Incremental Sliding Window Connectivity over Streaming Graphs

Chao Zhang
University of Waterloo
Waterloo, Canada
chao.zhang@uwaterloo.ca

Angela Bonifati
Lyon 1 University, CNRS & IUF
Lyon, France
angela.bonifati@univ-lyon1.fr

M. Tamer Özsu
University of Waterloo
Waterloo, Canada
tamer.ozsu@uwaterloo.ca

ABSTRACT

We study index-based processing for connectivity queries within sliding windows on streaming graphs. These queries, which determine whether two vertices belong to the same connected component, are fundamental operations in real-time graph data processing and demand high throughput and low latency. While indexing methods that leverage data structures for fully dynamic connectivity can facilitate efficient query processing, they encounter significant challenges with deleting expired edges from the window during window updates. We introduce a novel indexing approach that eliminates the need for physically performing edge deletions. This is achieved through a unique bidirectional incremental computation framework, referred to as the BIC model. The BIC model implements two distinct incremental computations to compute connected components within the window, operating along and against the timeline, respectively. These computations are then merged to efficiently compute queries in the window. We propose techniques for optimized index storage, incremental index updates, and efficient query processing to improve BIC effectiveness. Empirically, BIC achieves a 14× increase in throughput and a reduction in P95 latency by up to 3900× when compared to state-of-the-art indexes.

PVLDB Reference Format:
Chao Zhang, Angela Bonifati, and M. Tamer Özsu. Incremental Sliding Window Connectivity over Streaming Graphs. PVLDB, 17(10): 2473 - 2486, 2024.
doi:10.14778/3675034.3675040

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chaozhang-cs/bic.

1 INTRODUCTION

Graphs have been the natural representation of data in many domains [33, 40], where individual entities are represented as vertices and the relationships between entities are represented as edges. With graph-structured data, one of the most interesting operations is to compute *connected components* (CCs) [38, 39], which are basically subsets of vertices in a graph such that all vertices in the subset are connected via undirected paths. Analyzing CCs has many practical applications. In social networks, CCs represent distinct communities or groups of individuals who are tightly connected to each other, and analyzing these components helps in identifying friend circles and influential users within the larger social network [34]. In transport networks, identifying CCs helps in understanding traffic flow and connectivity between different parts of a city, which is crucial for optimizing traffic signal timings, planning public transportation routes, and managing emergency response systems [50]. In financial networks, CCs are used to detect unusual patterns or suspicious clusters of activity that differ from normal transaction patterns, which is crucial in fraud detection [17].

In modern data-driven applications, stream processing [1, 5, 25, 49, 55] is of significant importance, providing real-time data processing capabilities. In the stream model, data arrive at a processing site continuously and each data record contains a payload and a timestamp. Computations are typically performed over *windows* that are continuous finite subsets of streaming data over the unbounded input stream [12]. Of particular interest are *time-based sliding windows* that are characterized by a *window size* and a *slide interval*, denoted as α and β, respectively, which are given in time units. Each window contains data whose timestamps are within the window. For instance, a sliding window with window size of 3 hours and slide interval of 1 hour includes all streaming data of the last 3 hours, and the window is updated every hour by deleting expired data (*i.e.*, data whose timestamp falls outside the window) from the window and inserting new streaming data.

In this paper, we study the computation of CCs over a time-based sliding window in *a streaming graph* [30, 35] that is essentially a stream of edges. For ease of presentation, we focus on *connectivity* queries that check whether two vertices belong to the same CC. Computing connectivity is equivalent to computing CCs as the former requires computing and storing CCs. The problem of computing connectivity over sliding windows is referred to as *sliding window connectivity*. Computing sliding window connectivity allows for the continuous analysis of data streams in real-time, enabling the immediate detection of changes or anomalies in network structures. This is crucial in scenarios like social network monitoring [34], traffic monitoring [50], and fraud detection [17], where timely responses are essential.

Running example. Figure 1 shows a query to compute CCs over a sliding window with window size of 5 and slide interval of 1 (time units are not important). $\mathcal{W}_2$ is the instance of the sliding window ranging from timestamp τ_2 to τ_6, $\mathcal{W}_3$ is the one from τ_3 to τ_7, and so forth. $\mathcal{W}_2$ and $\mathcal{W}_3$ contain only one CC while $\mathcal{W}_4$ has two CCs. Vertices C and G are connected in both $\mathcal{W}_2$ and $\mathcal{W}_3$ but not in $\mathcal{W}_4$.

The naive approach to compute sliding window connectivity is to traverse the streaming graph in each window instance, *e.g.*, performing depth-first-search (DFS) in each $\mathcal{W}_i$ in Figure 1. However, this approach would lead to recomputing CCs from scratch in each window before processing queries, thus being inefficient in a streaming setting. A non-trivial method is to use data structures

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675040

designed for fully dynamic connectivity (FDC) [6, 9, 13–16, 18–20, 22–24, 36, 48, 48, 53, 54]. Specifically, FDC supports 3 operations: `insert`, `delete`, and `query`. As previously mentioned, when the sliding window needs to be updated, it is necessary to remove expired data while adding new data into the window, *e.g.*, from $\mathcal{W}_2$ to $\mathcal{W}_3$ in Figure 1, it is necessary to delete streaming edges with timestamp τ_2 and insert ones with timestamp τ_7. Obviously, the `insert` and `delete` operations supported by FDC can be used to deal with the updates required by sliding windows. The main performance issue of the FDC approaches is the `delete` operation. These use spanning trees to represent CCs, and deleting an edge of a spanning tree requires traversing the graph in the window to verify whether there exists an edge that can reconnect the two split sub-trees (or CCs). In the worst case, this takes the same time as the naive approach (see details in §2). For example, from $\mathcal{W}_3$ to $\mathcal{W}_4$ in Figure 1, edge *(B, D)* needs to be deleted from $\mathcal{W}_3$, which can lead to two CCs in the graph of $\mathcal{W}_3$. Then, it is necessary to traverse the entire graph of $\mathcal{W}_3$ to verify whether the two CCs can be reconnected.

In this paper, we propose the bidirectional incremental computation model (BIC) to process sliding window connectivity. The main idea of BIC is that streaming edges with contiguous timestamps are grouped to form disjoint *chunks*, windows are decomposed according to chunks, and queries are processed by applying partial computations in chunks followed by merging the corresponding partial results. Specifically, we compute two kinds of buffers for each chunk: *forward* and *backward* buffers. Both forward and backward buffers are computed incrementally, achieved by scanning streaming edges in chunks. For the forward buffer, streaming edges are processed sequentially, starting from the first edge and progressing to the last edge within the chunk. Conversely, to compute the backward buffer, the streaming edges are scanned in the reverse order, beginning with the last edge and moving towards the first in the chunk. These two kinds of buffers are stored and the elements in them are merged to compute the query result of each window.

Figure 2 demonstrates the BIC model, using the example provided in Figure 1. In this example, each chunk contains streaming edges spanning 5 timestamps. The forward buffer f_2 for chunk c_2 is computed by scanning streaming edges from timestamps τ_6 to τ_{10}. The backward buffer b_1 for chunk c_1 is computed by scanning streaming edges in reverse within c_1, from τ_5 back to τ_1. Intuitively, $f_2[1]$ captures the connectivity information of edges spanning from τ_6 to τ_7, whereas $b_1[2]$ encompasses the connectivity information of edges from τ_3 to τ_5 (see §5.1 for details). Consequently, any connectivity query Q over $\mathcal{W}_3$ can be computed by merging these segments of connectivity information (see §6.1 for details). The novelty of the BIC model is its ability to deal with the deletion of expired streaming edges from the window without necessitating any corresponding deletions in the maintained index to reflect these changes. For example, Q over $\mathcal{W}_3$ is computed by merging the segments of connectivity information stored in $b_1[2]$ and $f_2[1]$, respectively. Our approach contrasts with a FDC approach, which necessitates deleting expired edges with τ_2 from and inserting new edges with τ_7 into the index of W_2.

In order to leverage the BIC model to compute sliding window connectivity, two main challenges need to be addressed. Challenge C1 is related to the storage of backward buffers. Specifically, every

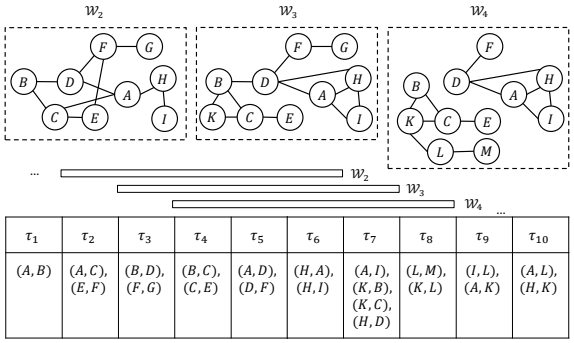

Figure 1: Ruining example.

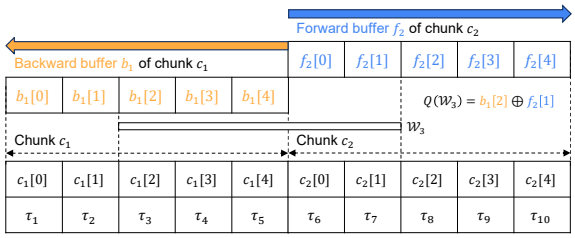

Figure 2: Using BIC for the running example in Figure 1.

element in a backward buffer needs to be stored, which will be retrieved for computing query results. However, storing all elements in a backward buffer will lead to significant overhead (see detail in §5.3). Challenge C2 is on efficiently merging backward and forward buffers to compute query results. Vertices s and t might not be connected in a single buffer (either forward or backward one) but may be connected transitively via vertices that exist in both kinds of buffers. Such inter-buffer checking can result in searching entire buffers, which is not feasible to achieve low latency query processing (see detail in §6.1).

This paper makes the following contributions:

- We introduce the BIC model, which transforms the updates of edge insertions and deletions in sliding window computations into bidirectional incremental computations involving only edge insertions. This approach effectively eliminates the need for the costly operation of deleting expired edges in maintaining indexes for processing connectivity queries in sliding windows.
- To tackle challenge C1, we propose an approach that stores just a single element in each backward buffer instead of all elements. The single element stored is capable of reconstructing all the other elements. This method introduces no additional overhead for query processing.
- To address challenge C2, we design a bipartite graph to bridge the global connectivity information between backward buffers and forward buffers, such that the search space of the merging operation can be significantly reduced.
- Our approach achieves near $O(\log n)$ complexity for both worst-case query time and amortized index update time, with n representing the number of vertices in a window.

