20). Since we maintain DCG incrementally, we can estimate the number of explicit data paths added/deleted for each query path. If there is a significant change in those statistics, we recompute the matching order based on them.

4.2 Building a DCG for g_0

In Line 5 of Algorithm 2, in order to build the initial DCG, we call BuildDCG for every v_s matching the starting query vertex u_s.

BuildDCG (Algorithm 3) builds a part of DCG by performing a depth-first traversal from the data vertex v. First, BuildDCG transits the state of the edge (v, u', v') from NULL to IMPLICIT by Case 1 of Transition 1 (Line 1). Here, $u_s \rightarrow P(u').u'$ matches $v_s \rightarrow v.v'$ for some starting data vertex v_s. We check if v' has an incoming implicit or explicit edge whose label is u' except the one transited in Line 1. If so, we avoid building the DCG for the subtrees of v' since the DCG for these subtrees has already built (Line 2). Otherwise, we try to match the subtrees of u' (Lines 3-5) by calling BuildDCG recursively. Then, the states of edges in the subtrees can be transited from NULL to IMPLICIT by Case 2 of Transition 1. After recursion, we call MatchAllChildren (Algorithm 4) to check if every subtree of u' matches the corresponding subtree of v', then transit the state of the edge (v, u', v') from IMPLICIT to EXPLICIT by Case 1 or 2 of Transition 2 (Lines 6-7). In our implementation, MatchAllChildren can be done in $O(1)$ by using a bitmap for each data vertex v where the i-th bit indicates whether v has any outgoing explicit edge whose label is u_i.

Algorithm 3. BuildDCG $(u', (v, v'))$

```
   Input: A query vertex u' and a data edge (v, v')
   /* Case 1 (if called non-recursively) or 2 (if called
      recursively) of Transition 1                          */
1: g_DCG.MakeTransition((v, u', v'), I);
   /* check-and-avoid:
      We check if there exist other implicit or explicit
      incoming edges except (v, u', v'). If so, we avoid
      building the DCG for the subtrees of v'.              */
2: if (| g_DCG.GetImplAndExplEdges(v', u', in) |= 1) then
3:     foreach (child query vertex u'_c of u' in q') do
4:         foreach (v'_c where (v', v'_c) matches (u', u'_c)) do
5:             BuildDCG(u'_c, (v', v'_c));
   /* Case 1 or 2 of Transition 2                           */
6: if (MatchAllChildren(v', u')) then
7:     g_DCG.MakeTransition((v, u', v'), E);
```

Algorithm 4. MatchAllChildren (v, u)

```
   Input: A data vertex v and a query vertex u
1: foreach (child query vertex u' of u in q') do
2:     if (| g_DCG.GetExplEdges(v, u', out) |= 0) then
3:         return false;
4: return true;
```

For example, when q and g_0 in Figure 4 are given, BuildDCG $(u', (v, v'))$ for v_0 and v_1 is called. (Line 5 of Algorithm 2). Here, u'=u_0 and v=v_s^*. First, when v'=v_0, the state of the edge (v_s^*, u_0, v_0) is transited from NULL to IMPLICIT. Then the recursive calls to BuildDCG transit the states of the edges (v_0, u_2, v_2) and (v_2, u_5, v_6) from NULL to IMPLICIT. At the end of each recursive call, their states are transited from IMPLICIT to EXPLICIT. Similarly, the state of (v_0, u_3, v_3) is transited from NULL to IMPLICIT and then to EXPLICIT. Next, after calling BuildDCG with $v' = v_1$, we obtain the g_{DCG_0} as in Figure 4c.

Lemma 4.1. *The time complexity of BuildDCG is* $O(|E(g)|*|V(q)|)$.

Proof. In the worst case, BuildDCG is called for every query vertex u' and every data vertex v'. Given u' and v', the time complexity for Lines 3-4 of Algorithm 3 is $O(deg(v') * |Children(u')|)$. Note that the time complexity for Lines 6-7 is $O(|Children(u')|)$. Thus, the time complexity of BuildDCG is $O(\sum_{u' \in q} \sum_{v' \in g} (deg(v') * |Children(u')|)) = O(|E(g)| * |V(q)|)$. □

4.3 Edge Insertion

Now, we explain InsertEdgeAndEval (Algorithm 5), which is invoked for each edge insertion (v, v'). The main idea of InsertEdgeAndEval is explained as follows: We try to match (v, v') with tree query edges and build the DCG downwards for the subtrees of v'. We then build the DCG upwards until we reach any of the starting data vertices and execute subgraph matching (Lines 1-10). We then try to match (v, v') with non-tree query edges, traverse upwards without building the DCG until we reach any of the starting data vertices, and execute subgraph matching (Lines 11-18). Note that when the edge inserted matches non-tree edges, it does not contribute to modify the DCG. For this purpose, BuildUpwardsAndEval (Lines 9 and 17) has a flag as the last parameter to indicate that it tries to either build upwards or traverse upwards.

Algorithm 5. InsertEdgeAndEval (mo, v, v')

```
   Input: Matching order mo and an inserted edge (v, v')
   /* find us satisfying u_s->u matches v_s->v              */
1: U <- {u| | g_DCG.GetImplAndExplEdges(v, u, in) |> 0};
   /* if U = ∅, we do not update g_DCG by Case 2 of
      Transition 0                                          */
2: foreach (u ∈ U) do
3:     foreach (child query vertex u' of u in q') do
           /* if (u, u') does not match (v, v'), we do not
              update g_DCG by Case 1 of Transition 0        */
4:         if ((u, u') matches (v, v')) then
5:             BuildDCG(u', (v, v'));
6:             if (g_DCG.GetState(v, u', v') = E) then
7:                 if (MatchAllChildren(v, u)) then
8:                     m(u') <- v';
9:                     BuildUpwardsAndEval(mo, u, v, (u, u'), m,
                         true);
10:                    m(u') <- NIL;
11: U <- {u| | g_DCG.GetImplAndExplEdges(v, u, in) |> 0};
12: foreach (u ∈ U) do
       /* A non-tree edge inserted does not modify g_DCG   */
13:    foreach (non-tree edge (u, u') of q that matches (v, v')) do
           /* check if u_s->u' matches v_s->v'              */
14:        if (| g_DCG.GetImplAndExplEdges(v', u', in) |> 0) then
15:            if (MatchAllChildren(v, u) and
                 MatchAllChildren(v', u')) then
16:                m(u') <- v';
17:                BuildUpwardsAndEval(mo, u, v, (u, u'), m,
                     false);
18:            m(u') <- NIL;
```

In order to identify every tree edge (u, u') that matches (v, v'), we first obtain the set of query vertices $\{u\}$ (=U) such that $u_s \rightarrow u$ matches $v_s \rightarrow v$ for some starting data vertex v_s. To do this, we obtain the labels $\{u\}$ of the incoming, implicit/explicit edges of v (Line 1). Then, for each child query vertex u' of u, we check if $L(u) \subseteq L(v)$ and $L(u') \subseteq L(v')$ (Lines 2-4). For each query edge (u, u') matched, we execute BuildDCG to traverse the subtrees of v' and make transitions of relevant edges in the DCG (Line 5). After the traversal, if the state of the edge (v, u', v') is EXPLICIT (Line 6), we first check if every subtree of u matches the corresponding subtree of v (Line 7). If so, then we start to map u' to v', call BuildUpwardsAndEval (Algorithm 6) to build the DCG upwards

(i.e., traversing upwards to starting data vertices, $\{v_s\}$) and execute subgraph matching whenever we reach any of $\{v_s\}$ (Lines 8-10).

