Enabling Window-Based Monotonic Graph Analytics with Reusable Transitional Results for Pattern-Consistent Queries

Zheng Chen
Renmin University of China
chenzheng123@ruc.edu.cn

Feng Zhang
Renmin University of China
fengzhang@ruc.edu.cn

Yang Chen
Renmin University of China
chenyang_inf22@ruc.edu.cn

Xiaokun Fang
Renmin University of China
fangxiaokun@ruc.edu.cn

Guanyu Feng
Tsinghua University
fgy18@mails.tsinghua.edu.cn

Xiaowei Zhu
Ant Group
robert.zxw@antgroup.com

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Evolving graphs consisting of slices are large and constantly changing. For example, in Alipay, the graph generates hundreds of millions of new transaction records every day. Analyzing the graph within a temporary window is time-consuming due to the heavy merging of slices. Fortunately, we have discovered that most queries exhibit consistent patterns and possess monotonic properties. As a result, transitional results can be computed within slice generation for reuse. Accordingly, we develop MergeGraph enabling window-based monotonic graph analytics with reusable transitional results for pattern-consistent queries. MergeGraph has three advantages over previous works. First, it is the first system specifically tailored for window-based monotonic graph analytics with pattern-consistent queries. Second, it effectively utilizes transitional results from different slices concurrently. Third, MergeGraph boasts a high degree of expressiveness, supporting a broad spectrum of monotonic graph queries. Experimental results demonstrate that MergeGraph delivers significant performance benefits. In evaluating four typical graph applications, MergeGraph achieves an average speedup of 11.30× compared to state-of-the-art methods.

PVLDB Reference Format:
Zheng Chen, Feng Zhang, Yang Chen, Xiaokun Fang, Guanyu Feng, Xiaowei Zhu, Wenguang Chen, and Xiaoyong Du. Enabling Window-Based Monotonic Graph Analytics with Reusable Transitional Results for Pattern-Consistent Queries. PVLDB, 17(11): 3003 - 3016, 2024.
doi:10.14778/3681954.3681979

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZhengChenCS/MergeGraph_AE_VLDB24.git.

1 INTRODUCTION

Graph analytics is a fundamental application of graph databases and warehouses [5, 6, 8, 12, 15, 18–22, 24–26, 29, 35, 38–40, 42, 44–50, 53, 58, 60, 63, 64, 69, 72, 75–77, 79, 83, 85–87, 89, 90, 92, 93, 95, 97, 101, 102]. In the real world, graphs are constantly evolving, often comprised of slices. For example, Alipay generates 680 million new transaction records daily [1]. It is common for commercial companies to conduct their queries on an evolving graph within a specific time window. Monotonic algorithms form a significant category within graph algorithms, characterized by their output R, which changes in a single direction—either monotonically increasing or decreasing—as the graph evolves. Window-based monotonic graph analytics not only holds great significance in graph theory [4, 23, 34], but also plays a foundational role in numerous online analytics operations, such as fraud detection [36, 91] and preference recommendation [31, 96]. However, executing graph analytics within a temporary window can be time-consuming due to the intensive slice merges required. Consequently, there is an urgent need to expedite window-based monotonic graph analytics. To address this problem, this paper introduces an efficient approach for window-based monotonic graph analytics, which involves reusing transitional results for slices within the window.

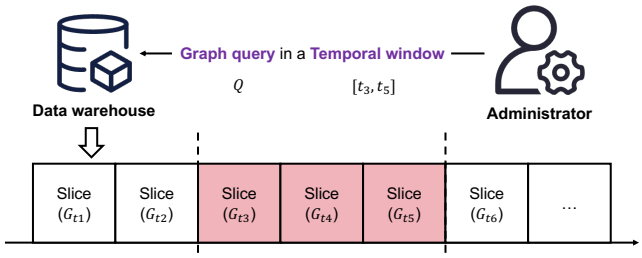


Figure 1: An example of a window-based monotonic graph analytics in Alipay.

We use Figure 1 to demonstrate the significance of window-based monotonic graph analytics. Alipay [3], Ant Group's financial product that serves more than 1 billion users, periodically stores data generated within a certain period (such as a day) in a data warehouse, and queries typically target a slice window. A graph is a set of vertices and edges, with edges connecting the vertices. Data from each period forms a slice, referred to as a subgraph in a graph data warehouse. For example, in a financial graph where users are vertices and transactions among them are edges, managers query the number of connected components within a graph composed of transaction records from the last month. Window-based monotonic graph analytics experience high latency due to the significant overhead involved in constructing the corresponding graph for each

Feng Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681979

uncertain window range and then performing monotonic analytics. Therefore, accelerating window-based monotonic graph analytics is of great importance for its practical use and the service quality of the data warehouse.

In the literature, there are two categories of mainstream graph processing systems to address such types of queries, but neither of them can efficiently handle window graph queries. First, static graph systems [14, 27, 28, 52, 55, 68, 70, 74, 82, 84, 88] require the construction of a separate graph for each query, resulting in significant latency. For example, we evaluate the query of BFS based on 32 subgraphs from the *StackOverflow* dataset. The cost of merging the graph structure accounts for 72.66% of the total time, making this query highly inefficient. Second, evolving/streaming graph systems [4, 13, 17, 37, 56, 57, 61, 66, 71, 73, 81] avoid re-computing the entire graph by pre-storing existing results and performing incremental computations when updates arrive. However, when the scale of a subgraph update exceeds the capabilities of streaming graph systems, their performance suffers. For instance, when inserting an update of a similar scale to a subgraph, its performance is 8.74× lower compared to recomputing on a static graph. Additionally, evolving graph systems operate based on the pattern of updates starting from an initial slice. In this pattern, the initial and updated portions are not treated equally, and the results of the updated portion are not fully leveraged.

Based on real-world graph query workloads and the above analysis, we observe three key insights and opportunities. First, in a data warehouse, data import and subsequent analytics typically do not occur simultaneously. This separation presents an opportunity to precompute the imported slices during periods when the system has idle resources. Second, the workloads consistently demonstrate a regular pattern, which involves querying graphs composed of different time windows in a fixed, regular manner. This pattern allows us to reuse intermediate results across queries targeting various windows. Third, the graph constructed for a specific query window is temporary and does not require persistent storage. As a result, we can avoid the high costs associated with constructing the query graph.