Although the number of CCs in the window influences these complexity results, it is noteworthy that the number of CCs is typically very small in real-world graphs.
- Comprehensive experimental evaluation using 8 real-world datasets and 2 synthetic datasets from industrial-grade benchmarks demonstrates that our approach achieves a 14× increase in throughput and a reduction in P95 latency by up to 3900× when compared to state-of-the-art indexes.

Due to space constraints, detailed proofs and algorithm pseudocodes are available in our online technical report [56].

2 RELATED WORK

Fully dynamic connectivity. FDCs [6, 9, 13–16, 18–20, 22–24, 36, 48, 48, 53, 54] are data structures that can reflect the window updates due to expirations and insertions. They support 3 operations: `insert`, `delete`, and `query`. Designing an algorithm that can perform all operations in $O(\log n)$ time (even amortized time) is a long-standing problem. An early work shows the lower bound of $\Omega(\log n/\log\log n)$ [13, 31], and the latest lower bound is proved to be $\Omega(\log n)$ [36]. The existing algorithms can be categorized into two classes: deterministic algorithms [9, 10, 18, 19, 24, 53] and randomized algorithms [11, 14, 15, 20, 23, 48, 51]. The former can return correct query result with a fixed time complexity while the latter has a random variable in either query results or time complexity. There exist two sub-classes in the randomized ones: Monte Carlo algorithms [11, 23, 51] and Las Vegas algorithms [14, 15, 20, 48]. Monte Carlo algorithms have a fixed time complexity but the query results might be incorrect, while Las Vegas algorithms can return correct query results but the time complexity contains a random variable. Deterministic algorithms can be further classified according to their primary focus: amortized algorithms [18, 19, 53] and worst-case algorithms [9, 10, 24], which are designed to optimize amortized and worst-case time complexity, respectively. According to the latest study [6], most of these algorithms rely on complicated data structures, which make them hard to implement. The seminal works, HK [14, 15] and HDT [18, 19], are the only two algorithms that have been successfully implemented and compared [2, 22]. HK is a Las Vegas randomized algorithm while HDT is a deterministic algorithm to amortize the time complexity. Both algorithms are designed based on a framework using spanning trees and incorporate specific techniques to deal with the main bottleneck in the framework. We briefly discuss the framework below. The overarching idea is to use Euler-Tour Tree [14] to store the spanning trees of the input graph. Updates related to non-tree edges are trivial as the connected components will not be changed. Tree edge insertions can be addressed by using the combine operation provided by Euler-Tour Tree. The main problem is the case of tree edge deletion because it requires splitting the spanning tree into two sub-trees and then checking whether there exists a non-tree edge that can reconnect the two sub-trees. Such a non-tree edge is known as a *replacement edge*. Searching for a replacement edge requires traversing the entire graph in the worst case, which takes $O(|V|+|E|)$ time if BFS or DFS is used. HK and HDT design advanced techniques to amortize the cost of searching for replacement edges. D-Tree [6] is a recent work that also uses the spanning tree framework, but includes a different design to deal with the problem. Specifically, D-Tree balances the length from each vertex to its root so as to reduce the average cost of searching. However, the worst-case time complexity of all existing approaches remains the same as using BFS or DFS. In this paper, we focus on designing a deterministic algorithm for computing connectivity queries within sliding windows. Thus, we adopt D-Tree as the current state-of-the-art FDC approach. We also include HDT as a baseline in our experimental evaluation as HDT can be faster than HK according to early experiments [2] and D-Tree is shown to be superior to HK.

Incremental connectivity. When the input graph only has edge insertions, the well-known Union-Find (UF) [45, 46] can be used to compute connectivity queries, which supports two operations: `insert` and `query` in $O(\log n)$ time, where n is the number of vertices in the graph. However, UF cannot deal with edge deletions that are necessary in computing sliding window connectivity. In this paper, we design the bidirectional computation model to completely avoid edge deletions, such that UF can be adopted.

Stream processing systems (SPSs). One of the primary requirements in general SPSs [1, 5, 25, 49, 55] is to achieve high-throughput and low-latency computations. Two general stream processing models exist: *micro-batch model*, *e.g.*, Apache Spark [55], and *continuous model*, *e.g.*, Apache Flink [5]. The former starts the computation when each fix-sized batch is full while the latter immediately processes streaming data. In this paper, we adopt the continuous model, where the query result within a window is computed immediately upon the arrival of a streaming edge that completes the window. This approach facilitates low-latency real-time computation, a key advantage of the continuous model [5]. There also exist graph streaming systems designed for graph analytics [8, 21, 26, 29, 42, 43] and SPARQL query processing [3, 4, 7, 27], and systems for temporary graph analytics [37]. However, these systems lack specific designs for sliding window connectivity queries, and our approach proposed in the paper can be adopted into these systems to support such fundamental operations in graph computation.

3 PROBLEM STATEMENT

Connectivity queries in graphs. We denote an (undirected) graph as $G = (V, E)$, where V is a finite set of vertices and $E \subseteq V \times V$ is a finite set of (undirected) edges. A connectivity query $Q_c(s, t)$ in G checks whether vertices s and t in G are connected, and $Q_c(s, t) = True$ if there exists a path of undirected edges from s to t in G.

Streaming graphs. A streaming edge is an undirected edge with a timestamp, denoted as $e = (u, v, \tau)$, where u and v are the endpoints of e, and τ is the timestamp of e. A streaming graph is an infinite sequence of streaming edges $(e_1, e_2, ...)$, where subscript i of e_i denotes the arrival order of e_i, which is strictly increasing, and $e_i.\tau \leq e_j.\tau$ for any pair of streaming edges e_i and e_j such that $i < j$.

Streaming graphs in sliding windows. In stream processing, the computations are done in windows as the input stream is infinite. A window is a fixed-size subsequence of the input stream, denoted as $\mathcal{W}$. In streaming graph processing, windows are usually *time-based* that have fixed time intervals. A time-based window $\mathcal{W}$ has a beginning timestamp $\mathcal{W}.\tau_b$ and an ending timestamp $\mathcal{W}.\tau_e$. $\mathcal{W}$ over a streaming graph $SG = (e_1, e_2, ...)$ consists of all the streaming edges e_i such that $\mathcal{W}.\tau_b \leq e_i.\tau \leq \mathcal{W}.\tau_e$. A sliding

window is defined with a *window size* α and a *slide interval* β given in time units, which essentially define the following sequence of window instances, *i.e.*, $(\mathcal{W}_1, \mathcal{W}_2, ...)$, such that for each $\mathcal{W}_i$, $\mathcal{W}_i.\tau_e = \mathcal{W}_i.\tau_b + \alpha$, and for every two adjacent $\mathcal{W}_i$ and $\mathcal{W}_{i+1}$, we have $\mathcal{W}_{i+1}.\tau_b = \mathcal{W}_i.\tau_b + \beta$.

Sliding window connectivity. Sliding window connectivity computes connectivity queries in each window instance of a sliding window. We formalize the problem below.

*Definition 3.1 (**Sliding Window Connectivity**).* Given a streaming graph SG and a time-based sliding window $\mathcal{W}(SG)$ defined by window size α and slide interval β, sliding window connectivity is to compute connectivity query $Q_c(s,t)$ between vertices s and t in all window instances $(\mathcal{W}_1, \mathcal{W}_2, ...)$ of $\mathcal{W}(SG)$.

Example 3.2. Figure 1 includes three window instances $\mathcal{W}_2$, $\mathcal{W}_3$, and $\mathcal{W}_4$ of the running example. For query $Q_c(C,G)$, a path exists between C and G in $\mathcal{W}_2$ and $\mathcal{W}_3$, making $Q_c(C,G) = True$, but in $\mathcal{W}_4$, C and G are not connected, so $Q_c(C,G) = False$.

We aim to build an index for efficiently processing connectivity queries within sliding windows due to the inefficiencies of the naive approach based on graph traversal. The main challenge is maintaining the index on the fly. Specifically, when the window is sliding, *expired edges* will be deleted and *new edges* will be inserted. For instance, in Figure 1, when $\mathcal{W}_3$ is sliding to $\mathcal{W}_4$, *expired edges* (B,D) and (F,G) with timestamp τ_3 are deleted and *new edges* (L,M) and (K,L) with timestamp τ_8 are inserted. The gist of the problem, therefore, is to effectively manage graph updates so they can be efficiently incorporated into an index, which in turn facilitates efficient processing of queries.

4 A BIDIRECTIONAL INCREMENTAL MODEL

In this section, we propose the bidirectional incremental computation (BIC) model to address the problem of dynamic computations in sliding windows over streaming graphs. BIC's main idea is to group streaming edges into *chunks* and perform *forward* and *backward* computations in chunks. Both forward and backward computations can be done incrementally. The final computation result in each window instance $\mathcal{W}_i$ can be obtained by merging sub-computation results obtained by the forward and backward computations. We explain the main BIC concepts below, where we use the running example in Figure 2 to guide the presentation.

In the BIC model, the result of a query Q in each window instance $\mathcal{W}$ is computed as follows:

$$Q(\mathcal{W}) = partial(\mathcal{W}^1) \oplus partial(\mathcal{W}^2), \quad (1)$$

where $\mathcal{W}^1$ and $\mathcal{W}^2$ are disjoint sub-windows of $\mathcal{W}$, *i.e.*, $\mathcal{W} = \mathcal{W}^1 \cup \mathcal{W}^2$, and $partial()$ and $\oplus$ are the partial and merging operations for processing Q, respectively. $partial()$ and $\oplus$ will be detailed in §5 and §6, respectively.

Definition 4.1 (Chunks). Given a streaming graph SG, streaming edges in SG are grouped into non-overlapping *chunks* $(c_1, c_2, ..., c_n)$. Each chunk c is an array of slide intervals. The number of slide intervals in each chunk is the chunk size, referred to as $|c|$.

Example 4.2. For the running example in Figure 1, we present (Figure 2) the BIC model with chunk size of 5 slide intervals. Chunks c_1 and c_2 are arrays of 5 elements, including $(c_1[0], ..., c_1[4])$ and $(c_2[0], ..., c_2[4])$, respectively. Window instance $\mathcal{W}_3$ is split into two sub-windows $\mathcal{W}_3^1 = \{\tau_3, \tau_4, \tau_5\}$ and $\mathcal{W}_3^2 = \{\tau_6, \tau_7\}$. Computing query $Q(\mathcal{W}_3)$ can be done by first computing $partial(\mathcal{W}_3^1)$ and $partial(\mathcal{W}_3^2)$, followed by merging the two partial results.