For example, consider Figure 4c where InsertEdgeAndEval is called for (v_0, v_1). Since v_0 has an incoming implicit edge with label u_0, and (u_0, u_1) matches (v_0, v_1), BuildDCG $(u_1, (v_0, v_1))$ is called. Then the DCG changes from Figure 4c to Figure 4g. Since the state of (v_0, u_1, v_1) now becomes EXPLICIT, we map u_1 to v_1 and call BuildUpwardsAndEval in order to change the states of the incoming edges to v_0 to EXPLICIT and execute subgraph matching from v_0 (Line 9).

The inserted edge can match non-tree edges of the query graph. First, we again obtain the set of query vertices $\{u\}$ that match v (Line 11), since the DCG could have been changed from Lines 5 and 9. Then, we need to check whether each non-tree edge (u, u') matches (v, v'). Next, for each (u, u') matched, we check whether both every subtree of u and every subtree of u' match the corresponding subtree of v and the corresponding subtree of v', respectively (Line 15). Here, we call BuildUpwardsAndEval with the transition flag (the last parameter) set to $false$ (Line 17), meaning that we make no transition in the DCG during the call to BuildUpwardsAndEval.

BuildUpwardsAndEval traverses the DCG upwards and makes transitions if necessary. Since it is called only when every subtree of u matches the corresponding subtree of v, we can safely map u to v (Line 1). Then, for each incoming implicit or explicit edge (v_p, u, v) of v (Line 2), if the transition flag f_t is set to true, and the state of the edge (v_p, u, v) is IMPLICIT, we transit its state to EXPLICIT by Case 2 of Transition 2 (Lines 3-5). When it reaches the starting data vertex (i.e., u=u_s), we execute SubgraphSearch (Algorithm 7) in Line 7. Otherwise, we traverse upwards through the edge (v_p, u, v) by calling BuildUpwardsAndEval (Lines 9-11).

Algorithm 6. BuildUpwardsAndEval (mo, u, v, e_q, m, f_t)

```
   Input: Matching order mo, a query vertex u, a data vertex v,
          a query edge e_q, mapping m, and a transition flag f_t
1:  m(u) ← v;
2:  foreach ((v_p, u, v) in g_DCG.GetImplAndExplEdges(v, u, in)) do
3:      if (f_t = true) then
            /* Case 2 of Transition 2                          */
4:          if (g_DCG.GetState(v_p, u, v) = I) then
5:              g_DCG.MakeTransition((v_p, u, v), E);
6:      if (u = u_s in q') then
7:          SubgraphSearch(mo, 1, e_q, m, +);
8:      else
9:          u_p ← P(u);
10:         if (MatchAllChildren(v_p, u_p)) then
11:             BuildUpwardsAndEval(mo, u_p, v_p, e_q, m, f_t);
12: m(u) ← NIL;
```

For example, consider a DCG in Figure 4g. Suppose that BuildUpwardsAndEval is called with $u = u_0$, and $v = v_0$. Then, we map u_0 to v_0 (Line 1) and transit the state of (v_s^*, u_0, v_0) from IMPLICIT to EXPLICIT (Line 5). Here, the DCG changes from Figure 4g to Figure 4h. Then, since u_0 is the starting query vertex (Line 7), we execute SubgraphSearch, which finds positive matches including (u_0, v_0) and (u_1, v_1) since we mapped u_0 and u_1 to v_0 and v_1 respectively.

In order to implement SubgraphSearch, we can use *any* existing subgraph matching algorithm with two modifications: 1) data edges inserted or deleted must map to query vertices for reporting positive/negative matches; 2) any duplicate results must be avoided, since an inserted data edge can match more than one query edge in a complete solution under graph homomorphism. In our implementation, we use a modified version of TurboHOM++ [17] instead of using worst-case optimal subgraph matching algorithms such as [2], since its performance is significantly faster than those worst-case optimal algorithms for real-world labeled graphs (refer to experimental results in [17] and [3]). However, in order to achieve the worst-case optimality, we can modify the existing wort-case optimal algorithm so that it can use the DCG rather than the data graph. For ease of explanation, we explain how the following standard backtracking based graph homomorphism algorithm is modified.

Algorithm 7. SubgraphSearch (mo, d_c, e_q, m, p)

```
   Input: Matching order mo, current matching order number d_c, a
          query edge e_q, mapping m, and positiveness p
1:  u ← mo[d_c];
2:  if (u = u_s in q') then
3:      v_p ← v_s^*;
4:  else
5:      v_p ← m(P(u));
6:  if (m(u) ≠ NIL) then
        /* for non-tree case in edge insertion/deletion    */
7:      if (g_DCG.GetState(v_p, u, m(u)) ≠ E) then
8:          return;
9:      if (IsJoinable(u, m(u), e_q, m, p)) then
10:         if (d_c <| q' |) then
11:             SubgraphSearch(mo, d_c + 1, e_q, m, p);
12:         else
13:             report < p, m >;
14: else
15:     foreach (v such that g_DCG.GetState(v_p, u, v) = E) do
16:         if (IsJoinable(u, v, e_q, m, p)) then
17:             m(u) ← v;
18:             if (d_c <| q' |) then
19:                 SubgraphSearch(mo, d_c + 1, e_q, m, p);
20:             else
21:                 report < p, m >;
22:             m(u) ← NIL;
    Function IsJoinable (u, v, (u_a, u_b), m, p)
        Input: A query vertex u, a data vertex v, a query edge (u_a, u_b),
               mapping m, and positiveness p
1:      foreach (non-tree edge (u, u') in q) do
2:          if (m(u') = NIL) then
3:              continue;
4:          if ((v, m(u')) match (u, u')) then
5:              if ((u_a, u_b) ≠ NIL and (v, m(u')) = (m(u_a), m(u_b)))
                then
6:                  if (p = +) then
7:                      if ((u, u') > (u_a, u_b)) then
8:                          return false;
9:                  else
10:                     if ((u, u') < (u_a, u_b)) then
11:                         return false;
12:         else
13:             return false;
14:     return true;
```