To capitalize on these key opportunities, we propose MergeGraph, a graph analytics system specifically designed for window-based monotonic graph analytics. First, to leverage the asynchronous opportunities between data import and analysis, we develop an offline-online design, which precomputes results within each slice at the time of data import, ensuring they are readily available for immediate use during the online phase. Second, to capitalize on the local results contained in each slice, we propose a *merge-continue-compute* model. It merges transitional results from individual subgraphs to obtain an intermediate result, which is then used to compute the final results, effectively leveraging the transitional results from different subgraphs to avoid redundant computations. Third, to avoid constructing graphs for each query window, we present a computing solution based on discrete subgraphs. We initially construct a discrete graph storage solution that incorporates only indexing for global access. Subsequently, we introduce locality optimizations and parallelism optimizations to enhance the performance of graph computations.

We compare MergeGraph against two state-of-the-art solutions (Ligra [74], Grazelle [9], CoroGraph [98] for static graph processing system, and KickStarter [81], RisGraph [23] for evolving/streaming graph processing system) on four monotonic algorithms: breadth-first search (BFS), single-source shortest path (SSSP), single-source widest path (SSWP), and weakly connected component (WCC). The experimental results on seven datasets demonstrate that MergeGraph achieves an average speedup of 11.30× compared to the best baseline. Specifically, our merge model achieves an average redundant computation savings of 12.49%-94.69% across different window sizes. The graph computing engine based on discrete graph storage achieves an average speedup of 6.71×-25.64×.

To the best of our knowledge, this is the first work specifically dedicated to efficient window-based monotonic graph analytics. We summarize our contributions as follows.

- We propose a *merge-continue-compute* model that equally combines the transitional results from individual subgraphs and performs necessary computations based on the merged result, maximizing the utilization of existing results and avoiding redundant computations across different queries.
- We develop a graph computing framework specifically designed for discrete graph storage, which includes a unified access interface and a computation engine, effectively eliminating the high overhead caused by merging graph structures for window-based graph analytics tasks.
- We conduct extensive experiments to demonstrate that MergeGraph significantly outperforms the state-of-the-art graph computing frameworks in handling window graph queries.

2 BACKGROUND AND PRELIMINARIES

2.1 Window-Based Monotonic Graph Analytics for Pattern-Consistent Query

This paper focuses on window-based monotonic graph analytics, which has been widely used in business operations of commercial companies [2]. This section introduces and formally defines window-based monotonic graph analytics for pattern-consistent queries. All symbols used in this paper are listed in Table 1.

Table 1: Symbol and meanings.

Symbols	Meaning
G_i	The i-th subgraph in a window query.
V	The set of vertex.
E_i	The edge set of the i-th graph.
$Q, Q([i, j])$	The graph query for the window from i to j.
$G_\cup$	The collection of subgraphs within a window.
$E_\cup$	The set of edges in all subgraphs within a window.
R, R_v	The transitional query result for vertices.

Graph, data slice, and subgraph. We define a *graph* as $G = (V, E)$, where V represents the set of vertices, and E represents the set of edges that connect these vertices. In a modern data warehouse, data producers import data into the warehouse at regular intervals, such as once a day. Each day's data constitutes a *data slice*. In this context, the i-th data slice comprises a graph $G_i = (V, E_i)$, where V denotes the vertices and E_i denotes the edges connecting these vertices. We refer to such a graph in a data slice as a *subgraph*.

Window-based query. We perform graph query Q on a collection of data slices within a window. Given a slice window from i to j, we return the query result of $Q([i, j])$ on the graph $G_\cup = (G_i \cup G_{i+1} \cup ... \cup G_j)$. We define the collection of all subgraphs within a window as $G_\cup = (V, E_\cup)$, where we assume that subgraphs share all vertices V, and $E_\cup$ represents the set of all edges among these subgraphs.

Monotonic algorithm. Monotonic algorithms are a significant category within graph algorithms [81], characterized by their output R, which always changes in a single direction – either monotonically increasing or decreasing – as the graph evolves. This principle is widely applied in evolving/streaming graph systems [23, 81]. For instance, when identifying the largest connected component in a graph, the size of the component can only remain constant or increase with the addition of more edges. It will not decrease because adding edges either connects more vertices or maintains the current size of the largest connected component. The assumption of monotonicity ensures that when merging subgraphs, their results consistently evolve in a predictable direction, allowing us to leverage intermediate results.

Pattern-consistent query. We have noticed that numerous daily queries within the data warehouse demonstrate pattern-consistent characteristics, where the identical query is executed on different windows. For instance, this could involve querying the number of connected components in a graph during various time intervals. Despite the changes in window ranges, the query content itself remains unchanged.

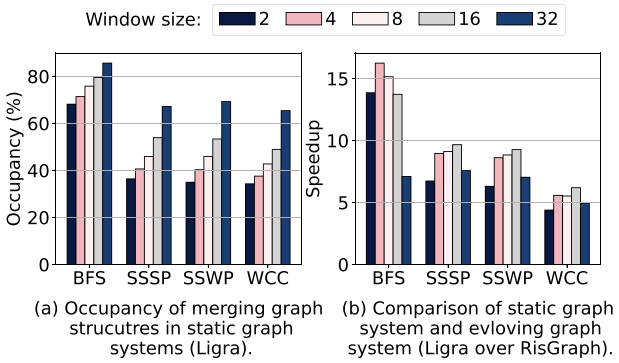
2.2 Iterative Monotonic Graph Analytics

Many graph computing systems [4, 23, 56, 57, 73, 81] support monotonic graph analytics by employing iterative graph computation. In these systems, given a graph $G = (V, E)$, computation begins with one or a group of vertices. It iteratively activates its neighbors based on user-defined functions until the current set of active vertices becomes empty or reaches a user-defined convergence condition. Modern graph computing systems mostly adopt a vertex-centric programming model due to its higher scalability [4, 23, 56, 57, 73, 81].

Algorithm 1: Iterative monotonic graph analytics

Input: Graph G, initial frontier U, update function F, condition function C
Output: Graph query result R

```
1  Create an result array R of size |G.V|;
2  Frontier ← U;
3  while Frontier is not empty do
4      vertexSubset output(|G.V|);
5      for src in Frontier do
6          adj_list ← G.getNeighbors[src];
7          for dst in adj_list do
8              F(src, dst, R);
9              if C(dst) then output.active(dst) ;
10     Frontier ← output;
11 return R;
```



(a) Occupancy of merging graph strucutres in static graph systems (Ligra).

(b) Comparison of static graph system and evloving graph system (Ligra over RisGraph).

Figure 2: Performance of existing systems.