Definition 4.3 (Backward and Forward Buffers). Given a streaming graph SG and chunks $(c_1, c_2, ..., c_n)$ over SG. Two kinds of buffers of size $|c|$ are computed in each chunk c_i.

- *Forward buffer* f_i consists of $|c|$ elements $(f_i[0], ..., f_i[|c|-1])$, and each element $f_i[j] = partial(c_i[0] \cup ... \cup c_i[j])$;
- *Backward buffer* b_i consists of $|c|$ elements $(b_i[0], ..., b_i[|c|-1])$, and each element $b_i[j] = partial(c_i[j] \cup ... \cup c_i[|c|-1])$.

In Figure 2, $b_1[2]$ and $f_2[1]$ are computed as follows: $b_1[2] = partial(c_1[2] \cup c_1[3] \cup c_1[4])$; $f_2[1] = partial(c_2[0] \cup c_2[1])$. Note that each chunk has a backward and a forward buffer, and we only show b_1 and f_2 in Figure 2 for ease of presentation.

In this paper, we focus on using the chunk size that matches the window size divided by the slide interval, facilitating the merging of single elements from backward and forward buffers, respectively.

Definition 4.4 (The BIC Model). Given a query Q and a sliding window over streaming graph SG, the BIC model computes a forward buffer f_i and a backward buffer b_i for each chunk c_i in SG. The computation of Q in each window instance $\mathcal{W}$ is decomposed into partial computations of Q over sub-windows: $Q(\mathcal{W}) = partial(\mathcal{W}^1) \oplus partial(\mathcal{W}^2)$, where $partial(\mathcal{W}^1)$ and $partial(\mathcal{W}^2)$ are obtained by using b_i and f_i.

Example 4.5. For the running example in Figure 2, we have $Q(\mathcal{W}_3) = b_1[2] \oplus f_1[2]$ because $Q(\mathcal{W}_3)$ is split into sub-windows $\mathcal{W}_3^1$ and $\mathcal{W}_3^2$, and $partial(\mathcal{W}_3^1) = b_1[2]$ and $partial(\mathcal{W}_3^2) = f_2[1]$.

For each chunk c_i, f_i, and b_i can be computed incrementally as long as the partial operation has the following property:

$$partial(e_1 \cup ... \cup e_m) = partial(partial(e_1 \cup ... \cup e_{m-1}) \cup e_m). \quad (2)$$

The partial operation that has the above property for computing connectivity queries is called *incremental connectivity* [45, 46], which will be detailed in §5.

The remainder of the paper will address the problem of deploying sliding window connectivity Q_c into the BIC model. In particular, we design $partial()$ in §5 and $\oplus$ in §6 for Q_c such that Equations (1) and (2) in the BIC model can be satisfied.

5 INCREMENTAL SUB-CONNECTIVITY

In this section, we discuss the design of $partial()$ for computing Q_c over sliding windows in BIC, such that Equations (1) and (2) are satisfied. Intuitively, the Union-Find Tree (UFT) [45, 46] algorithm can be applied to incrementally compute connectivity in backward and forward buffers. We identify the underlying challenges on buffer storage in §5.2 and propose a snapshot-based approach to address the challenges in §5.3.

5.1 Incremental sub-connectivity in buffers

Our main finding is that *incremental connectivity can be the partial operation for computing Q_c in the BIC model, which allows fully*

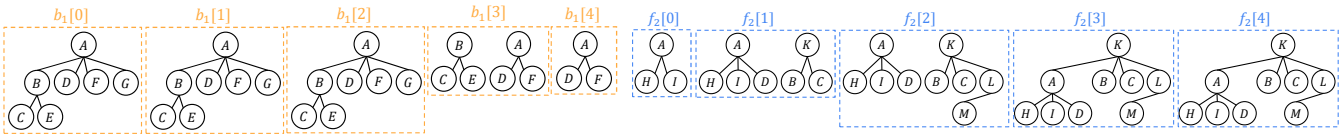

Figure 3: The forward buffer f_2 over chunk c_2 and the backward buffer b_1 over chunk c_1 in the running example in Figure 2.

incremental computations in the backward and forward buffers. Incremental connectivity is the case of computing connectivity queries over a dynamic graph with only edge insertions, which indeed satisfies the property required in Equation (2). Incremental connectivity can be efficiently computed using UFTs. Thus, in order to compute Q_c using the BIC model, each buffer (either f or b) computes UFTs to support the partial operation. We briefly review how UFT operates with respect to edge insertions and use the running example to explain the data structures in buffers.

Each UFT represents a CC in the graph. UFTs are equipped with two operations: *find* and *union*. The *find* operation computes the root of a vertex in the UFT, and the *union* operation links the two roots by making one of them a child of the other. A connectivity query between s and t can be simply computed by checking whether $find(s) = find(t)$. When an edge (u, v) is inserted into the graph, if $find(u) = find(v)$, no update in the UFT is required since u and v are already in the same CC. Otherwise, the *union* operation is performed to link the two UFTs with roots $find(u)$ and $find(v)$, respectively. In this case, u and v are connected by having $find(u)$ as a child of $find(v)$, or vice versa.

In BIC, we compute UFTs in each forward buffer f_i of chunk c_i, which are computed by continuously inserting streaming edges in $(c_i[0], ..., c_i[|c| - 1])$. Each $f_i[j]$ corresponds to the *snapshot* of the UFTs after inserting streaming edges in $c_i[j]$. The computation of the backward buffer b_i is the same as the computation of f_i except that UFTs in b_i are computed by inserting streaming edges in the order of $(c_i[|c| - 1], ..., c_i[0])$. Hereafter, we use the term snapshot to denote $f_i[j]$ or $b_i[j]$.

Example 5.1. Consider running the example in Figures 1 and 2. The corresponding forward buffer f_2 and backward buffer b_1 are presented in Figure 3. f_2 is computed by inserting streaming edges in chunk c_2 from $c_2[0]$ to $c_2[4]$ into the UFTs of f_2, with its UFT snapshots illustrated in Figure 3. Two situations may arise when inserting an edge into f_2. The first situation is that vertices are already connected. For instance, the edge (A, I) in $c_2[1]$ inserted into $f_2[0]$ does not alter the UFT because A and I are already connected, sharing the same root. The second situation is that vertices are not connected, so *union* needs to be performed. For instance, inserting edge (I, L) from $c_2[3]$ into $f_2[2]$ necessitates a *union* operation since I and L have different roots in $f_2[2]$. A is linked as a child of K as the UFT rooted at A has fewer vertices than the UFT rooted at K (an optimization technique explained later). The computation of b_1 works similarly but scans chunk c_1 in reverse, from $c_1[4]$ to $c_1[0]$. $b_1[3]$ shows the snapshot after inserting edges from $c_1[4]$ and $c_1[3]$.

In UFTs, *find* is the building block for query processing and edge insertions. Thus, its cost should be reduced. We define an optimization technique on UFTs in backward and forward buffers.

Definition 5.2 (Optimized UFT). If *union* always makes the root of the smaller UFT as a child of the larger UFT, then the resulting UFT is an optimized UFT, where the size of a UFT ($|UFT|$) is the number of vertices in it.

Lemma 5.3. *The worst-case time complexity of performing find in an optimized UFT is* $O(\log(|UFT|))$.

The lemma can be proved by induction [41]. In the remainder of the paper, any UFT in the BIC model is an optimized UFT. By abuse of notation, we simply denote optimized UFTs as UFTs.

5.2 Computing and accessing buffers

We explore the challenges of computing and storing forward and backward buffers in BIC, starting with the definition of computing and accessing orders. The computing order is forward if snapshots are computed from the first to the last, and backward if from the last to the first. Accessing order follows the same logic; forward for accessing from the first to the last snapshot, and backward for the reverse. Thus, forward buffers have both forward computing and accessing orders, while backward buffers have backward computing and forward accessing orders.

Example 5.4. Consider the running example in Figures 2 and 3. The computations of Q_c in window instances $\mathcal{W}_2$, $\mathcal{W}_3$, and $\mathcal{W}_4$ using b_1 and f_2 are shown as follows: $Q_c(\mathcal{W}_2) = b_1[1] \oplus f_2[0]$; $Q_c(\mathcal{W}_3) = b_1[2] \oplus f_2[1]$; $Q_c(\mathcal{W}_4) = b_1[3] \oplus f_2[2]$. Both b_1 and f_2 are accessed in the forward manner. However, b_1 is computed in the backward manner while f_2 is computed in the forward manner.

To sum up, snapshots in forward buffers are first-computed-first-accessed (FCFA) while snapshots in backward buffers are first-computed-last-accessed (FCLA). FCFA is trivial to address, but FCLA will bring challenges in storing, discussed in §5.3.

5.3 Backward buffer storage

Since forward buffer snapshots are FCFA, they can be processed and accessed immediately. For instance, in Figure 2 at timestamp τ_8, only $f_2[2]$ needs storage; $f_2[0]$ and $f_2[1]$ are unnecessary. Conversely, backward buffers are FCLA, needing more storage; for example, $b_1[3]$ is computed before and accessed after $b_1[1]$, thus must be stored during $b_1[1]$'s access.

The naive solution to deal with FCLA in backward buffer b is to record every snapshot in b. The corresponding storage cost is unfeasibly high because the size of UFTs in b can be very large, considering that each chunk can have millions of streaming edges, and we have to make $|c|$ copies of each UFT. To address the issue, we design a snapshot isolation approach. We denote the edges in UFTs as Union-Find Tree Edges (UFTEs). Notice that UFTEs are not necessarily the same as streaming edges. Our approach labels each UFTE with a snapshot index in b to mark when the UFTE is

inserted into b. Consequently, $b_i[j]$ becomes a subset of UFTEs in $b[0]$, with each UFTE's snapshot index equal to or great than j.

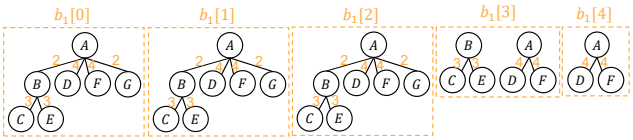

Figure 4: Storing b_1 in Figure 3 using snapshot isolation.