SubgraphSearch (Algorithm 7) outlines the graph homomorphism search to generate complete solutions. Since a parent query

vertex precedes its children in the matching order, we can always find v_p in Line 5. If u has been mapped to a data vertex (Line 6), we first check whether the state of $(v_p, u, m(u))$ is EXPLICIT in the g_{DCG} (Lines 7-8). In the case of a non-tree edge (u_a, u_b), we map u_a and u_b to data vertices and traverse upwards from only one vertex u_a. Thus, we need to check whether the state of $(m(P(u_b)), u_b, m(u_b))$ is EXPLICIT. Next, by calling IsJoinable (in Algorithm 7), we check if the non-tree edges between u and already matched query vertices of q have corresponding edges between $m(u)$ and already matched data vertices of g (Line 9). If IsJoinable succeeds, we call SubgraphSearch recursively, or report a mapping once all query vertices are mapped (Lines 10-13). If u has no data vertex mapped (Line 14), for each data vertex v in the outgoing explicit edges of v_p whose label is u (Line 15), we call IsJoinable for v (Line 16). If IsJoinable succeeds, we map u to v (Line 17) and then execute SubgraphSearch or report the mapping (Lines 18-21). To avoid generating duplicate solutions under graph homomorphism, TurboFlux uses the total order (<) for all query edges. In Line 7 of IsJoinable, we enforce this total order to prevent duplication. Unlike edge insertion which incrementally inserts explicit edges in the DCG while we match inserted data edge with query edges, edge deletion incrementally deletes explicit edges in the DCG. Thus, we use the reverse order of '<' in Line 10 of IsJoinable.

Now, we explain the trade-off of TurboFlux on the storage and subgraph matching costs compared to SJ-Tree. While SJ-Tree generates all possible partial solutions for each edge insertion regardless of whether they can contribute to complete solutions or not, TurboFlux maintains minimal intermediate results in the DCG. Clearly, according to Lemma 4.1, the cost for building the DCG is much cheaper than generating all possible partial solutions by SJ-Tree. Even if there is a relative overhead in traversing the DCG during subgraph matching compared to SJ-Tree, which reads partial solutions materialized during join, the benefit using the DCG by TurboFlux significantly outweighs the benefit of reading materialized partial solutions by SJ-Tree, as we show in experiments in Section 5. Furthermore, since the DCG is typically much smaller than the partial solutions stored in SJ-Tree, and a large portion of the DCG fits in the CPU cache, graph exploration can be done very fast, while SJ-Tree needs to read an enormous number of partial solutions from main memory, not from the CPU cache.

Algorithms for edge deletion are very similar to those for edge insertion except that they use the transitions 3∼5, instead of transitions 1 and 2. The algorithms and detailed explanation are included in Appendix A.

5 EXPERIMENTS

In this section, we evaluate the performance of TurboFlux against the state-of-the-art continuous subgraph matching methods, SJ-Tree [7], and Graphflow [16]. Comparison with IncIsoMat is in Appendix B.3. Note that IncIsoMat is extremely slower than the others; TurboFlux outperforms IncIsoMat by up to 2,214,086 times. The source code of SJ-Tree was obtained from its authors. The source code of Graphflow was downloaded from github* on Sep 24, 2017. The key goals of our experiments are as follows.

- TurboFlux consistently and significantly outperforms the state-of-the-art methods for varying querysets and datasets.
- TurboFlux has significantly smaller intermediate result sizes than SJ-Tree.
- For varying insertion rate and dataset size, TurboFlux significantly outperforms the state-of-the-art methods.

5.1 Experimental Setup

Datasets/Queries. We use two datasets referred as LSBench and Netflow as used in [7]. The first dataset is generated by the Linked Stream Benchmark data generator with the number of users as a scale factor. We generate three different sizes of datasets with 0.1, 1, and 10 million users, and use the first one as a default. This dataset consists of a initial graph g_0 and a graph update stream Δg. g_0 contains 20,988,361 triples while Δg consists of insertions of 2,332,065 triples. Netflow is a real dataset which contains anonymized passive traffic traces monitored from high-speed internet backbone links. Netflow consists of 18,520,759 triples, where g_0 contains 16,668,683 triples, and Δg consists of insertions of 1,852,076 triples.

We generate a set of 100 queries for each size and query type. We use two types of queries: tree and graph (having cycles). In order to generate queries having a wide range of selectivity values, we randomly choose an edge label regardless of the edge distribution. For tree queries, we generate a set of 100 queries of size 12 by randomly traversing schema graphs. Here, the size of a query is defined as the number of triples. For tree queries, we first generate queries of size 12 and construct smaller queries by randomly removing edges while the smaller queries are still connected. For graph queries, we first generate 33, 33, and 34 cyclic query graphs of size 3, 4, and 5 (i.e., triangle, square, and pentagon), respectively. From these cyclic graphs, we construct larger query graphs up to size 12 by randomly adding triples. Note that there are only 21 different queries of size 3 for graph queries. Thus, we generate the graph queries of size from 6 to 12 in 3 increments. Note also that, we excluded queries that have no positive matches for the entire insertion stream. The distribution of query selectivity is included in Appendix C.

For all these RDF datasets, we transform one triple into one labeled edge and do not use any embedding technique that uses triples as vertex attributes as in [9, 17] for fair comparison.

Running Environments. We conducted experiments on a Linux machine having four Intel E5-4640 CPUs, 1.5TB RAM, and 6TB (1.5TB * 4) SAS Disks. We used a single thread for all experiments.

Measure. We use two measures, the average elapsed time for incremental subgraph matching (denoted by cost($M(\Delta g, q)$)) and the size of intermediate results. Here, the cost($M(\Delta g, q)$) of TurboFlux and SJ-Tree are calculated as the difference between the costs (in terms of the elapsed times) for processing the graph update stream with and without continuous subgraph matching. Thus, the cost for updating the data graph is excluded for fair comparison. For the cost($M(\Delta g, q)$) of Graphflow, we measure time for join processing only and exclude the other costs, such as the data graph sorting cost. Also, we insert edges in batches of 100K for Graphflow, as in [16], since it is too slow when we update one edge at a time.

5.2 Experimental Results

The performance of TurboFlux was evaluated using various parameters such as query size, query type, insertion rate, dataset size, deletion rate, and matching semantics. Unless specified otherwise, values in boldface in Table 1 are used as default parameters in the experiments. We set a 2-hour timeout for each query.

*https://github.com/graphflow/graphflow

The experiments for varying deletion rate and subgraph isomorphism semantics are included in Appendix B.2 and B.1, respectively.

Table 1: Parameters used in the experiments.