Graph processing workflow. Algorithm 1 illustrates the graph processing workflow. The input consists of a graph G, an initial frontier U, an update function F, and a condition function C. First, we create a result array R of size $G.V$ to store the results for each vertex (Line 1). Then, we assign the initial frontier U to the current *Frontier* (Line 2). After that, starting from the initial frontier, we process all activated vertices in the current frontier (Lines 5-9). For each activated vertex, we scan its neighbors (Line 7), apply the update function F (Line 8) and activate the neighbors if it satisfies the condition function C (Line 9). Finally, we update the *Frontier* to be a vertex subset of vertices activated in this round and repeat this process until the current *Frontier* becomes empty.

3 MOTIVATION

In this section, we first systematically review the limitations of existing solutions in handling window-based monotonic graph analytics, and then highlight our motivation for a new line of research.

Why static graph systems do not apply? In static graph systems, the initial step involves combining individual subgraphs within a specified window range to create a complete graph $G_\cup$ for querying. Subsequently, a graph query Q is executed on $G_\cup$. However, the use of static graph analysis systems necessitates the reconstruction of the entire graph for each query, resulting in significant inefficiency and resource consumption. Moreover, these systems are unable to reuse existing results, leading to redundant computations. We evaluate the performance of Ligra [74], the state-of-the-art static graph system, in handling window-based graph queries on *StackOverflow* dataset with window sizes ranging from 2 to 32. The experiment results are shown in Figure 2 (a). Although Ligra achieves excellent performance in querying static graphs, the cost of graph construction accounts for 34.30%-85.62% of the total query execution time. This renders our efforts to optimize graph systems futile.

Why evolving/streaming graph systems do not apply? Evolving/streaming graph systems adopt the idea of incremental computation, where they start with an initial result maintained from the state of a graph. As the graph updates, they use incremental computation methods to update the results of graph queries, avoiding the need for a full recomputation of the entire graph. For a window query $Q([i, j])$, we consider the portion $[i, k]$, where $i \le k \le j$, as

the initial part and pre-store the computed results for this portion. Then, we treat the slice portion $[k+1, j]$ as the update part of the graph and incrementally compute the final result.

Although evolving/streaming graph systems can leverage the benefits of incremental computation, they do not align well with our query workload. Several graph systems [4, 23, 56, 57, 73, 81] have observed that when the scale of updates surpasses a certain threshold, their performance becomes inferior to recomputing the entire graph. A key contributing factor is the significant overhead associated with inserting updated graph structures into the initial graph in this scenario. In our workload, we treat one or more slices as a single update, and the scale of this update exceeds the benefit threshold of the streaming graph system, resulting in inferior performance compared to static graph systems. For instance, Figure 2 (b) demonstrates the performance of updating one subgraph over another subgraph using the state-of-the-art evolving graph system, RisGraph [23]. The results on the *StackOverflow* dataset indicate that RisGraph's performance is 4.38-16.24 times slower than Ligra.

Motivation for a new line of research. In summary, existing works primarily focus on static and streaming monotonic graph queries, but often neglect the complexities of window-based graph queries that involve multiple slices. This oversight is particularly evident in the substantial computational costs associated with constructing windowed graphs. Such costs can significantly impede the performance of executing windowed graph queries.

To elaborate further, existing works have primarily concentrated on the performance aspects of graph analytics. However, they often overlook the fact that graph analytics workloads in modern data warehouses entail more than merely executing graph computations on a prepared graph. We observe that constructing window-based graphs is the bottleneck in the entire analysis process, implying that merely optimizing graph computation is insufficient to enhance the overall system performance. Therefore, in practical application scenarios of Alipay, we do not rely on existing algorithms and systems due to their limitations. Instead, we develop a novel approach of a window-based monotonic graph analytics solution, which is critical for improving end-to-end system performance. This paper represents the first systematic effort to focus on the comprehensive performance of windowed monotonic graph analytics, encompassing both graph construction and computation.

4 MERGEGRAPH SYSTEM

We propose a system for window-based monotonic graph analytics, called MergeGraph, which enables the reuse of transitional results and direct computation on the discrete graph storage of each slice.

4.1 Overview

We show the overview of MergeGraph in Figure 3. To utilize the opportunities and address the challenges mentioned in Section 3, MergeGraph adopts an offline-online hybrid architecture to achieve efficient window-based graph queries.

Architecture. MergeGraph consists of two main modules: the offline module and the online module. The offline module includes the graph storage within each slice and a computation component

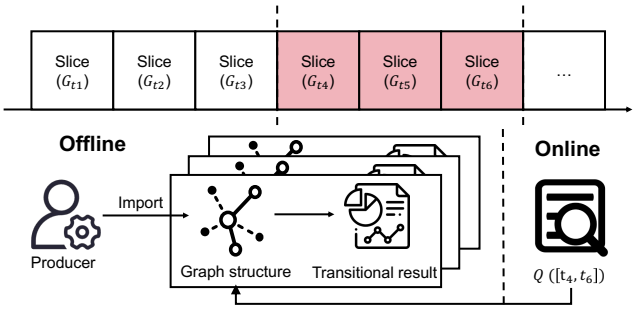


Figure 3: MergeGraph overview.

to obtain the reusable transitional results. The online module comprises a graph analytic engine based on the transitional results and the discrete graph structure.

Algorithm 2: MergeGraph workflow.

Input: Subgraph of each slice $\{G_1, G_2, ..., G_n\}$, Monotonic graph query Q, query window $[start, end]$
Output: Query result R for the window W

```
// Offline phase
```
1 $G_{total} \leftarrow$ set of the graph structure of each slice
2 $R_{total} \leftarrow$ set of the local result of each slice
3 **for** $i \leftarrow 1$ **to** n **do**
4 $\quad G_{total}$.import(G_i);
5 $\quad R_i \leftarrow Q(G_i)$;
6 $\quad R_{total}$.import(R_i);
```
// Online phase
```
7 $G_{\cup} \leftarrow G_{total}[start, end]$ ▷ Load graph for the window
8 $R_{\cup} \leftarrow R_{total}[start, end]$ ▷ Load local result for the window
9 $R_{inter}, frontier \leftarrow init(R_{\cup})$
10 $R \leftarrow compute(R_{inter}, frontier, G_{\cup})$
11 return R

Workflow. Algorithm 2 demonstrates the overall workflow. MergeGraph is structured into two phases: an offline phase and an online phase. In the offline phase, we store the local results of the graph structure for all data slices (Lines 1 and 2). Specifically, whenever a slice is imported into the data warehouse, we first import its graph structure (Line 4), then generate the corresponding local results (Line 5), and save them (Line 6). Note that this step is completed offline during data import, ensuring that it does not affect response time during the online phase. In the online phase, we first retrieve all subgraphs and their associated local results within the window (Lines 7 and 8). Then, based on the results from each slice, we initialize an intermediate result that is closer to the final outcome along with a new frontier (Line 9). Finally, we iteratively compute the final results using this intermediate result, the frontier, and the graph structure within the window (Line 10). During this process, for a specific slice, any window query that includes it can leverage its transitional result R, enabling the reuse of transitional results.