Example 5.5. Figure 4 illustrates the snapshot isolation approach, using the running example in Figure 3. UFTEs (A, D) and (A, F) are inserted into $b_1[4]$ and labeled with snapshot index 4. Similarly, other UFTEs are labeled accordingly. To compute $b_1[3]$ from $b_1[0]$, we retrieve UFTEs in $b_1[0]$ labeled with snapshot indexes 3 and 4, specifically (A, D), (A, F), (B, C), and (B, E).

Lemma 5.6 shows that the *find* operation can be correctly computed by using snapshot isolation to store backward buffer b.

Lemma 5.6. *Given vertex v in $b_i[j]$, computing $find(v)$ in $b_i[j]$ is equivalent to computing $find(v)$ in $b_i[j']$, $j' < j$, if the traversal from v in $b_i[j']$ terminates whenever a UFTE labeled with snapshot index that is smaller than j is visited.*

Lemma 5.6 can be simply transformed into an algorithm for computing *find* in UFTs labeled with snapshot indexes, which has the same time complexity $O(\log |UFT|)$ as the original *find* operation.

In Figure 3, $find(C)$ in $b_1[3]$ is B. If we perform the same computation using $b_1[1]$ in Figure 4, UFTE (B, C) will be visited as it is labeled with 3. However, the next UFTE (A, B) will not be visited as it is labeled with 2 which means (A, B) does not exist in $b_1[3]$. Then, *find* stops and B is returned as the root of C.

Lemma 5.6 allows us to store only $b_i[1]$ as the *find* operation in any $b_i[j]$, $j > 1$ can be correctly computed ($b_i[0]$ is not needed as it contains the same information as $f_i[0]$). Therefore, snapshot isolation necessitates only $O(|UFT|)$ space, compared to $O(|UFT||c|)$ in the naive approach. For storing b_1 in the example of Figure 3, the naive approach needs to store 24 UFTEs while snapshot isolation only needs to store 6 UFTEs labeled with 6 integers.

6 MERGING SUB-CONNECTIVITY

In this section, we propose the merging operation $\oplus$ for computing sliding window connectivity using the BIC model. The primary goal is to merge the sub-connectivity information over sub-windows stored in backward and forward buffers for computing the query result in a full window. We discuss the challenges of merging, followed by a tailored data structure for efficient merging.

6.1 Merging sub-connectivity

In the BIC model, any query Q_c over each window $\mathcal{W}$ is computed as follows $Q_c(\mathcal{W}) = partial(\mathcal{W}^1) \oplus partial(\mathcal{W}^2)$, where $partial(\mathcal{W}^1)$ and $partial(\mathcal{W}^2)$ are stored in $b_i[j]$ and $f_{i+1}[j-1]$, and $\oplus$ is the operation for merging these two partial results. Given $Q_c(s, t)$, the main idea of merging is that if s and t are connected in either $b_i[j]$ or $f_{i+1}[j-1]$, then $Q_c(s, t) = True$. This case is referred to as *intra-buffer checking*. If intra-buffer checking cannot determine the query result, then we check whether s and t are connected via all vertices in $b_i[j]$ or $f_{i+1}[j-1]$, referred to as *inter-buffer checking*.

Example 6.1. For an example of inter-buffer checking, consider query $Q_c(C, G)$ in Figures 2 and 3. $Q_c(C, G)$ over $\mathcal{W}_3$ is evaluated using $b_1[2]$ and $f_2[1]$ according to the BIC model. In $b_1[2]$, C and G are connected as they have the same root A, such that $Q_c(C, G)$ is $True$ in $\mathcal{W}_3$. Consider query $Q_c(I, C)$ in $\mathcal{W}_3$ in Figure 3 for an example of inter-buffer checking. In this case, intra-buffer checking can not determine the query result because I and C are not connected in either $b_1[2]$ or $f_2[1]$. Thus, inter-buffer checking is necessary. In $f_2[1]$, there exist vertices D and B, which have the same root as I and C, respectively, and D and B are connected in $b_1[2]$. Thus, $Q_c(I, C)$ is $True$ in $\mathcal{W}_3$.

Intra-buffer checking is a trivial task as it only requires performing the *find* operation. The main challenge in merging is inter-buffer checking. We denote the vertices that appear in both $f_{i+1}[j-1]$ and $b_i[j]$ as *inter-vertices*. Then, inter-buffer checking requires searching inter-vertices transitively as they can make s and t connected. For instance, in Figure 3, for $b_1[2]$ and $f_2[1]$, vertices D and B are inter-vertices as they appear in both $b_1[2]$ and $f_2[1]$, which can make I and C connected, as discussed in Example 6.1.

Searching for such inter-vertices during inter-buffer checking is computationally expensive because a UFT can have thousands or millions of vertices. More importantly, this procedure needs to be performed recursively, *i.e.*, if s and t are not connected in $f_{i+1}[j-1]$ and the inter-vertices s' and t' that are respectively connected to s and t in $f_{i+1}[j-1]$ are still not connected in $b_i[j]$, then it is necessary to check whether there exist inter-vertices that can make s' and t' connected, and so forth. Such kind of checking needs to be performed exhaustively to determine the final query result.

6.2 Indexing for inter-buffer checking

In order to support efficient inter-buffer checking, we store the connection via inter-vertices between $b_i[j]$ and $f_{i+1}[j-1], \forall j \geq 1$. The main idea is to store the connection between $b_i[j]$ and $f_{i+1}[j-1]$ by recording the roots of inter-vertices. For instance, in Figure 3, inter-vertex C has root A in $b_1[2]$ and root K in $f_2[1]$. Thus, the connected information between A and K via C is recorded, which will be used for inter-buffer checking. Storing the connection between the roots of inter-vertices is more efficient than simply storing inter-vertices. The reason is that the number of inter-vertices can be large but the number of their roots in $b_i[j]$ and $f_{i+1}[j-1]$ is very small (the number of CCs is not large in practice). Consequently, a significant amount of inter-vertices can have the same root, and storing the roots will naturally remove redundancy.

We use a backward-forward bipartite graph (BFBG) to store the connection between $b_i[j]$ and $f_{i+1}[j-1]$. The vertices in a BFBG are the roots in $b_i[j]$ and $f_{i+1}[j-1]$, and the edges represent the connection between the roots via inter-vertices. Each BFBG is computed for each pair of (b_i, f_{i+1}), and the BFBG is updated as the snapshot index j increases. The snapshots of the BFBG can be used to provide the connectivity information between $b_i[j]$ and $f_{i+1}[j-1]$. In order to efficiently compute BFBGs, edges in BFBGs are labeled with additional information.

Definition 6.2. A BFBG $(V_b, V_f, E_{b,f})$ for (b_i, f_{i+1}) records the connection between roots of UFTs in $(b_i[j], f_{i+1}[j-1]), j \geq 1$. V_b and V_f are the subsets of vertices in $b_i[j]$ and $f_{i+1}[j-1]$, respectively. If there exists an inter-vertex v between $b_i[j]$ and $f_{i+1}[j-1]$, there must exist an edge $e_{b,f} = (v_b, v_f) \in E_{b,f}$, such that v_b and v_f are the roots of v in $b_i[j]$ and $f_{i+1}[j-1]$, respectively. In addition, each $e_{b,f}$ is labeled with one or multiple intervals $[j_s, j_e]$ such that the root of the inter-vertex v is v_b in $b_i[j], \forall j \in [j_s, j_e]$.

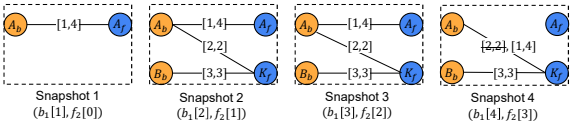

Figure 5: The snapshots of BFBG for b_1 and f_2 in Figure 4.

Example 6.3. The BFBG for $(b_1[1], f_2[0])$ of the running example is shown in snapshot 1 in Figure 5. As the roots of inter-vertex A in $b_1[1]$ and $f_2[0]$ are A and A, respectively. Thus, edge (A_b, A_f) is included in the BFBG. In addition, according to b_1, the root of A is A from $b_1[1]$ to $b_1[4]$. Thus, (A_b, A_f) in snapshot 1 of Figure 5 is labeled with $[1, 4]$.

In BIC, a BFBG is created for each pair of (b_i, f_{i+1}), which will be used to perform inter-buffer checking between $b_i[j]$ and $f_{i+1}[j-1], j \geq 1$. We say the j-th snapshot of the BFBG is the snapshot of the BFBG that can be used to perform inter-buffer checking for $(b_i[j], f_{i+1}[j-1])$, *e.g.*, Figure 5 shows the 4 snapshots of the BFBG for (b_1, f_2) in the running example. We note that the intervals assigned to edges in a BFBG are used to deal with the issues of computing and updating the BFBG, and the corresponding detail will be explained later. We discuss below how a BFBG can be used, followed by how to compute a BFBG.

Inter-buffer checking using a BFBG. The main idea of performing inter-buffer checking using a BFBG is to compute for s and t in $Q_c(s, t)$ their roots in $b_i[j]$ and $f_{i+1}[j-1]$, and then check whether the roots are connected in the BFBG. In checking connectivity using the j-th snapshot of the BFBG, only edges labeled with an interval such that j is in the interval are visited.

Example 6.4. Consider $Q_c(I, C)$ in $\mathcal{W}_3$ in Figure 3. I and C are not connected in $b_1[2]$ and $f_2[1]$. Then, snapshot 2 of the BFBG for (b_1, f_2) shown in Figure 5 is used to perform inter-buffer checking. The computation starts with finding the roots of I and C in $f_2[1]$, *i.e.*, A and K, respectively, and then checks whether A_f and K_f are connected in snapshot 2 of the BFBG. We have $Q_c(I, C) = True$ as A_f and K_f are connected. In this example, edge (A_b, A_f) in snapshot 2 of the BFBG is visited because it has an interval $[1, 4]$ and $2 \in [1, 4]$. Edge (A_b, K_f) is visited because of the same reason. However, edge (B_b, K_f) is pruned because (B_b, K_f) is labeled with only one interval $[3, 3]$ and $2 \notin [3, 3]$.