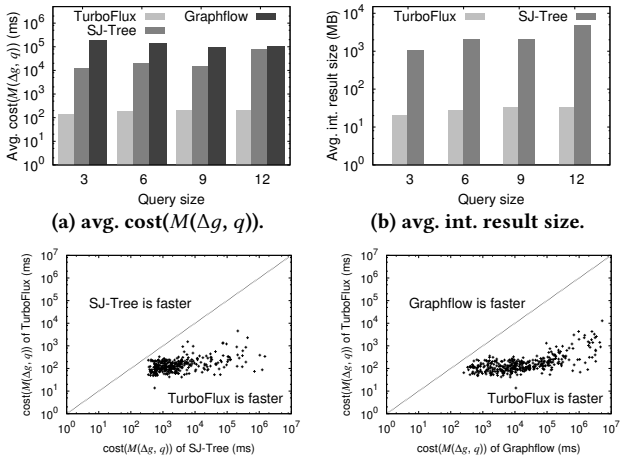
Parameters	Values Used
Datasets	**LSBench**, Netflow
Query size	3, **6**, 9, 12 (tree queries) **6**, 9, 12 (graph queries)
Insertion rate	2, 4, **6**, 8, 10
Dataset size	**0.1** , 1 , 10 million users (LSBench)
Deletion rate	2, 4, 6, 8, 10
Matching semantics	**graph homomorphism**, subgraph isomorphism

5.2.1 *Experiments using LSBench.*

Varying the query size. Figure 6 shows the performance results for tree queries in LSBench. Here, we vary the query size from 3 to 12. Note that, the matching cost of queries does not always increase as the query size increases [14, 18]. Figure 6a shows the average elapsed time for performing continuous subgraph matching. Since SJ-Tree reached the timeout for 2, 7, 12, and 19 queries and Graphflow reached the timeout for 1, 2, 3, and 2 queries of sizes 3, 6, 9, and 12, respectively, we report results for the remaining queries. TurboFlux significantly outperforms both SJ-Tree and Graphflow regardless of query size. Specifically, TurboFlux outperforms SJ-Tree by 77.30 ~ 379.22 times and Graphflow by 515.01 ~ 1,275.68 times. Figure 6b shows the average size of intermediate results. SJ-Tree shows the notorious intermediate result size problem. Specifically, the average size of intermediate results of SJ-Tree is significantly larger than that of TurboFlux by up to 142.34 times.

Figure 6c is a scatter plot of the cost($M(\Delta g, q)$) on TurboFlux and SJ-Tree for all tree queries except the 40 timeout queries in SJ-Tree. It shows that TurboFlux consistently outperforms SJ-Tree for all of those queries. Among them, we analyze two queries Q_{87} of size 12 and Q_{71} of size 9 which have the largest and smallest performance gap between TurboFlux and SJ-Tree, respectively. For Q_{87}, SJ-Tree is 14,192 times slower than TurboFlux. This is because Q_{87} does not have selective edges, and some query vertices are connected to other vertices through one-to-many relationships. Thus, SJ-Tree generates a large size of partial solutions (13.1GB). The join cost of SJ-Tree is also very expensive due to these enormous intermediate results. However, Q_{71} contains highly selective edges, which leads to a small number of partial/total solutions. Thus, both TurboFlux and SJ-Tree have a small number of intermediate results. Nevertheless, due to efficiency in graph exploration and the check-and-avoid strategy, TurboFlux still outperforms SJ-Tree for Q_{71} by 1.50 times.

Figure 6d is a scatter plot between TurboFlux and Graphflow for all tree queries except the eight timeout queries in Graphflow. It shows that TurboFlux consistently outperforms Graphflow for all of those queries. Among them, we analyze two queries, Q_{90} and Q_{74}, of size 9 which have the largest and smallest performance gap between TurboFlux and Graphflow, respectively. For Q_{90}, Graphflow is 20,844 times slower than TurboFlux. This is because Graphflow generates a much larger number of partial solutions than TurboFlux. Q_{90} has a relatively large number of matching results (1.41 times the average) but a small size of DCG (0.38 times the average). It means that TurboFlux efficiently reduces the search space using the DCG. However, Q_{74} has a relatively small number of results (0.0001 times the average) and a large size of DCG (1.67 times the average). Nevertheless, TurboFlux outperforms Graphflow for Q_{74} by 2.44 times.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

(c) cost($M(\Delta g, q)$) on TurboFlux and SJ-Tree. (d) cost($M(\Delta g, q)$) on TurboFlux and Graphflow.

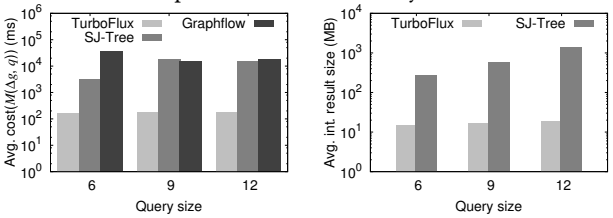
Figure 6: Tree Queries in LSBench.

Figure 7 shows the performance results for graph queries in LSBench. Here, SJ-Tree reached the timeout for 19, 18, and 18 queries and Graphflow reached the timeout for 2, 4, and 6 queries of sizes 6, 9, and 12, respectively. Note that, although the performance gap for graph queries seems to be slightly smaller than that for tree queries, the number of timeout graph queries is larger than the number of timeout tree queries in SJ-Tree and Graphflow. Even if such queries are excluded, TurboFlux consistently outperforms both SJ-Tree and Graphflow. Specifically, TurboFlux outperforms SJ-Tree by 21.61 ~ 114.50 times and Graphflow by 90.73 ~ 240.07. Figure 7b shows the average size of intermediate results. Specifically, the average size of intermediate results of SJ-Tree is significantly larger than that of TurboFlux by up to 76.10 times. Note that Graphflow outperforms SJ-Tree for the query set of size 9 due to one query, Q_{89}. If we exclude Q_{89}, SJ-Tree outperforms Graphflow by 4.55 times for the size 9. One of the subgraphs in Q_{89} leads to significant amount of intermediate result size (20.9 GB). SJ-Tree takes about 1,215 seconds due to the enormous size of intermediate results, while Graphflow takes about 4.4 seconds.

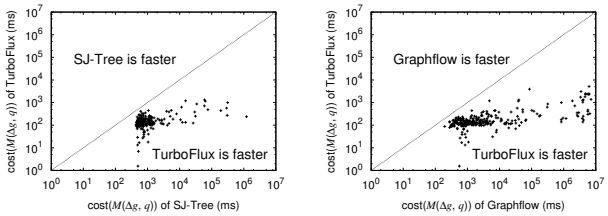
Figure 7c is a scatter plot between TurboFlux and SJ-Tree for all graph queries except the 55 timeout queries in SJ-Tree. Among them, we analyze two queries, Q_{89} and Q_{69} of size 9 showing the largest and the smallest performance gap. Similarly to the two tree queries above, SJ-Tree generates an excessive amount of intermediate results (30.6GB) for Q_{89}, and thus incurs an explosive join cost. However, for Q_{69}, both have much smaller intermediate result sizes. Nevertheless, TurboFlux still outperforms SJ-Tree by 1.48 times.