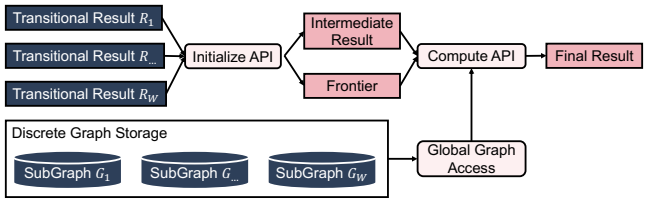


Figure 4: Online window-based graph analytics workflow.

Processing pattern. The processing pattern of MergeGraph for computing monotonic algorithm queries is as follows. In the offline phase, we generate and save corresponding local results for monotonic algorithms, which can be derived through a common iterative graph computation step, as detailed in Section 2.2. In the online phase, it consists of 1) an initialization stage and 2) a computation stage. In the initialization stage of the online phase, we first merge the results saved during the offline phase. In this part, we use a unified initialization API to generate an intermediate result and a frontier for subsequent computations. For monotonic algorithms, we choose the intermediate result from all slices that are closest to the final result as the initial result for the online phase (Section 4.2). For example, for the single-source shortest path, we select the shortest distance for each vertex from all slices, while the single-source widest path does the opposite. In the computation stage of the online phase, we continue the computation based on these generated intermediate results, utilizing a graph computation engine designed for discrete storage. We can categorize different monotonic algorithms as iterative graph computations, for which we have developed access patterns for graph structures based on discrete storage (Section 4.3). The development of the computing engine includes a series of optimizations, including intra-node access optimization and parallelism optimization (Section 4.4).

Novelties. To our knowledge, MergeGraph is the first work focusing on window-based monotonic graph analytics for pattern-consistent queries. We summarize the novelties of MergeGraph as follows. First, MergeGraph introduces an offline-online hybrid framework to leverage precomputed results from individual slices, aiming to enhance performance. Second, MergeGraph proposes a merging model based on reusable transitional results from each slice, which minimizes redundant computations in window-based queries. Third, MergeGraph supports graph analytics on the discrete subgraph structures within each slice, avoiding the high cost of merging graph structures.

4.2 Storage and Utilization of Transitional Result

The core of MergeGraph lies in the storage and utilization of transitional results. By precomputing the transitional result for each slice during the offline phase, we can summarize these results and continue computation based on them, achieving significant computation savings.

Transitional result storage. In MergeGraph, we store the Compressed Sparse Row (CSR) structure of the subgraphs for each slice. This structure is widely used for efficient graph storage due to its excellent analytical performance. During the import of each slice, we utilize the iterative monotonic graph analytics mentioned in Section 2.2 to obtain the results for each vertex within each slice. All these results are stored in the form of arrays.

Transitional result utilization. Figure 4 illustrates the workflow of online window-based graph analytics, which consists of two main stages: the initialization stage and the graph computation stage. For a window-based query consisting of W slices, we first generate an intermediate result and a $frontier$ based on the transitional results of each subgraph. Then, we start from this $frontier$ and intermediate result and iteratively compute the final result. During the graph computation process, we provide a global graph access abstraction on top of discrete subgraphs to support the analysis of the global graph.

Observation. MergeGraph focuses on monotonic algorithms, which approximates the final result from the initial value. For such algorithms, the intermediate results are always closer to the final result than the initial result. Also, for an entire graph composed of multiple subgraphs, the results computed on each of its subgraphs are always closer to the final result compared to the initial result. For example, in the case of the shortest path, the shortest distance of a vertex within a subgraph is always smaller than the initial value and greater than the final result obtained on the entire graph.

Initialization design. Based on the aforementioned observations, we first generate the intermediate result for each vertex that is closest to the final result from the transitional results of each slice. Then, we activate the vertices that may affect the final result and add them to the frontier, which supports subsequent computations. Specifically, for a given vertex v, if its results are inconsistent across different subgraphs, it indicates that its values need to be propagated in other subgraphs, and thus we activate it.

Algorithm 3: Initialization function

Input: Transitional result of each slice $[R_1, R_2, ..., R_W]$
Output: Intermediate result $R_{inter}[V]$, Frontier set $F[W]$

```
1 for each v ∈ V do
2 |   R_inter[v] ← BestValue(R_1[v], R_2[v], ..., R_W[v]);
3 F ← {∅, ∅, ..., ∅};
4 for each v ∈ V do
5 |   for i ← 0 to W do
6 |   |   if R_inter[v] ≠ R_i[v] then
7 |   |   |   F[i].insert(v);
8 return R_inter, F[W];
```

Algorithm design. We show our algorithm design of the initialization function in Algorithm 3. The inputs are transitional results of each slice. For a query with a window size of W, the algorithm takes the transitional results corresponding to each slice as the input. First, we derive the intermediate results for each vertex from the transitional results of each slice, selecting the value closest to the final result (Lines 1-2). Then, we generate W frontier structure (Line 3). For any given vertex v, we examine the transitional results from all the slices. If the result $R_i[v]$ is inconsistent with the best result across all slices, we add v to the corresponding frontier $F[i]$ (Lines 4-7). Finally, the algorithm returns the intermediate results for each vertex across all slices and its corresponding frontier (Line 8).

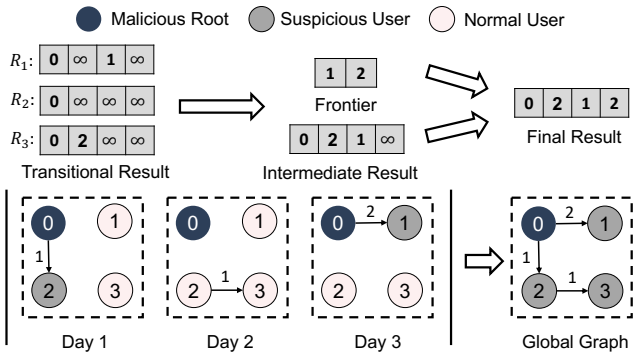


Figure 5: Example of detecting suspicious users whose distance from known malicious users is less than 2 in MergeGraph.