Computing a BFBG. The BFBG for each pair of (b_i, f_{i+1}) is computed incrementally in the sense that the j-th snapshot is computed by performing updates on top of the $(j-1)$-th snapshot, where the first snapshot is computed on top of the empty BFBG. There exist two kinds of updates in a BFBG: edge insertions and updating $v_f \in V_f$. Edge insertion updates are for the case that a vertex is identified as an inter-vertex, such that the connection between $b_i[j]$ and $f_{i+1}[j-1]$ via the inter-vertex needs to be recorded. Updating $v_f \in V_f$ is used to deal with the issue that the root of an inter-vertex in $f_{i+1}[j-1]$ is changed as j increases (this is because of processing new streaming edges). One may notice that the root of an inter-vertex in $b_i[j]$ may also be changed as j increases. However, such kind of updates do not need to be performed because of the interval assignment mechanism on the edges in BFBGs. The intuition is to insert all the possible roots of an inter-vertex v in different snapshots of b_i when v is identified as an inter-vertex and to distinguish the roots using the assigned intervals. This tailored design is to avoid recomputing the roots of inter-vertices in $b_i[j]$ for different j. We note that it is feasible to have all the roots of an inter-vertex in $b_i[j], \forall j \geq 1$ because b_i is computed in the backward manner.

Edge insertions in a BFBG. For each streaming edge inserted in $f_{i+1}[j-1]$, we check whether each of the two endpoints of the streaming edge is an inter-vertex. If v is identified as an inter-vertex with root v_f in $f_{i+1}[j-1]$ and root v_b in $b_i[j]$, edge (v_b, v_f) is inserted into the current snapshot of the BFBG, which is labeled with $[j_s, j_e], j_s = j$, such that v_b is the root of v from $b_i[j_s]$ to $b_i[j_e]$. If inter-vertex v has different roots in $b_i[j'], j' > j_e$, for each such root v'_b, edge (v'_b, v_f) labeled with $[j'_s, j'_e]$ is inserted, such that v'_b is the root of v from $b_i[j'_s]$ to $b_i[j'_e]$. These additional edge insertions allow the BFBG to keep track of changes of the roots of inter-vertices in b_i and avoid recomputing the roots. We note that computing all possible intervals and the corresponding roots in b_i is feasible (details are discussed in §6.3).

Example 6.5. Snapshot 2 of the BFBG in Figure 5 is computed by inserting edges into snapshot 1. C is identified as an inter-vertex, which has root A in $b_1[2]$ and root K in $f_2[1]$, such that edge (A_b, K_f) is inserted. In addition, C has root B in $b_1[3]$. Thus, (A_b, K_f) is labeled with $[2, 2]$, and another edge (B_b, K_f) labeled with $[3, 3]$ is also inserted. Notice that C does not exist in $b_1[4]$ and thus it is not an inter-vertex in $(b_1[4], f_2[3])$. Such kind of information is recorded in BFBG by the corresponding inserted intervals, *i.e.*, 4 is neither in $[2, 2]$ nor in $[3, 3]$. Inter-vertex B has the same roots and the same intervals as inter-vertex C, such that no edge is inserted into the BFBG for B.

When inserting edge (v_b, v_f) with interval $[j_s, j_e]$, (v_b, v_f) might already exist in BFBG but labeled with a different interval $[j'_s, j'_e]$. In this case, if $[j_s, j_e]$ and $[j'_s, j'_e]$ overlap, they can be merged to condense BFBG, *e.g.*, for edge (A_b, K_f) in snapshot 4 in Figure 5, $[2, 2]$ is subsumed by $[1, 4]$ such that $[2, 2]$ can be deleted.

Updating v_f in edge (v_b, v_f) in a BFBG. When a new streaming edge e is inserted into f_{i+1}, the *union* operation will link the roots of the two endpoints of e in f_{i+1}. Assuming root v is linked as a child root of u. In this case, if v also exists in V_f in the current snapshot of the BFBG, the edges adjacent to v need to be moved to u. Consider $f_2[2]$ and $f_2[3]$ in Figure 3. Both A and K are roots in $f_2[2]$, but A becomes a child of K in $f_2[3]$. Thus, in Figure 5, edge (A_b, A_f) in snapshot 3 is changed to edge (A_b, K_f) in snapshot 4.

6.3 Augmented UFTs in backward buffers

We discuss how to compute the intervals when inserting edges into a BFBG. Interval $[j_s, j_e]$ of edge (v_b, v_f) indicates that v_b is

the root of an inter-vertex v from $b_i[j_s]$ to $b_i[j_e]$. Obviously, it is computationally expensive to have the intervals by computing the root of v in each snapshot of b_i, because the *find* operation in the UFT has to be called repeatedly, *i.e.*, $O(|c|)$ times. We aim at performing the *find* operation once to get all the possible roots and the corresponding intervals. This is possible because backward buffer b_i is computed incrementally and in the backward manner. Therefore, all the information related to the changes of the roots in b_i can be obtained. To achieve this goal, we augment UFTs in each b_i with additional information, and such UFTs are denoted as augmented UFTs (AUFTs), defined below.

Definition 6.6. Given a backward buffer b, augmented UFTs (AUFTs) in each snapshot in b records the following information:

- Each vertex v in an AUFT is labeled with a snapshot index j in b, such that j is the largest of all the snapshot indexes of b, which contains v, *i.e.*, $j = max(\{j' | b[j'] \text{ contains } v\})$.
- Each root r in an AUFT is labeled with an interval $[1, j_e]$, such that j_e is the largest of all the snapshot indexes of b, where r is a root, *i.e.*, $j_e = max(\{j' | r \text{ is a root in } b[j']\})$.
- If vertex v is a root labeled with $[1, j_e]$ in $b[j]$ but v is not a root in $b[j-1]$, then v is labeled with interval $[j, j_e]$.

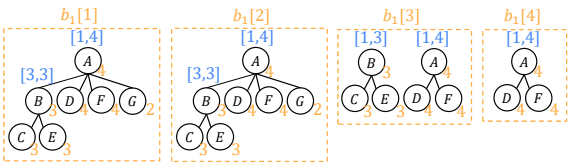

Figure 6: The AUFTs in backward buffer b_1 in Figure 3.

Example 6.7. We present in Figure 6 the AUFTs in the snapshots of b_1 in the running example of Figure 3. Vertices A, D, and F are labeled with 4 because 4 is the largest snapshot index. Vertex B is labeled with interval $[1, 3]$ in $b_1[3]$ as 3 is the largest snapshot index such that B is a root. In $b_1[2]$, the interval of B is changed to $[3, 3]$ because B becomes a child of A.

Although AUFTs record additional information compared to UFTs, AUFTs can be computed and stored in the same way as UFTs. The intuition is that backward buffer b is computed in the backward manner, such that the vertex label of v corresponds to the first snapshot where v appears, and j_e in interval $[1, j_e]$ of root r corresponds to the first snapshot where r is a root. For instance, in Example 6.7, $b_1[4]$ is first computed, where A appears and A is also a root, such that A has a vertex label 4 and interval $[1, 4]$.

We note that snapshots of backward buffer b with AUFTs can be stored using the snapshot isolation approach presented in §5.3 because vertex labels and root intervals can be naturally stored with vertices and roots in AUFTs.

We discuss below how to use the augmented information in AUFTs to compute the roots of inter-vertex v in the snapshots of b_i. The main idea is to retrieve the path from v to the root of v in the current snapshot $b_i[j]$ and then leverage the vertex label of v and root intervals of vertices along the path to compute the roots of v in all the snapshots $b_i[j']$, $j' \geq j$. This is possible because the vertex label of v indicates when v exists in the snapshots of b_i and the root interval of any vertex u along the path indicates when u is a root in the snapshots of b_i. Thus, we can derive the root of v at each snapshot of b_i. For ease of presentation, we use the running example to explain this procedure below.

Example 6.8. Consider Example 6.5, where C is identified as an inter-vertex between $b_1[2]$ and $f_2[1]$, and the root of C in $f_2[1]$ is K. In order to insert edges into the BFBG for $(b_1[2], f_2[1])$, the AUFT stored in $b_1[2]$ shown in Figure 6 is used. The path from C to the root in the AUFT is (C, B, A), where B and A have intervals in the AUFT. The label of C is 3, indicating that C is inserted at $b_1[3]$, and the interval of B is $[3, 3]$, indicating that B is a root in $b_1[3]$. Therefore, we can derive that B is the root of C in $b_1[3]$, which leads to inserting bipartite edge (B_b, K_f) with $[3, 3]$ in snapshot 2 of the BFBG in Figure 5. The next vertex in the path (C, B, A) is A labeled with $[1, 4]$, indicating that A is a root from $b_1[1]$ to $b_1[4]$. Since the current snapshot index is 2 (when C is identified as an inter-vertex) and C has root B in $b_1[3]$, such that we can derive that A is the root of C in $b_1[2]$, which leads to inserting bipartite edge (A_b, K_f) with $[2, 2]$ in snapshot 2 of the BFBG in Figure 5.

6.4 End-to-end computation & complexity

Consider query $Q_c(s, t)$ over sliding windows. Let n and m be the number of vertices and edges in a window, respectively.

Query time. A simple case in query processing is a forward buffer stores the connectivity information of the entire window, *e.g.*, $f_2[4]$ for $\mathcal{W}_6$ in Figure 2. In this case, $Q_c(s, t)$ can be simply processed by checking whether $find(s)$ and $find(t)$ are the same in the forward buffer, which takes at most $O(\log n)$ time (Lemma 5.3). In other cases, the merging operation is necessary. Intra-buffer checking in f_{i+1} or b_i is first applied, taking at most $O(\log n)$ time. If the query result cannot be determined, inter-buffer checking is performed by using the current snapshot of the BFBG for b_i and f_{i+1}, taking $O((|V_b| + |V_f| + |E_{b,f}|) \log |c|)$ time (the intervals of each bipartite edge can be stored using a interval tree [52] and each edge has at most $\log |c|$ intervals). Thus, the total query time is $O(\log n + C)$ in the worst case, where C is $(|V_b| + |V_f| + |E_{b,f}|) \log |c|$. Notice that, $|V_b|$ and $|V_f|$ are at most the number of CCs in b_i and f_{i+1}, respectively, which are very small in practice. Thus, $|E_{b,f}|$, bounded by $|V_b| \times |V_f|$, is also very small. $|c|$ is window size divided by slide interval, which is small in practical settings [35]. Consequently, C can be negligible, and the query time can be $O(\log n)$ in practice.