Figure 7d is a scatter plot between TurboFlux and Graphflow for all graph queries except the 12 timeout queries in Graphflow. Among them, we analyze two queries, Q_{62} of size 12 and Q_{46} of size 6 showing the largest and smallest performance gap. Q_{62} of size 12 contains a cycle that has a huge amount of matching in the data graph. However, the number of complete solutions is relatively small, since the query contains a highly selective subgraph as well. Graphflow generates a large number of partial solutions due to

the cycle, while TurboFlux efficiently prunes them using the DCG. Thus, TurboFlux outperforms Graphflow for Q_{62} of size 12 by 28,380 times. However, Q_{46} of size 6 has a very small number of matching for both the cycle and the entire query. Nevertheless, TurboFlux still outperforms Graphflow by 1.96 times.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

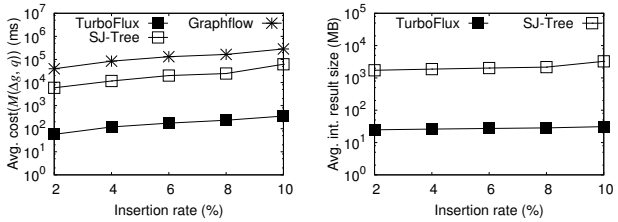


(c) cost($M(\Delta g, q)$) on TurboFlux and SJ-Tree. (d) cost($M(\Delta g, q)$) on TurboFlux and Graphflow.

Figure 7: Graph Queries in LSBench.

Since SJ-Tree shows poor performance due to the notorious intermediate result size problem, we apply two graph compression techniques [14, 24] to SJ-Tree to reduce its intermediate result size. The experimental results are included in Appendix B.5.

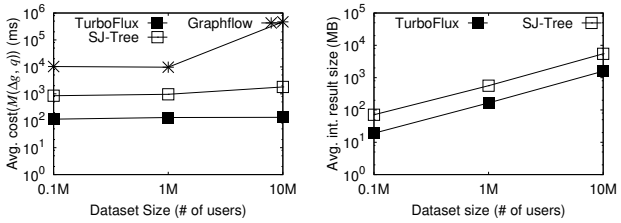
Varying the insertion rate. Figure 8 shows the performance results using LSBench for varying insertion rates. Here, we used tree queries of size 6 and varied the insertion rate from 2% to 10% in 2% increments with respect to the number of triples in the graph update stream. Note that 7 queries that timed out in SJ-Tree and 2 queries that timed out in Graphflow were excluded. TurboFlux has consistently better performance than both SJ-Tree and Graphflow. Specifically, TurboFlux outperforms SJ-Tree by up to 174.87 times and Graphflow by up to 804.98 times at insertion rate 10%. TurboFlux also outperforms SJ-Tree in terms of the average size of intermediate results by up to 105.15 times as shown in Figure 8b.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

Figure 8: Varying insertion rate.

Varying dataset size. Figure 9 shows the performance results using LSBench for varying dataset size. Here, we fixed the size of the graph update stream and varied the size of the initial data graph from 20,988,361 triples (0,1M users) to 2,100,348,700 triples (1M users). Note that, the performance for varying the size of the graph update stream is shown in Figure 8. Since SJ-Tree and Graphflow reached timeout for 48 and 4 queries, respectively, those queries are excluded. TurboFlux consistently outperforms both SJ-Tree and Graphflow regardless of the dataset size. Specifically, TurboFlux outperforms SJ-Tree by up to 13.47 times, and Graphflow by up to 3,510 times. The cost($M(\Delta g, q)$) of Graphflow in the 10M users dataset is much larger than that of the other sizes of datasets. This is because for some queries, Graphflow generates a much larger size of intermediate results in the 10M users dataset. For one of those queries, Graphflow generates 2, 2, and 286GB of intermediate join results in the datasets of sizes 0.1M, 1M, and 10M users, respectively. However, TurboFlux and SJ-Tree show similar performance for all sizes of datasets, since they exploit the intermediate results that are incrementally maintained. Figure 9b shows similar scalability of intermediate result sizes for TurboFlux and SJ-Tree. However, we exclude the results of timeout queries in SJ-Tree, and the actual difference is much larger, as in Figure 6b.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

Figure 9: Varying dataset size.

5.2.2 *Experiments using Netflow.*

Since SJ-Tree and Graphflow did not finish the experiments using Netflow in a reasonable time, we take a query with the minimum cost($M(\Delta g, q)$) on TurboFlux at each query size and perform the experiments with these queries. For these queries, TurboFlux outperforms SJ-Tree and Graphflow *at least* 45,886 and 69,221 times on average, respectively. Due to space limitations, details on the experimental results are included in Appendix B.4.

6 CONCLUSION

In this paper, we presented a fast, continuous subgraph matching system called TurboFlux in dynamic graph databases. We showed that TurboFlux solved the problems of existing methods and efficiently processed continuous subgraph matching for each update operation.

We first proposed the novel notion of a data-centric graph, which is an efficiently updatable graph for storing partial solutions. We then proposed the edge transition model, which efficiently identifies which update operation can affect the current partial solutions and/or contribute to generating positive/negative matches. We next presented the detailed algorithms of TurboFlux under the edge transition model and explained how the recomputation of subgraph matching for each update operation can be minimized.

Extensive experiments showed that TurboFlux outperformed existing competitors by up to orders of magnitude. Overall, we believe our continuous subgraph matching solution provides comprehensive insight and a substantial framework for future research.

7 ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1A2B3007116). Oracle provided a gift as a donation in support of our academic research.

REFERENCES

[1] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. 2017. Incremental Frequent Subgraph Mining on Large Evolving Graphs. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2710–2723.

[2] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 431–446.

[3] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. Old techniques for new join algorithms: A case study in RDF processing. In *Data Engineering Workshops (ICDEW), 2016 IEEE 32nd International Conference on*. IEEE, 97–102.

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, Vol. 24. ACM, 1–10.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. ACM, 1199–1214.

[6] Sutanay Choudhury, Lawrence Holder, George Chin, Abhik Ray, Sherman Beus, and John Feo. 2013. StreamWorks: a system for dynamic graph search. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1101–1104.

[7] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*. 157–168. https://doi.org/10.5441/002/edbt.2015.15

[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE PAMI* 26, 10 (2004), 1367–1372.

[9] Roberto De Virgilio. 2017. Smart RDF Data storage in Graph Databases. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 872–881.

[10] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 925–936.

[11] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 18.

[12] Wenfei Fan, Xin Wang, and Yinghui Wu. 2014. Querying big graphs within bounded resources. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 301–312.

[13] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. 2016. Toward continuous pattern detection over evolving large graph with snapshot isolation. *The VLDB Journal — The International Journal on Very Large Data Bases* 25, 2 (2016), 269–290.

[14] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 337–348.

[15] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*. 405–418.

[16] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1695–1698.

[17] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1238–1249.

[18] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2013, http://www-db.knu.ac.kr/vldb13.pdf. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB* 6, 2 (2013, http://www-db.knu.ac.kr/vldb13.pdf).

[19] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1307–1318.

[20] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2014. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)* 39, 1 (2014), 4.

[21] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.

[22] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.

[23] Andrea Pugliese, Matthias Bröcheler, VS Subrahmanian, and Michael Ovelgönne. 2014. Efficient multiview maintenance under insertion in huge social networks. *ACM Transactions on the Web (TWEB)* 8, 2 (2014), 10.

[24] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[25] G Sadowski and Philip Rathle. 2014. Fraud detection: Discovering connections with graph databases. *White Paper (Neo4J)* (2014).

[26] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB* 1, 1 (2008), 364–375.