Complexity analysis. The time complexity of executing the monotonic algorithm is $O(V+E_i)$, where V is the number of vertices, and E_i is the number of edges in slice i. We store the transitional result for each vertex whose space complexity is $O(V)$. The complexity of initializing the intermediate results is $O(V \times W)$, where W represents the window size.

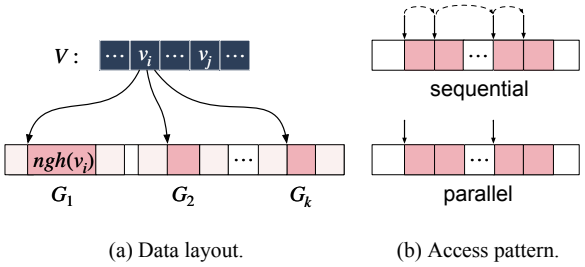
Example. In Alipay within Ant Group, identifying suspicious users in transaction records is a critical business operation that informs our decisions regarding the termination of specific transactions. As shown in Figure 5, we use the shortest distance algorithm to detect suspicious users. Regularly, Ant Group's data warehouse collects daily transaction records. We employ the shortest distance algorithm to identify users who are within a predefined distance from a known malicious user over a recent timeframe.

In the context of Figure 5, assuming a malicious user is identified as 0, we detect suspicious users within a slice window from day 1 to day 3, provided they are within a distance of 2 from user 0. Across the three subgraphs for these days, the local results for the vertices are as follows: $\{0, \infty, 1, \infty\}$, $\{0, \infty, \infty, \infty\}$, and $\{0, 2, \infty, \infty\}$. After initialization, the intermediate result is $\{0, 2, 1, \infty\}$. Given the inconsistency in local results for vertices 1 and 2 across the subgraphs, we add them to the frontier. We then proceed from the intermediate result and the frontier to perform Single Source Shortest Path (SSSP) on the global graph composed of these subgraphs until the computation concludes. Consequently, users 1, 2, and 3 are identified as suspicious, as each falls within the specified distance from user 0 within this time window.

4.3 Pattern-Consistent Query Computation

After obtaining the intermediate results for vertices and the frontier corresponding to each slice, we continue to perform monotonic graph analytics based on discrete graph storage to obtain the final result.

General design. We have shown our design to generate a frontier for each slice in Section 4.2, where each vertex in the frontier indicates that we need to propagate its intermediate result within the corresponding slice. We perform one iterative graph computation based on the frontier of each slice. Then, we aggregate the generated frontiers from each slice to obtain a new unique frontier.



(a) Data layout. (b) Access pattern.

Figure 6: Data layout and access pattern for discrete storage.

Starting from this frontier, we perform iterative monotonic graph analysis on the graph composed of all the slices until the program converges. During this process, we provide a global access interface to enable direct iterative graph analysis based on the discrete graph structure between different slices.

Global graph access pattern. In MergeGraph, the graph structures of individual subgraphs are discretely stored. We utilize an adjacency list to store the outgoing and incoming edges of vertices, a commonly used format in existing graph computing systems [74]. This structure stores the neighbors of each vertex in a contiguous memory segment, with each vertex referencing the starting location of its neighbor list through a pointer. A fundamental proposition of this paper is to circumvent the substantial overhead associated with graph merging by directly executing graph computing on multiple discretely stored graphs. To facilitate this approach, we support a global access pattern that enables seamless access to the graph structure across different subgraphs.

Figure 6 (a) illustrates our data layout. Assuming that each subgraph is stored in different locations, we use a double-layered pointer structure to organize the neighbors of vertices. The first layer of pointers assists in locating the first neighbor of a vertex within its initial subgraph, while the second layer of pointers records the starting positions of the vertex's neighbors across all subgraphs. Furthermore, we maintain the local degree of each vertex in all subgraphs and its global degree in the whole graph, equivalent to the sum of degrees across all subgraphs.

The graph access patterns in MergeGraph are primarily divided into two patterns: *sequential access* and *parallel access*, as shown in Figure 6 (b). In the sequential access pattern, since only one thread handles all neighbors of a vertex, we only need to provide an abstraction from the first neighbor to the last neighbor. We visit all the neighbors of a vertex one by one within each subgraph. During this process, we only need to perform pointer jumps when the neighbor reaches the boundary of a subgraph. In the parallel access pattern, different threads access all neighbors of a vertex. Each thread only retrieves the neighbor at position i of the vertex. Therefore, we first use binary search to determine in which subgraph the $i-th$ neighbors are located. Then, we calculate the corresponding offset within the subgraph and return the neighbor ID.

Complexity analysis. MergeGraph does not change the time complexity of the monotonic graph algorithm during execution, which remains $O(E_\cup)$. The space overhead of MergeGraph for storing the graph includes the edges in each subgraph, and the double-layer pointers point to each vertex. Therefore, the space complexity

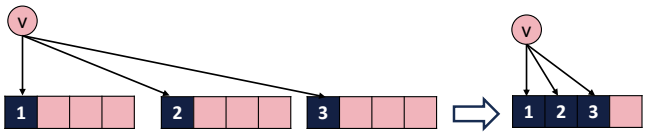


Figure 7: Intra-node access optimization.

of MergeGraph is $O(E_{\cup} + W \times V)$, where $E_{\cup}$ is the total number of edges in all subgraphs, V is the number of vertices, and W is the number of subgraphs. In the sequential access pattern, the average time complexity to access a neighbor is $O(1)$. In the parallel access pattern, the average time complexity to access a neighbor is $O(logW)$, where W is the number of subgraphs. Although the parallel access pattern requires higher access complexity and instruction numbers, it can improve program parallelism. We propose an adaptive selection strategy to choose between these two access patterns.

Correctness proof. We rigorously prove the correctness of MergeGraph. First, we define correctness as the consistency of our algorithm's results with those obtained directly from the original graph composed of windows. For monotonic algorithms, this result is a value related to a user-specified vertex. Second, we employ the dependency tree model, as proposed in the work [81]. In this model, monotonic algorithms designate a single vertex as the root, with each vertex having at most one parent. Consequently, the result at each vertex is influenced by one of its in-neighbors, ensuring that changes from the root to any vertex on the graph follow a single, unique path. Third, consider a vertex v in the graph, and let the path from $root$ to v in the dependency tree be $\{root, u_1, u_2, ..., v\}$. If the value of v has converged following the initialization phase, then its result is inherently accurate. If not, and assuming convergence from $root$ to u_k, our model activates u_k and updates iteratively up to v, thereby confirming the correctness of our approach.