Update time. After receiving a streaming edge $e = (u, v)$, we need to update the forward buffer, the backward buffer, and the BFBG between them. The update in forward buffer f inserts e into the maintained UFTs, which takes $O(\log n)$ time. If the insertion requires performing the *union* operation, we might also need to reflect the changes in the current BFBG (see updating v_f in (v_b, v_f) in §6.2). The corresponding change takes at most $O(|V_B||c| \log |c|)$ time because there can be at most $|V_b|$ edges adjacent to a vertex in V_f, each edge can have at most $|c|$ intervals, and inserting each interval takes $O(\log |c|)$ time. If the current chunk is full, the backward buffer needs to be computed over the chunk, taking $O(m \log n)$ time because of scanning all the edges in the chunk. If the chunk is not full and u (or v) is an inter-vertex between $b_i[j]$ and $f_{i+1}[j-1]$, the updates in the current BFBG are

needed. It takes at most $O(\log |n|)$ time to compute the roots of inter-vertices in b_i, and there can be at most $O(\log |n|)$ roots. Each root corresponds to an edge insertion into the BFBG, taking at most $O(\log |V_b| + \log |V_f| + \log |c|)$ time. Thus, the update for processing e takes at most $O(C' + m \log n)$ or $O(C' + C'' \log n)$ time, where $C' = |V_B||c| \log |c|$ and $C'' = \log |V_b| + \log |V_f| + \log |c|$. As $|V_f|$, $|V_b|$, and $|c|$ are very small compared to n and m in practice, the update can take $O(m \log n)$ or $O(\log n)$ time.

Notice that the computation of the backward buffer is performed per chunk, such that the cost of $O(m \log n)$ is amortized over all edges in a chunk. Thus, the amortized update time is $O(\log n)$. It is also noteworthy that our algorithm does not need to delete expired edges from each window. Deleting an expired edge takes $O(n + m)$ time in the worst case using the FDC approach (see §2), and each slide interval can have millions of edges, all of which need to be deleted. To sum up, the near $O(\log n)$ worst-case query time and amortized update time demonstrate the high-throughput and low-latency properties of our approach, which is a fundamental requirement in stream processing.

Space. BIC's space complexity is $O(|c|n + m)$. Each backward or forward buffer necessitates at most $O(n)$ space, with BFBG containing at most $O(n)$ edges, each spanning up to $|c|$ intervals. Additionally, storing edges in each chunk is essential for computing the backward buffer of the chunk. The space complexity of BIC mirrors that of FDC approaches [6, 15, 19], which also require $O(n + m)$ space. However, FDC methods store all edges per window instance, while BIC stores all edges per chunk, leading to BIC's lower memory usage, as demonstrated in our experiments (§7.5).

7 EXPERIMENTAL EVALUATION

We evaluate BIC against current state-of-the-art methods using 8 real-world datasets and 2 synthetic datasets from industrial-grade benchmarks. Our evaluation includes throughput and latency analysis in §7.2, emphasizing windows of a few million edges. We then examine the settings with various window sizes and slide intervals in §7.3, focusing on large windows of up to 80 million edges. We analyze the impact of workload size in §7.4. Finally, we analyze the memory usage in all these settings.

7.1 Experimental setup

In the experiments, we denote our approach as **BIC** and compare it against the following approaches : **Euler-Tour Tree (ET-Tree)** [47], **HDT** [18, 19], **D-Tree** [6], **Depth-First Search (DFS)**, and **Recalculating Window Connectivity (RWC)**. ET-Tree, HDT, and D-Tree are FDC data structures, designed based on spanning trees (see §2 for details), and D-Tree is the current state-of-the-art approach. DFS corresponds to executing a depth-first search for each query in each window instance. RWC recalculates all CCs in each window instance for query processing.

Datasets and workloads. In the experiments, we use 10 datasets shown in Table 1, including 8 real-world graphs [28], YG, WT, PR, LJ, SO, OR, FS, and SC, and 2 synthetic graphs from industrial-grade benchmarks, LK [44] with scale factor 1000 and GF [32] with scale factor 25. We present average distance (AD) and diameter (D) in Table 1 for those such that their AD and D are available online. We simulate a streaming graph using each dataset. Edges in SO and LK have timestamps, which we use in the experiments. For the other datasets, we assign timestamps to edges - each timestamp is assigned to 100 edges on average. We randomly generate workloads of queries for each graph. The source code on the dataset and workload setups are provided in our codebase.

Table 1: Overview of datasets.

Dataset	$\|V\|$	$\|E\|$	AD	D
Youtube-growth (YG)	3.2M	14.4M	5.2	31
Wiki-top (WT)	1.7M	28.5M	-	9
Pokec (PR)	1.6M	30.6M	4.6	14
LiveJournal (LJ)	3.9M	34.6M	-	17
StackOverflow (SO)	2.6M	63.4M	3.9	11

Dataset	$\|V\|$	$\|E\|$	AD	D
Orkut (OR)	3M	117.1M	4.2	10
LDBC SNB Knows (LK)	3.3M	187.2M	-	-
Graph-500 (GF)	17M	523.6M	-	-
Friendster (FS)	63.6M	1.8B	-	32
Semantic Scholar (SC)	65M	8.27B	-	-

Evaluation metrics. We report both throughput and tail latency in our experiments, which are crucial in stream processing systems. Throughput is computed as the processing time of a dataset divided by the number of edges in the dataset. For latency, we record the response time of each approach when a streaming edge indicates the current window is complete. This includes processing queries in the window and window updates, *i.e.*, a round of `query`, `insert` and `delete`. Due to space considerations, we do not show breakdown figures, but we can report that per-edge processing latency is the dominant overhead. Notice that, the response time includes the execution time of the most expensive operation in each compared approach, *e.g.*, computing backward buffers in BIC, computing connected components in RWC, performing traversals in DFS, and deleting expired edges in FDC-based approaches (ET-Tree, HDT, and D-Tree). We report 99th and 95th percentile latency, referred to as P99 latency and P95 latency, respectively. These are widely used metrics to evaluate system response time.

Settings. As queries and window updates (edge insertions and deletions) are performed in sliding windows, such that the number of edges in windows and slide intervals has an impact on the performances (throughput and tail latency) of all compared approaches. Thus, we consider various settings with respect to the number of edges in windows and slide intervals. We start the experiments in §7.2 with a window size and a slide interval such that each window and each slide interval contain on average 3M edges and 150K edges, respectively, across all datasets. We report the throughput and tail latency of each approach for each dataset. Then, in §7.3, we focus on the cases where windows increase to up to 80M edges and slide intervals increase up to 8M edges. Specifically, we consider the following two scenarios in §7.3. Scenario 1 fixes the slide interval but varies the window sizes. Each slide interval contains on average 1M edges, and windows contain on average 10M, 20M, 40M, and 80M edges, respectively. Scenario 2 fixes the window size but varies slide intervals. Each window contains on average 80M edges, and slide intervals contain on average 1M, 2M, 4M, and 8M edges, respectively. We report throughput and tail latency (P95 latency and P99 latency) in Scenario 1 and in Scenario 2, respectively. We use GF, FS, and SC in the experiments in §7.3 as they have a large number of edges that allow to test different settings. In the experiments in §7.2 and §7.3, we focus on computing a workload of 100 queries, and all approaches are compared except DFS as performing DFS for each query in each window instance has very poor performance. Then, in the experiments in §7.4, we study the impact of the number of queries in workloads, where we consider 1, 10,

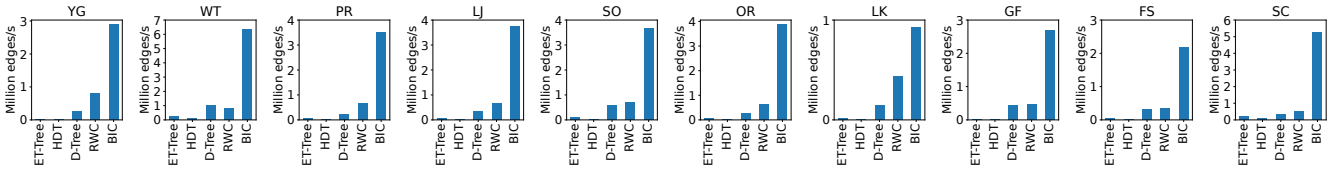

Figure 7: Throughput analysis using windows of on average 3M edges and slide intervals of on average 150K edges.

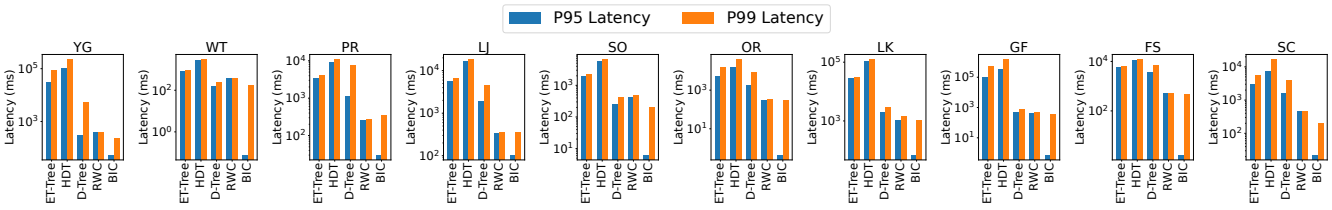

Figure 8: P95 and P99 tail latency analysis using windows of on average 3M edges and slide intervals of on average 150K edges.

100, 1000, and 10000 queries, and DFS is included. Each window and slide interval contains on average 20M and 1M edges, respectively, for the experiment in §7.4. We use these settings of window sizes and slide sizes to study the memory usage of all approaches in §7.5.

The implementation of BIC, all compared approaches, and experiment setups are included in our codebase that has been made publicly available. We run the experiments on a server with Ubuntu 22.04, 80 CPUs of 2.30GHz, and 1TB main memory. We note that all approaches are single-threaded.

7.2 Throughput and tail latency

Throughput. The results of our throughput analysis experiments are displayed in Figure 7. We conducted throughput experiments for all methods across all datasets. Notably, the results show that BIC significantly outperforms the compared methods in all datasets. Specifically, BIC improves throughput by up to 14× over D-Tree, up to 500× over ET-Tree, up to 1000× over HDT, and up to 7× over RWC. Delving deeper, we find that BIC and RWC exhibit superior throughput over FDC-based approaches, including D-Tree, ET-Tree, and HDT. The primary reason for this is the considerable overhead associated with deleting expired edges in FDC-based approaches, a process not required by RWC and BIC (detailed in §2). Interestingly, although D-Tree does not boast better time complexities than ET-Tree and HDT, it demonstrates higher throughput. This is largely due to its simplicity in implementation, unlike the complex data structures used in ET-Tree and HDT. Further distinguishing the performance, the comparison between RWC and BIC reveals a significant throughput difference. RWC computes all connected components for each window instance, whereas BIC handles this computation for each chunk. In BIC, the computed components in each chunk are then utilized for query processing across 20 window instances in our tested sliding window setup (where the window size is 20× larger than the slide interval). This is a key factor in why BIC shows superior throughput results compared to RWC.