[27] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. 2014. Event pattern matching over graph streams. *Proceedings of the VLDB Endowment* 8, 4 (2014), 413–424.

[28] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[29] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23 (January 1976), 31–42. Issue 1.

[30] Changliang Wang and Lei Chen. 2009. Continuous subgraph pattern search over graph streams. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 393–404.

[31] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 265–276.

[32] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *PVLDB* 3, 1 (2010), 340–351.

[33] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. 2011. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment* 4, 8 (2011), 482–493.

APPENDIX
A ALGORITHMS FOR EDGE DELETION

Algorithm 8. DeleteEdgeAndEval (mo, v, v')

Input: Matching order mo and a deleted edge (v, v')

$U \leftarrow \{u \mid g_{DCG}$.GetImplAndExplEdges$(v, u, in) \mid > 0\}$;
/* if $U = \emptyset$, we do not update g_{DCG} by Case 2 of Transition 0 */
foreach $(u \in U)$ **do**
  foreach (child query vertex u' of u in q') **do**
    /* if (u, u') does not match (v, v'), we do not update g_{DCG} by Case 1 of Transition 0 */
    if $((u, u')$ matches $(v, v'))$ **then**
      if $(g_{DCG}$.GetState$(v, u', v') = E)$ **then**
        if (MatchAllChildren(v, u)) **then**
          $m(u') \leftarrow v'$;
          ClearUpwardsAndEval$(mo, u, v, u', (u, u'), m, true)$;
          $m(u') \leftarrow$ NIL;
        ClearDCG$(u', (v, v'))$;
$U \leftarrow \{u \mid g_{DCG}$.GetImplAndExplEdges$(v, u, in) \mid > 0\}$;
foreach $(u \in U)$ **do**
  foreach (non-tree edge (u, u') in q that matches (v, v')) **do**
    if $(\mid g_{DCG}$.GetImplAndExplEdges$(v', u', in) \mid > 0)$ **then**
      if (MatchAllChildren(v, u) and MatchAllChildren(v', u')) **then**
        $m(u') \leftarrow v'$;
        ClearUpwardsAndEval$(mo, u, v,$ NIL, $(u, u'), m, false)$;
        $m(u') \leftarrow$ NIL;

TurboFlux invokes DeleteEdgeAndEval to update the DCG according to Transition 3~5 and to report all negative matches for each edge deletion. DeleteEdgeAndEval is similar to InsertEdgeAndEval except that BuildDCG and BuildUpwardsAndEval are substituted with ClearDCG and ClearUpwardsAndEval, respectively. Another difference is that, we execute ClearUpwardsAndEval before ClearDCG in order to keep explicit edges until we execute SubgraphSearch which reports all negative matches. For

this, Transtions 3 and 5 should be applied in ClearDCG (Line 10) after Transition 4 is applied in ClearUpwardsAndEval (Line 8).

ClearUpwardsAndEval (Algorithm 9) traverses the DCG upwards, makes transitions according to Case 1 of Transition 4, and executes SubgraphSearch to report negative matches when it reaches a starting data vertex. The algorithm of ClearUpwardsAndEval is very similar to BuildUpwardsAndEval. There are two major differences between BuildUpwardsAndEval and ClearUpwardsAndEval: 1) ClearUpwardsAndEval applies Transition 4 rather than Transition 2. 2) We need to keep explicit edges until SubgraphSearch is executed. Thus, transitions of edges should be made after ClearUpwardsAndEval (Lines 15-16).

Algorithm 9. ClearUpwardsAndEval (mo, u, v, u', e_q, m, f_t)

Input: Matching order mo, a query vertex u, a data vertex v, a query vertex u', a query edge e_q, mapping m, and a transition flag f_t

1: $m(u) \leftarrow v$;
2: $precondition \leftarrow false$;
/* precondition for Case 1 of Transition 4 */
3: **if** ($f_t = true$ and | g_{DCG}.GetExplEdges(v, u', out) |= 1) **then**
4: $precondition \leftarrow true$;
5: **foreach** ((v_p, u, v) in g_{DCG}.GetExplEdges(v, u, in)) **do**
6: **if** ($u = u_s$ in q') **then**
7: SubgraphSearch(mo, 1, e_q, m, –);
8: **else**
9: $u_p \leftarrow P(u)$;
10: **if** (MatchAllChildren(v_p, u_p)) **then**
11: **if** ($precondition = true$) **then**
12: ClearUpwardsAndEval(mo, u_p, v_p, u, e_q, m, $true$);
13: **else**
14: ClearUpwardsAndEval(mo, u_p, v_p, u, e_q, m, $false$);
/* Case 1 of Transition 4 */
15: **if** ($precondition = true$) **then**
16: g_{DCG}.MakeTransition((v_p, u, v), I);
17: $m(u) \leftarrow$ NIL;

ClearDCG (Algorithm 10) is similar to BuildDCG except that ClearDCG uses Transitions 3 and 5, while BuildDCG uses Transitions 1 and 2. For each deleted edge (v, v'), ClearDCG first transits the state of (v, u', v') to NULL (Line 1). Then, it traverses the subtree of v' and tries to make transitions of relevant edges (from IMPLICIT or EXPLICIT to NULL) in the DCG (Lines 2-5).

Algorithm 10. ClearDCG (u', (v, v'))

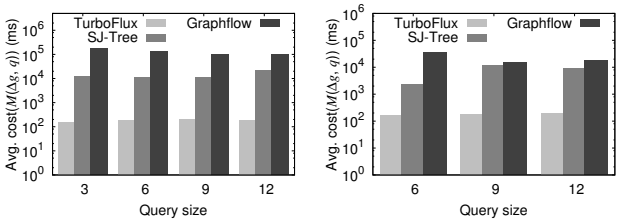
Input: A query vertex u' and a data edge (v, v')

/* Case 1 (if called non-recursively) or 2 (if called recursively) of Transition 3 (if the state of (v, u', v') is EXPLICIT) or 5 (if the state of (v, u', v') is IMPLICIT) */
1: g_{DCG}.MakeTransition((v, u', v'), N);
/* if the transited edge is the last incoming edge of v' whose label is u' */
2: **if** (| g_{DCG}.GetImplAndExplEdges(v', u', in) |= 0) **then**
3: **foreach** (child query vertex u'_c of u' in q') **do**
4: **foreach** ((v', u'_c, v'_c) in g_{DCG}.GetImplAndExplEdges(v', u'_c, out)) **do**
5: ClearDCG(u'_c, (v', v'_c));

B ADDITIONAL EXPERIMENTS

B.1 Comparison in Subgraph Isomorphism

In this section, we compare the performance in subgraph isomorphism semantics. Figure 10a and 10b show the performance results for LSBench tree and graph queries, respectively. Here, we exclude timeout queries in graph homomorphism semantics. TurboFlux again significantly outperforms both SJ-Tree and Graphflow regardless of query type and size. Specifically, TurboFlux outperforms SJ-Tree by 56.43 to 114.95 times for tree queries and 13.63 to 64.35 times for graph queries. TurboFlux also outperforms Graphflow by 274.64 to 1,118.43 times for tree queries and 48.87 to 71.82 times for graph queries. The performance gap is reduced compared to graph homomorphism since the size of intermediate results is reduced due to the injection constraint.