4.4 Optimizing Graph Computation with Discrete Storage

MergeGraph adopts a vertex-centric programming model to perform graph computations. Previous works [74, 99, 100] have proposed a series of optimizations, including parallel schemes and traversal directions, among others. The discrete storage of graph structures brings new challenges, such as the impact on locality and the choice of parallel patterns. This part discusses our optimization for the graph computing engine based on discrete graph storage.

Intra-node access optimization. To optimize the access to all neighbors within a vertex, we take into consideration the impact of no-contiguous memory location on performance. We observe that accessing the neighbors of a vertex in non-contiguous memory locations increases the number of cache loads, ultimately leading to decreased performance. This is because the operating system loads data from a contiguous space in memory based on the cache-line size. When vertex neighbors are stored in different locations, it disrupts the locality of memory access, further hampering performance.

Figure 7 illustrates an example where vertex v has three neighbors in different subgraphs. Accessing them would require at least three cache loads if they are stored separately. However, if we store them together, only one cache line load is needed. In graphs that satisfy the power-law property, these low-degree vertices dominate the graph and significantly impact system performance. Therefore, there is a trade-off between access locality and merging cost. We selectively merge the neighbors of vertices with small degrees and store them in a contiguous memory space to improve performance.

Parallelism optimization. MergeGraph adopts vertex-level parallelism, where each thread processes a different vertex. Additionally, when a vertex has a high degree (greater than 1024), we parallelize the processing of all its neighbors. In a discrete storage architecture, we develop *subgraph-level parallelism*, which allows for the parallel processing of vertex neighbors within individual subgraphs. However, this approach can lead to load imbalances among threads, as the degree of the same vertex can vary significantly across different subgraphs. To tackle this problem, we introduce an *edge-level parallelism* strategy that distributes an equal number of vertex neighbors to each thread. Although this method introduces additional overhead due to the need to identify the storage locations of the neighbors to be processed, it effectively addresses the load imbalance problem. We dynamically select between subgraph-level and edge-level parallelism based on the distribution of vertex degrees across subgraphs.

5 IMPLEMENTATION

We develop MergeGraph using C++ and use OpenMP to support multi-core parallelism on CPUs. This section shows the implementation details.

Discrete graph storage. We store the graph structure in each data slice in an adjacency list format. To meet the requirements of the computation engine, we also maintain the transposed version of the graph. In MergeGraph, we assume that all the graph data from the slices involved in the window-based query has been loaded into memory. We ensure that all slices share the same vertex ID space. In scenarios where this condition is unmet, we align the vertex IDs to achieve consistent access.

Graph computing engine. We develop the graph computing engine based on the discrete graph storage. During the computation stage of the online phase, we adopt APIs similar to those of Ligra [74] to ensure a smooth transition and user-friendly experience with our system. Also, by leveraging Ligra's design for higher-level graph computations, we naturally benefit from its existing optimization techniques. Note that Ligra's optimizations do not include the optimizations mentioned in Section 4.4. The optimization techniques mentioned in Section 4.4 aim to address the performance degradation caused by discrete storage, a consideration that falls outside the scope of Ligra's design. Consequently, these optimizations are not native to Ligra, as Ligra's graph storage mechanism employs a contiguous storage model for the neighbors of each vertex.

Programming API. We provide a user-friendly programming API for MergeGraph, as shown in Table 2. In the *initialization* phase, we use the `init_val` interface to generate the intermediate result based on the local results from each subgraph. Then, we use `need_upd` to determine whether to add a vertex to the

Table 2: MergeGraph's API.

init_val(trans_result[n])	→	init_value
need_upd(trans_result[n])	→	is_needed
edgeMap(frontier, result, graph, F, C)	→	frontier
vertexMap(frontier, F)		
get_out_neighbor(vid, eid)	→	vertex_id
get_in_neighbor(vid, eid)	→	vertex_id
get_out_degree(vid)	→	degree
get_in_degree(vid)	→	degree
get_local_out_neighbors(vid, gid)	→	adj_list
get_local_in_neighbors(vid, gid)	→	adj_list

frontier. In the *graph computation* phase, we adopt a programming interface similar to Ligra [74], including the **edgeMap** and **vertexMap** interface. The **edgeMap** interface applies a user-defined edge function to all neighbors of each vertex in the frontier and adds the neighbors that satisfy the condition function to the returned frontier. The **vertexMap** interface applies a user-defined function to each vertex in the frontier. To support graph computations on discrete subgraphs, we provide a global graph access API, as shown on the bottom of Table 2. In addition to providing a global access API that allows access to a vertex's neighbors across all subgraphs (**get_out_neighbor/get_in_neighbor**), we also offer an API that returns the adjacency list within a single subgraph(**get_local_out_neighbors/get_local_in_neighbors**).

6 EVALUATION

6.1 Experimental Setup

Methodology. We compare two types of state-of-the-art solutions: static graph systems and evolving/streaming graph systems, as shown below.

- **Static graph systems**. We compare MergeGraph against three leading static graph processing systems. (1) Ligra [74]: A lightweight graph processing framework optimized for shared-memory parallel/multicore machines. Ligra excels in many tasks and continues to be actively maintained, demonstrating its robustness and efficiency. (2) Grazelle [30]: A hybrid graph processing framework designed for single-machine, combining high-performance push-based and pull-based methods. (3) CoroGraph [98]: A state-of-the-art graph computing system effectively balances cache efficiency and work efficiency.
- **Evolving/streaming graph systems.** We assess MergeGraph against two cutting-edge evolving/streaming graph systems. (1) RisGraph [23]: A real-time streaming system specially designed for evolving graphs, which supports per-update analysis to handle dynamic graph changes effectively. (2) KickStarter [81]: This system introduces a dependency-tree model that facilitates efficient incremental computation for monotonic algorithms, enhancing performance in dynamic graph environments.

Benchmarks. We employ four widely used graph applications. These applications include Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Single-Source Widest Path (SSWP), and Weakly Connected Components (WCC). Notably, these benchmarks have been used in numerous prior studies [23, 81]. We compare four algorithms across MergeGraph, Ligra, RisGraph, and KickStarter. Grazelle supports only BFS and WCC, while CoroGraph provides SSSP and WCC.

Table 3: Datasets used in the experiments.

Graph Dataset	Abbr.	Vertices	Edges	Type
HepPh[67]	HP	28.1K	4.60M	Collab.
WikiTalk[43]	WT	1.14M	7.83M	Int.
Flickr[67]	FC	2.30M	33.1M	Social.
StackOverflow[43]	SO	2.60M	63.5M	Int.
BitCoin[67]	BC	24.6M	123M	Txn.
LinkBench[7]	LB	128M	560M	Social.
EnWiki[10, 11]	EW	7.7M	1.32B	Int.