Tail latency. The results of tail latency analysis experiments are depicted in Figure 8. Overall, BIC significantly outperforms other tested methods in both P99 latency and P95 latency. In terms of P99 latency, BIC's improvements are notable: it enhances performance by up to 28× compared to D-Tree, up to 1500× over ET-Tree, up to 4500× over HDT, and up to 2.3× over RWC. BIC's advancements are even more pronounced in P95 latency, where it surpasses other methods by a larger margin: up to 3900× over D-Tree, up to 100000× over ET-Tree, up to 400000× over HDT, and up to 4700× over RWC. BIC's performance improvements are due to its specialized computation process. The most compute-intensive scenario in BIC occurs when a streaming edge completes a chunk, necessitating the computation of the chunk's backward buffer. This involves scanning all edges in the chunk to construct the AUFTs (as discussed in §6.3). Such intensive computation is only required for the last edge in a chunk, not for every streaming edge. Given that each computed chunk contributes to processing queries in 20 window instances in this experiment, this costly computation significantly impacts P99 latency but is less influential in P95 latency. Contrastingly, in other methods like RWC and FDC-based approaches, there is no substantial difference between P99 latency and P95 latency. RWC necessitates the computation of all connected components for each window instance before query processing. FDC-based methods face the significant challenge of deleting expired edges, a mandatory step for updating each window instance.

Discussion on average distance and diameter. Average distance and diameter, which are the mean shortest path length and maximum shortest path length in a graph, respectively, affect the performance of FDC methods like ET-Tree, HDT, and D-Tree. This is because FDC approaches, which are based on spanning trees, necessitate performing a graph traversal (*e.g.*, BFS) to find a replacement edge when a tree edge is deleted in a spanning tree (see §2 for details). Additionally, average distance and diameter directly impact the efficiency of graph traversal. In contrast, RWC and BIC, based on Union-Find Trees, are unaffected by these metrics since they don't require graph traversal. Due to the computational cost, average distance and diameter are reported in Table 1 only for the graphs such that their average distance and diameter are available online. Generally, these values are low, favoring D-Tree's performance; however, BIC consistently outperforms D-Tree in these scenarios.

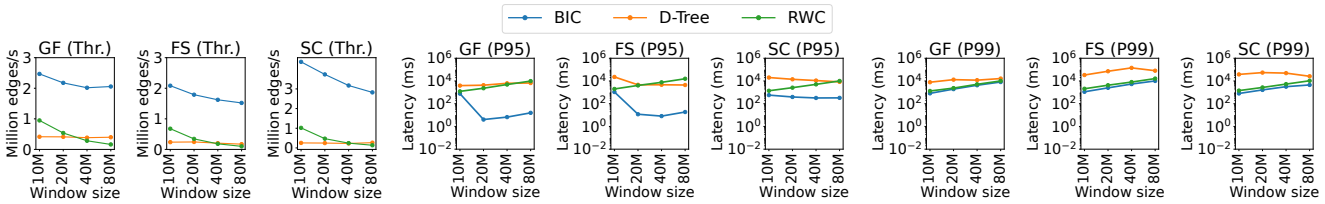

Figure 9: Throughput, P95 latency, and P99 latency analysis using slides of 1M edges and windows of various sizes on average.

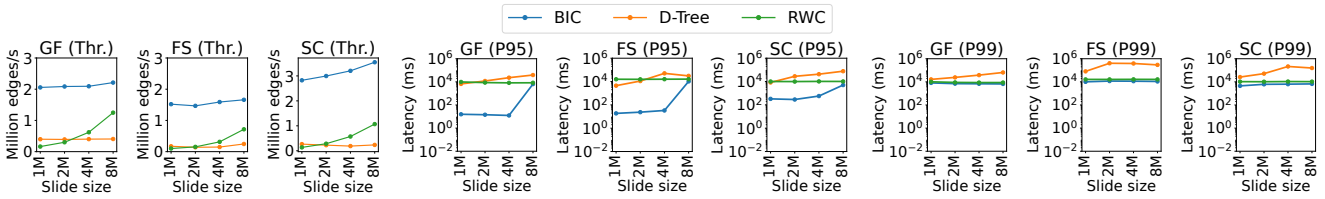

Figure 10: Throughput, P95 latency, and P99 latency analysis using windows of 80M edges and slides of various sizes on average.

7.3 Impact of window sizes and slide sizes

Window sizes. In Figure 9, we present the results of our experiments analyzing throughput and tail latency using windows with a fixed slide interval but varying window sizes. Overall, we observe that as the window size increases, the improvement of BIC over D-Tree and RWC remains consistent with the trends reported in Figures 7 and 8. Below, we delve into more detailed findings in throughput and latency results, respectively. Specifically, we observe a significant decrease in RWC's throughput with larger window sizes. This reduction is attributed to RWC's approach of computing connected components for each window instance, where a larger window size directly translates to increased computation time and subsequently reduced throughput. In contrast, BIC's throughput is less adversely affected by increasing window sizes. This is because, in BIC, the increment in chunk size, which is proportional to the window size, allows for the computed backward buffer to be applied across a broader range of window instances. This efficient utilization helps mitigate the increased computation cost. D-Tree also experiences a decrease in throughput as the window size grows. This decrease is linked to the expansion of D-Tree's primary data structure (spanning trees), leading to more resource-intensive operations. Turning to tail latency, we see a clear increase in both P99 latency and P95 latency for RWC as the window size expands. This is a direct consequence of the longer time needed to compute connected components in each larger window instance. For BIC, while the P99 latency similarly increases with window size, an interesting trend is observed in P95 latency. We note a significant reduction in P95 latency when the window size shifts from 10M to 20M edges. This phenomenon is explained by the frequency of computing backward buffers in chunks, which is a crucial factor in P95 latency. In smaller windows, this computation happens every 10 instances, but in larger windows, it occurs every 20 instances, effectively reducing the P95 latency. D-Tree's tail latency demonstrates irregular patterns, varying depending on the graph. This inconsistency is anticipated as D-Tree's latency is more sensitive to the slide size rather than the window size.

Slide sizes. In Figure 10, we present the results from our experiments analyzing throughput and tail latency using windows of a fixed size but with varying slide sizes. Generally, we observe that as the slide size increases, BIC's improvement over D-Tree and RWC in terms of throughput and tail latency remains consistent with the improvements reported in Figures 7 and 8. An exception to this trend is noted in the case of BIC's throughput improvement over RWC. We delve into these details below. There is a noticeable increase in RWC's throughput with larger slide sizes. This is because a larger slide size means less frequent query processing over window instances, thereby reducing the frequency of RWC's major bottleneck — computing the connected component in windows. In extreme cases, where the slide interval equals the window size, window instances become disjoint (aka *tumbling windows*), reducing the need for incremental computation methods like BIC and D-Tree. Tumbling windows are generally less challenging than sliding windows. Both BIC and D-Tree also experience a slight increase in throughput, attributed to the less frequent processing of queries. Turning to tail latency, we see that the tail latency for RWC remains largely unchanged due to the fixed window size. BIC shows similar behavior in tail latency, except for P95 latency in cases where the slide size increases from 4M to 8M edges. This pattern mirrors the trend observed in Figure 9 when the window size increased from 10M to 20M edges. In cases where the slide size is 8M, the computation time for backward buffers is factored into P95 latency, unlike in scenarios with smaller slide sizes. Both P95 latency and P99 latency in D-Tree exhibit a noticeable increase with larger slide sizes. This is because D-Tree's tail latency is predominantly influenced by the cost of deleting expired edges, and a larger slide size results in a greater number of edges to delete.

7.4 Impact of workload size

In Figure 11, we present our experimental results on throughput and tail latency across workloads of varying sizes. Interestingly, in scenarios with small workloads, such as a single query, neither D-Tree nor RWC exhibit superior performance compared to DFS

Figure 11: Throughput, P95 latency, and P99 latency analysis using various workload sizes.

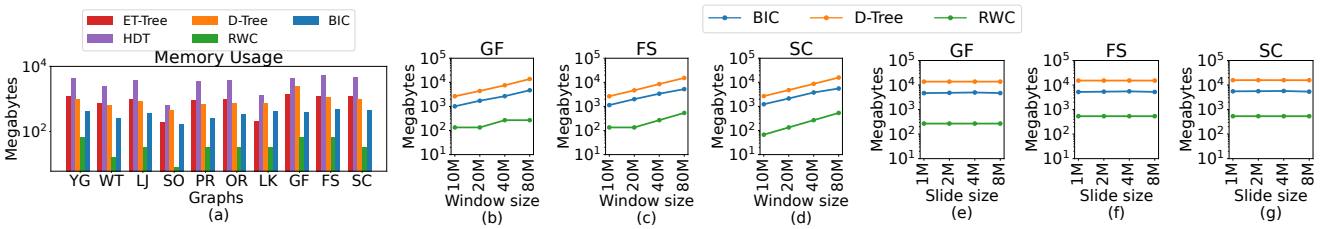

Figure 12: Memory usage analysis: (a) analyzes per graph; (b)-(d) assess window size impact; (e)-(g) evaluate slide size impact.

in terms of throughput and latency. However, BIC significantly outperforms DFS, particularly in terms of P95 latency. This finding suggests that BIC is consistently beneficial for index construction to accelerate query processing. The performance of DFS decreases dramatically as the workload size increases, encountering timeouts in large workloads within a 10-hour limit (indicated by 'X' in Figure 11). This is because processing each query requires performing a single DFS. The influence of workload size on the performance of RWC is minimal. This is because once the connected components for each window instance in RWC are computed, the time taken for query processing becomes insignificant. D-Tree exhibits similar behaviors as its performance bottleneck is in index updating instead of query processing, and the window and slide size are fixed in this setting. The impact of workload size on BIC's throughput is slight. For P99 latency, the workload size does not significantly affect BIC since the query processing time is relatively minor compared to the time required for computing backward buffers. P95 latency in BIC tends to increase with larger workloads. This increase is due to the fact that, for each query, BIC might need to search over the maintained BFBG (§6.2) in the worst-case scenario.