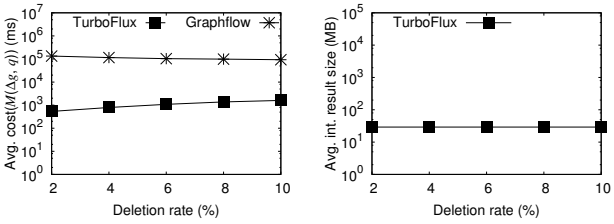


(a) LSBench tree queries. **(b) LSBench graph queries.**

Figure 10: Avg. cost($M(\Delta g, q)$) in subgraph isomorphism.

B.2 Varying the deletion rate

In this section, we compare the performance for the graph update stream containing edge deletions. Here, the deletion rate is defined as *(# of edge deletion / # of edge insertion)* in the graph update stream. Figure 11 shows the performance results for LSBench for varying deletion rates. We fixed the insertion rate to 6% and varied the deletion rate from 2% to 10% in 2% increments. Thus, the number of total update operations also increases accordingly. Note that we conducted this experiment using TurboFlux and Graphflow only because SJ-Tree does not support deletion. Note also that deletion of an edge (v, v') could affect all subtrees of v'. As the deletion rate increases, incremental subgraph matching time of TurboFlux increases, however that of Graphflow slightly decreases. This is because, the edge deletions reduce the input data size of Graphflow directly. Nevertheless, TurboFlux consistently outperforms Graphflow by about two orders of magnitude, and the performance trend shows good scalability of TurboFlux for varying deletion rates. Since we delete a small fraction of triples compared to edge insertions, the average size of intermediate results remains almost the same regardless of the deletion rate as shown in Figure 11b.

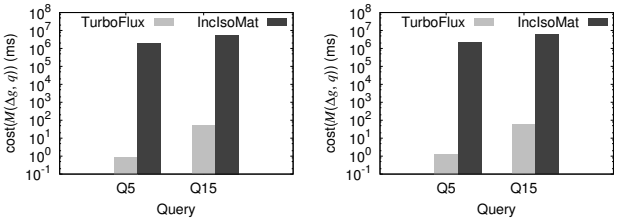


(a) avg. cost($M(\Delta g, q)$). **(b) avg. int. result size.**

Figure 11: Varying deletion rate.

B.3 Comparison with IncIsoMat

In this section, we compare the performance of TurboFlux with IncIsoMat using LSBench dataset and queries. For fair comparison, we use IncIsoMat that is re-implemented in C++ based on the Java source code obtained from its original authors. Since IncIsoMat did not finish the experiments in a reasonable time, we took just two queries and performed experiments using a graph update stream consisting of 10,000 edge insertions. The two queries among 100 tree queries of size 6 have the minimum and maximum costs in TurboFlux. Figure 12a shows the performance of continuous subgraph matching with 10,000 insertions, while Figure 12b shows the performance of continuous subgraph matching with a mix of 10,000 insertions and 600 deletions. TurboFlux significantly outperforms IncIsoMat by up to almost 2,214,086 times! This is because IncIsoMat repeatedly executes subgraph matching and set difference for each update.



(a) Insertion only. (b) Mix of Insertion/Deletion.

Figure 12: Comparison with IncIsoMat.

B.4 Experiments using Netflow

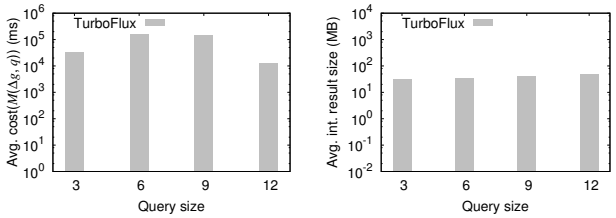
SJ-Tree and Graphflow did not finish the experiments using Netflow in a reasonable time; 100 and 72 queries among the 100 tree queries of size 12 timed out, taking about 200 hours. This is because Netflow has only eight edge labels and no vertex label. Hence, the size of intermediate results is enormous, and cost($M(\Delta g, q)$) in SJ-Tree and Graphflow are very expensive.

Thus, we take a query with the minimum cost($M(\Delta g, q)$) (24.08, 50.37, and 82.46 ms) on TurboFlux at each query size (3, 6, and 9) and perform the experiments with these three queries.

However, all these queries still time out in SJ-Tree. We estimate the cost($M(\Delta g, q)$) for these queries as 40 minutes, since the cost($M(\Delta g, q)$) for slow queries in LSBench was about 1/3 of total execution time. Thus, TurboFlux outperforms SJ-Tree by *at least* 45,886 times on average for these queries. For these queries, the cost($M(\Delta g, q)$) on Graphflow are 2,939ms and 78,876ms at query sizes 3 and 6, respectively, and the query size 9 reaches timeout. For the timeout query, we estimate the cost($M(\Delta g, q)$) as two hours, since Graphflow spends most of its execution time for incremental query processing in slow queries in LSBench. Thus, TurboFlux outperforms Graphflow by *at least* 69,221 times on average for theses queries.

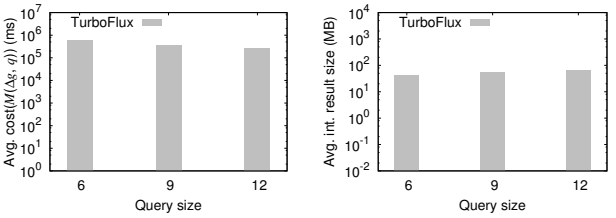
Figure 13 shows the performance results for tree queries on Netflow. Again, we vary the query size from 3 to 12. Figure 13 shows that TurboFlux works well with non-selective queries, although its performance is relatively slower than LSBench. We also perform experiments with graph queries for Netflow. Figure 14 shows that TurboFlux finishes the processing within 10 minutes on average. Considering that the graph update stream consists of more than 1 million edge insertions, TurboFlux can process continuous subgraph matching very fast for non-selective queries as well.

Since SJ-Tree and Graphflow reached timeout for most of our Netflow queries, we performed additional experiments using the queries used in [7]. The experimental results are included in Appendix B.6.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

Figure 13: Tree Queries in Netflow.



(a) avg. cost($M(\Delta g, q)$). (b) avg. int. result size.

Figure 14: Graph Queries in Netflow.

B.5 SJ-Tree with Graph Compression Techniques

We implemented two graph compression techniques of [24] (compression for the data graph) and [14] (compression for the query graph) in SJ-Tree to solve the notorious intermediate result size problem. First, we applied the graph compression technique of [24] that merges multiple data vertices into a hyper-vertex. However, the performance was severely worse than the original SJ-Tree for two main reasons. First, we need to incrementally update the hyper-vertices in query processing time, unlike [24] which compresses a static data graph only. More importantly, whenever a hyper-vertex is updated, we need to update all partial solutions containing the updated hyper-vertex. Thus, we select the fastest and slowest queries in SJ-Tree for each size of LSBench tree and graph queries. Here, SJ-Tree with the compression technique reached the timeout for half of these 14 queries. For the remaining queries, although the intermediate result size is reduced by 28.72%~97.74%, the cost($M(\Delta g, q)$) is increased by 4.32~486.59 times due to the very costly update operations to hyper-vertices and partial solutions containing hyper-vertices.