Datasets. We assess the performance of MergeGraph using seven distinct graphs outlined in Table 3. Except for the LinkBench dataset, which is generated from the graph database benchmark [7], the remaining graphs are accessible through the Stanford Network Analysis Project (SNAP) [43], Webgraph [11], and Network Data Repository [67]. EnWiki(EW) is a compilation of the English segment of Wikipedia spanning from 2013 to 2023. In EW, the nodes represent Wikipedia pages, and the edges represent the citations and links between the pages. All datasets, except for EW, which is divided based on the year, are segmented into 2/4/8/16/32 subgraphs according to the timestamp on the edge.

Query generation. We generate the queries and slices according to real-world datasets and workloads. First, we collect datasets containing timestamps from the real world. Second, we divide them into different slices based on the timestamps. Third, we execute time window queries on these slices. Regarding the queries, we select four widely used monotonic algorithms, similar to those described in KickStater [81], RisGraph [23], et al. When using the WCC algorithm, we convert the graph into an undirected format. For BFS, SSSP, and SSWP, we randomly select starting points for the queries. To ensure the accuracy of our results and eliminate the influence of randomness, we average the times from three runs for each algorithm to represent its performance.

Platforms. We evaluate MergeGraph on a Ubuntu 20.04 server. The server is equipped with an Intel Core i9-10900X CPU with 20 threads and 256 GB of main memory.

6.2 End-to-End Performance

We evaluate the performance of MergeGraph on four different algorithms. On average, MergeGraph achieves speedups of 11.30×, 7.32×, and 6.17× compared to Ligra, Grazelle, and CoroGraph, respectively. Furthermore, MergeGraph achieves a 45.24× speedup compared to RisGraph and 25.93× compared to KickStarter. KickStarter and RisGraph perform poorly in our experiments because they are unable to handle such large-scale updates. In contrast, static graph systems can directly merge the graph structure of different subgraphs.

Figure 8 depicts the detailed speedup of MergeGraph in comparison to static graph systems. Our observations are as follows. First, Figure 8 (a) demonstrates the performance speedup ratio achieved using the BFS algorithm. Compared to the baseline, we achieve

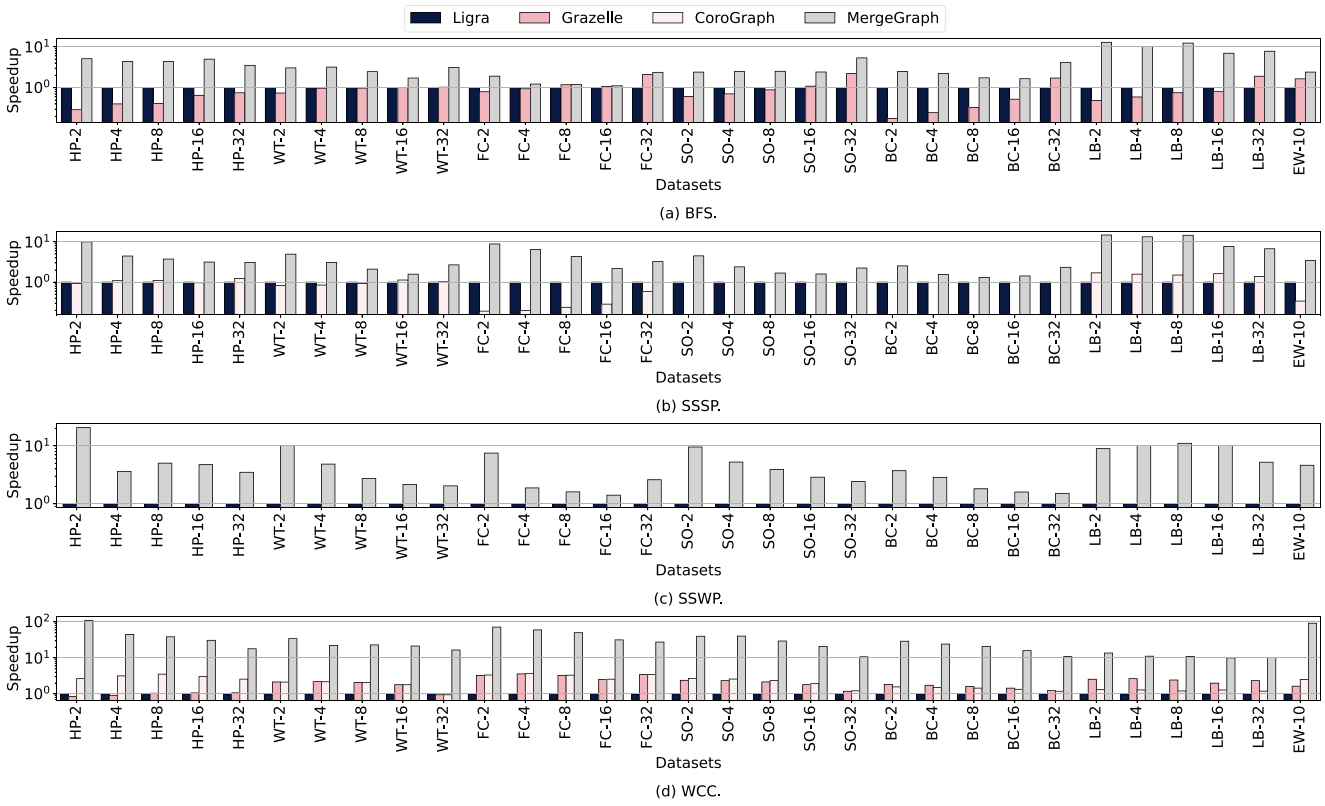


Figure 8: Performance speedup. For the x-axis, *X-Y* represents dataset *X* with a window size of *Y*.

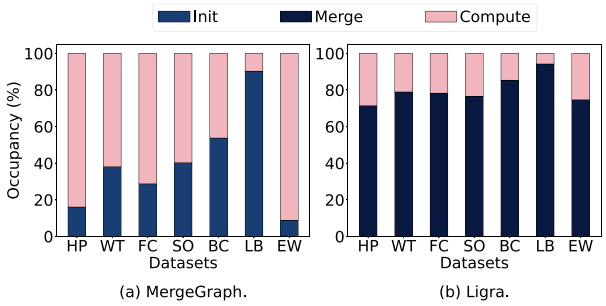


Figure 9: Time breakdown (window size is 16).

an average speedup of 3.80×. Additionally, Figures 8 (b), (c), and (d) show the performance speedup ratios of MergeGraph for the SSSP, SSWP, and WCC algorithms, with respective speedup ratios of 4.56×, 5.10×, and 38.94×. These significant speedups across all benchmarks demonstrate the effectiveness of our approach. Second, the performance speedup gradually diminishes as the window size increases. This decline can be attributed to the decreased potential for computational savings as the window size expands, coupled with the heightened cost of accessing different subgraphs. Consequently, beyond a certain range of window sizes, our method may no longer surpass Ligra. Third, among the four algorithms, WCC attains the highest overall speedup. This is attributed to the WCC algorithm's characteristics, in contrast to the other three single-source algorithms, to effectively leverage the local results (connected components of vertices) from various subgraphs.