7.5 Memory usage

In Figure 12, we present the memory usage results for all methods, detailing memory usage on each graph in Figure 12(a), a sliding window with a slide size of 1M edges and different window sizes in Figures 12(b)-(d), and a sliding window with a window size of 80M edges and different slide sizes in Figures 12(e)-(g). Index-based methods like ET-Tree, HDT, D-Tree, and BIC maintain an index for streaming edge processing, with their memory efficiency gauged by the median index size across window instances. RWC, differentiating from index-based approaches, constructs Union-Find Trees per window instance by scanning all edges, thus we capture its memory usage based on the size of these trees and report the median one of all window instances. Generally, RWC consumes the least memory as it stores only vertices. In contrast, FDC methods (ET-Tree, HDT, and D-Tree) require more memory for holding both edges and vertices per window instance. Among index-based approaches, BIC is the most memory-efficient, utilizing Union-Find Trees for indexing and storing edges per chunk rather than per window instance. Although RWC is less memory-intensive than BIC, it faces latency challenges in constructing Union-Find Trees, as depicted in Figures 8, 9, and 10. Memory usage escalates with window size due to the increased count of vertices and edges, as shown in Figures 12(b)-(d), but remains unaffected by slide size changes (Figures 12(e)-(g)) when the window size is constant.

8 CONCLUSION

We study index-based query processing in sliding windows over streaming data, focusing specifically on connectivity queries over streaming graphs. The dynamic nature of sliding windows, characterized by deleting expired edges and inserting new edges, presents inherent challenges in index maintenance. To address these challenges, we propose BIC, a generic computation model towards high-throughput and low-latency sliding window query processing. BIC is tailored to circumvent physically performing edge deletions in the maintained indexes, a process typically identified as a performance bottleneck in existing indexes for connectivity queries in dynamic graphs. We then propose specialized data structures that synergize with BIC to efficiently process sliding window connectivity. The results from our comprehensive experimental evaluation highlight the effectiveness of our approach, showcasing up to a 14× increase in throughput and a reduction in P95 latency by up to 3900× when compared to state-of-the-art indexes.

We recognize several promising directions for future research. These include the development of optimization and parallelization techniques and the study of using different chunk sizes in BIC, aimed at further enhancing BIC's performance. Additionally, we envisage the application of the BIC model in broader contexts, extending its capabilities to achieve high throughput and low latency in processing a variety of queries within sliding windows.

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endowment* 6, 11 (2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[2] David Alberts, Giuseppe Cattaneo, and Giuseppe F. Italiano. 1997. An Empirical Study of Dynamic Graph Algorithms. *ACM J. Exp. Algorithmics* 2 (jan 1997), 5–es. https://doi.org/10.1145/264216.264223

[3] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2009. C-SPARQL: SPARQL for Continuous Querying. In *Proc. 18th Int. World Wide Web Conf.* 1061–1062. https://doi.org/10.1145/1526709.1526856

[4] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. 2010. Enabling Ontology-Based Access to Streaming Data Sources. In *Proc. 9th Int. Semantic Web Conf.* 96–111.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[6] Qing Chen, Oded Lachish, Sven Helmer, and Michael H. Böhlen. 2022. Dynamic Spanning Trees for Connectivity Queries on Fully-Dynamic Undirected Graphs. *Proc. VLDB Endowment* 15, 11 (2022), 3263–3276. https://doi.org/10.14778/3551793.3551868

[7] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. 2015. Towards a Unified Language for RDF Stream Query Processing. In *Proc. 12th Extended Semantic Web Conf.* 353–363.

[8] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *Proc. 28th Int. Conf. on Data Engineering*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[9] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification—a Technique for Speeding up Dynamic Graph Algorithms. *J. ACM* 44, 5 (sep 1997), 669–696. https://doi.org/10.1145/265910.265914

[10] Greg N. Frederickson. 1985. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM J. on Comput.* 14, 4 (1985), 781–798. https://doi.org/10.1137/0214055

[11] David Gibb, Bruce Kapron, Valerie King, and Nolan Thorn. 2015. Dynamic graph connectivity with improved worst case update time and sublinear space. arXiv:1509.06464 [cs.DS]

[12] Lukasz Golab and M. Tamer Özsu. 2003. Issues in Data Stream Management. *ACM SIGMOD Rec.* 32, 2 (jun 2003), 5–14. https://doi.org/10.1145/776985.776986

[13] Monika Rauch Henzinger and Michael L Fredman. 1998. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica 22*, 3 (1998), 351–362.

[14] Monika Rauch Henzinger and Valerie King. 1995. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. In *Proc. 27th Annual ACM Symp. on Theory of Computing*. Association for Computing Machinery, New York, NY, USA, 519–527. https://doi.org/10.1145/225058.225269

[15] Monika R. Henzinger and Valerie King. 1999. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM* 46, 4 (1999), 502–516. https://doi.org/10.1145/320211.320215

[16] Monika R. Henzinger and Valerie King. 2001. Maintaining Minimum Spanning Forests in Dynamic Graphs. *SIAM J. on Comput.* 31, 2 (2001), 364–374. https://doi.org/10.1137/S0097539797327209

[17] Amy Hodler. 2023. White Paper: Financial Fraud Detection with Graph Data Science.

[18] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 1998. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. In *Proc. 30th Annual ACM Symp. on Theory of Computing*. 79–89. https://doi.org/10.1145/276698.276715

[19] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (2001), 723–760. https://doi.org/10.1145/502090.502095

[20] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. 2017. Fully Dynamic Connectivity in O(Log n(Log Log n)2) Amortized Expected Time. In *Proc. 28th Annual ACM-SIAM Symp. on Discrete Algorithms*. 510–520.

[21] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-Evolving Graph Processing at Scale. In *Proc. 4th Int. Workshop on Graph Data Management Experiences & Systems*. Article 5, 6 pages. https://doi.org/10.1145/2960414.2960419

[22] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. 2002. An Experimental Study of Polylogarithmic, Fully Dynamic, Connectivity Algorithms. *ACM J. Exp. Algorithmics* 6 (dec 2002), 4–es. https://doi.org/10.1145/945394.945398

[23] Bruce M. Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic Graph Connectivity in Polylogarithmic Worst Case Time. In *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms*. 1131–1142.

[24] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. 2016. Faster Worst Case Deterministic Dynamic Connectivity. In *24th Annual European Symposium on Algorithms (ESA 2016)*, Vol. 57. 53:1–53:15. https://doi.org/10.4230/LIPIcs.ESA.2016.53

[25] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 239–250. https://doi.org/10.1145/2723372.2742788

[26] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (jan 2020), 40 pages. https://doi.org/10.1145/3364180

[27] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proc. 10th Int. Semantic Web Conf.* 370–388.

[28] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[29] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proc. 14th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.* Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[30] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *ACM SIGMOD Rec.* 43, 1 (may 2014), 9–20. https://doi.org/10.1145/2627692.2627694

[31] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. 1994. Complexity models for incremental computation. *Theor. Comput. Sci.* 130, 1 (1994), 203–236.

[32] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[33] Mark E. J. Newman. 2010. *Networks: An Introduction.* Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199206650.001.0001

[34] Evelien Otte and Ronald Rousseau. 2002. Social network analysis: a powerful strategy, also for the information sciences. *J. Inf. Sci.* 28, 6 (2002), 441–453. https://doi.org/10.1177/016555150202800601 arXiv:https://doi.org/10.1177/016555150202800601

[35] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1415–1430. https://doi.org/10.1145/3318464.3389733

[36] Mihai Patrascu and Erik D. Demaine. 2006. Logarithmic Lower Bounds in the Cell-Probe Model. *SIAM J. on Comput.* 35, 4 (2006), 932–963. https://doi.org/10.1137/S0097539705447256

[37] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2021. Distributed Temporal Graph Analytics with GRADOOP. *VLDB J.* 31, 2 (may 2021), 375–401. https://doi.org/10.1007/s00778-021-00667-4

[38] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endowment* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[39] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[40] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: A community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[41] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms, 4th Edition.* Addison-Wesley. I–XII, 1–955 pages.

[42] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proc. 22nd Int. Euro-Par Conf.* 319–333.

[43] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates. In *Proc. 9th ACM Symp. on Cloud Computing*. 301–312. https://doi.org/10.1145/3267809.3267811

[44] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endowment* 16, 4 (2022), 877–890.

[45] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. https://doi.org/10.1145/321879.321884

[46] Robert Endre Tarjan. 1979. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.* 18, 2 (1979), 110–127. https://doi.org/10.1016/0022-0000(79)90042-4

[47] Robert E. Tarjan and Uzi Vishkin. 1985. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. on Comput.* 14, 4 (1985), 862–874.
[48] Mikkel Thorup. 2000. Near-Optimal Fully-Dynamic Graph Connectivity. In *Proc. 32nd Annual ACM Symp. on Theory of Computing*. 343–350. https://doi.org/10.1145/335305.335345
[49] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 147–156. https://doi.org/10.1145/2588555.2595641
[50] Christian Von Ferber, Taras Holovatch, Yu Holovatch, and V Palchykov. 2009. Public transport networks: empirical analysis and modeling. *Eur. Phys. J. B* 68 (2009), 261–275.
[51] Zhengyu Wang. 2015. An Improved Randomized Data Structure for Dynamic Graph Connectivity. arXiv:1510.04590 [cs.DS]
[52] Wikipedia. 2023. Interval tree — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Interval%20tree&oldid=1178718117. [Online; accessed 23-October-2023].
[53] Christian Wulff-Nilsen. 2013. Faster Deterministic Fully-Dynamic Graph Connectivity. In *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms*. 1757–1769. https://doi.org/10.1137/1.9781611973105.126
[54] Christian Wulff-Nilsen. 2017. Fully-Dynamic Minimum Spanning Forest with Improved Worst-Case Update Time. In *Proc. 49th Annual ACM Symp. on Theory of Computing*. 1130–1143. https://doi.org/10.1145/3055399.3055415
[55] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. 24th ACM Symp. on Operating System Principles*. 423–438. https://doi.org/10.1145/2517349.2522737
[56] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2024. Incremental Sliding Window Connectivity over Streaming Graphs. arXiv:2406.06754 [cs.DB]