Second, we applied the compression technique of [14] that compresses a query graph into an NEC tree. For the LSBench tree queries, only 9.5% of the 400 queries can be compressed using the NEC tree, and the cost($M(\Delta g, q)$) and intermediate result size of SJ-Tree are reduced by 4.66%~23.94% and 2.42%~27.99%, respectively. TurboFlux still outperforms SJ-Tree using the NEC compression by up to 288.42 and 102.50 times in terms of cost($M(\Delta g, q)$) and intermediate result size, respectively. For LSBench graph queries, only

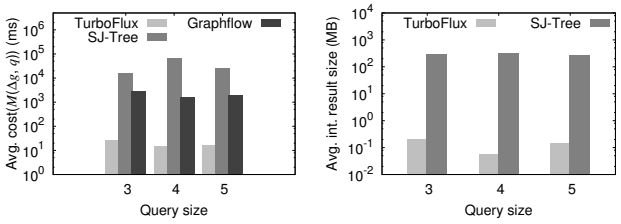
3% of the 300 queries can be compressed, and the cost($M(\Delta g, q)$) and intermediate result size of SJ-Tree are reduced by up to 0.60% and 1.17%, respectively. Thus, TurboFlux still outperforms SJ-Tree by up to 288.42 and 102.50 times in terms of cost($M(\Delta g, q)$) and intermediate result size, respectively. Since SJ-Tree using the NEC compression does not finish the experiments for Netflow in a reasonable time, we select only one compressible query for each size and type, which has the minimum cost($M(\Delta g, q)$) on TurboFlux. 4 of these 7 queries time out, and for the remaining queries, TurboFlux outperforms SJ-Tree by 438.49 and 7.35 times on average in terms of cost($M(\Delta g, q)$) and intermediate result size, respectively.

B.6 Experiments using Netflow Queries in [7]

In this section, we compare the performance using the Netflow queries used in the SJ-Tree paper [7]. Note that this queryset has major drawbacks compared to that which we generated. In [7], they used only two types of queries, path and binary tree, while we used the general shape of tree queries and graph queries. Also, most of the queries from [7] are highly selective; more than a half of them have less than 10 positive matches. Figure 17c-17f in Appendix C shows that our queries have a much wider range of selectivity values.

In this experiment, g_0 contains 16,668,683 triples, and Δg consists of insertions of 1,111,245 triples. Regarding this setting, we were unable to obtain the exact information from the authors. We also varied the size of Δg to see the effect of this setting. We had performance trends similar to Figures 15 and 16 for varying the size of Δg. Specifically, when we decreased the size of Δg, the number of timeout queries decreased accordingly. However, overall performance trends were similar for all the variations of $|\Delta g|$ we tried.

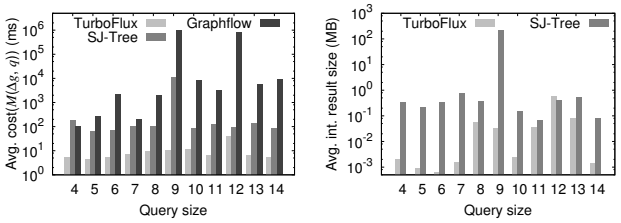
Figure 15 shows the performance results for the path queries. There are 30 queries for each size 3~5, and Figure 17e in Appendix C shows that some of these queries have many matching results. Since SJ-Tree reached the timeout for 5, 1, and 3 of these non-selective queries for size 3, 4, and 5, respectively, we report the results for remaining queries. TurboFlux still outperforms SJ-Tree and Graphflow by up to 4,715.32 and 115.54 times, respectively.



(a) avg. cost($M(\Delta g, q)$). **(b) avg. int. result size.**

Figure 15: Netflow path queries from [7].

Figure 16 shows the performance results for the tree queries. There are 3 queries for each size 4~14, and Figure 17e in Appendix C shows that some of these queries have many matching results. Since SJ-Tree and Graphflow reached the timeout for 6 non-selective queries and of 3 queries, respectively, we report the results for the non-timeout queries only. TurboFlux still outperforms SJ-Tree and Graphflow by up to 1,052.44 and 92,245.38 times, respectively.

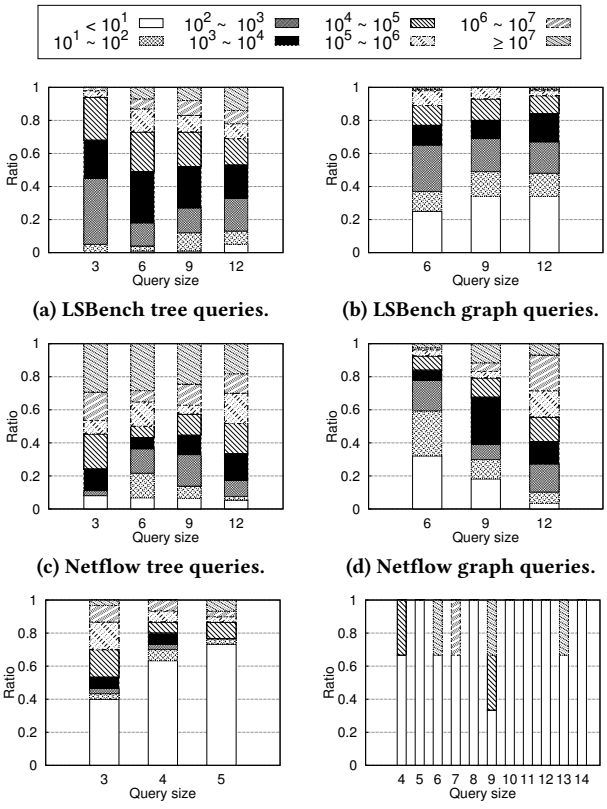


(a) avg. cost($M(\Delta g, q)$). **(b) avg. int. result size.**

Figure 16: Netflow tree queries from [7].

C DISTRIBUTION OF QUERY SELECTIVITY

Figure 17 shows a stacked bar chart about the distribution of the number of positive matches of queries we test. We use eight ranges so that eight segments in each bar represent fractions relative to the total number of queries. It shows that our tree queries have a wide range of selectivity values. It also shows that graph queries are more selective than tree queries, and that the LSBench queries have a smaller number of results than the Netflow queries. Figures 17e, f show that most of the queries from [7] are highly selective.



(a) LSBench tree queries. **(b) LSBench graph queries.**

(c) Netflow tree queries. **(d) Netflow graph queries.**

(e) Netflow path queries from [7]. **(f) Netflow tree queries from [7].**

Figure 17: Selectivity distribution of the LSBench and Netflow queryset.