Time breakdown. Figure 9 illustrates the time occupancy of different program parts during the execution of MergeGraph and Ligra with a window size of 32. In Ligra, the average percentage of time spent on merging graph structure is as high as 79.9%, while only 20.1% of the time is dedicated to computation. This indicates the inefficiency of Ligra in addressing this type of workload. In contrast, MergeGraph primarily focuses on the computational aspect, with a significant portion of time allocated to the computation phase. On average, the initialization phase constitutes 39.4% of the total time, while the computation phase represents 60.6% of the overall time. The different time allocations highlight the advantages of our approach compared to Ligra. In Section 6.3, we analyze the performance advantages of MergeGraph in different parts.

Memory consumption. We compare the memory consumption of MergeGraph with other baselines on StackOverflow (SO), as detailed in Table 4. On average, MergeGraph not only stores the graph structures within each slice but also retains reusable transitional results, which are dependent on the number of subgraphs within the window. These reusable transitional results constitute only 0.87%-3.36% of the total memory consumption across different window sizes. In contrast, static graph systems require storage

Table 4: Peak memory with different window size (GB). (WS is the abbreviation for *window size*, dataset is SO)

WS	Method				Merge subgraph
	ours	Ligra	Grazelle	Corograph	
2	3.20	6.28	7.66	7.81	5.52
4	4.27	6.75	7.44	7.59	5.30
8	6.74	7.82	8.02	8.17	5.88
16	11.96	10.16	10.16	10.36	8.07
32	21.52	13.85	14.53	14.68	12.39

Table 5: Comparison of active edges in MergeGraph. (WS is the abbreviation for *window size*, dataset is WT)

WS	BFS		SSSP		SSWP		WCC	
	MG	Ligra	MG	Ligra	MG	Ligra	MG	Ligra
2	1.75M		5.03M		2.22M		0.91M	
4	2.13M		9.07M		4.48M		1.81M	
8	2.23M	7.65M	10.2M	14.1M	8.01M	13.8M	3.84M	34.0M
16	2.25M		12.3M		8.11M		7.76M	
32	2.26M		13.9M		8.60M		14.5M	

for the merged graph structure. On average, merging subgraphs occupies 82.09%, 76.26%, and 75.07% of the total system memory in Ligra, Grazelle, and CoroGraph, respectively. Thus, while MergeGraph incurs certain costs to store indices between subgraphs, it achieves lower or comparable memory consumption compared to other methods. Overall, MergeGraph requires 94.69%, 89.81%, and 115.01% of the memory consumption compared to Ligra, Grazelle, and CoroGraph, respectively, to perform window-based monotonic analytics.

6.3 Benefit Breakdown

Computation savings. We compare the total number of active edges during the computation phase between MergeGraph and Ligra on the WikiTalk (WT) dataset, as shown in Table 5. On average, MergeGraph achieves savings of 80.64% / 67.45% / 57.14% / 50.31% / 41.59% in the total number of active edges for window sizes of 2/4/8/16/32. This demonstrates the effectiveness of our merge model. As the window size increases, the number of active edges in our approach gradually increases, while the reusable results decrease. It can be anticipated that as the window becomes larger, our method eventually ceases to provide benefits. Among these four algorithms, the WCC exhibits the most significant savings, amounting to 97.31% / 94.69% / 88.70% / 77.15% / 57.45%. This is because WCC computes possible connected components in all subgraphs, maximizing the utilization of local results. Similarly, this also explains why WCC achieves the best end-to-end performance in Section 6.2.

Graph computing engine with discrete storage. We evaluate the benefits and overhead of our computing engine, as shown in Figure 10. To eliminate the performance impact of computation savings, we perform the same computation tasks as Ligra in this part, with the only difference being the graph storage. We set up two baselines for comparisons. In the first baseline, Ligra computes directly on the complete graph, and we only measure the computation time. The second baseline includes the computation

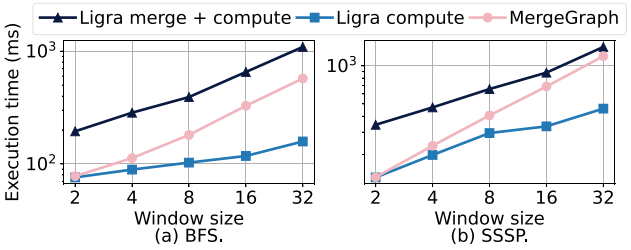


Figure 10: Performance of computing engine.

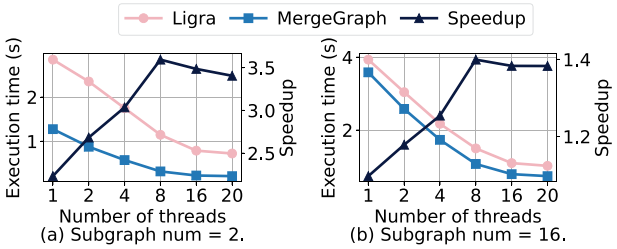


Figure 11: Performance across different numbers of threads.

time and the time required to merge the structures of individual subgraphs. Due to factors such as impaired locality during graph access, the first baseline is the theoretical performance upper limit of our computing engine. On average, MergeGraph is slower than Ligra when considering only computation time, with a difference of 2.49%-72.36% on BFS and 0.35%-61.41% on SSSP. As the window size increases, MergeGraph's runtime grows. However, our method still outperforms Ligra regarding computation and merging time, as the cost of merging graph structures also rises. On overage, our method speeds up the 2.22× on BFS and 1.74× on SSSP.

Performance across different numbers of threads. We evaluate the performance of MergeGraph and Ligra across different numbers of threads, as shown in Figure 11. We measure the performance of SSSP algorithm on the SO dataset with subgraph counts of 2 and 16. The results demonstrate a speedup of 2.23×-3.59× and 1.09×-1.39× across thread counts ranging from 1 to 20. In both settings, MergeGraph and Ligra exhibit similar trends. MergeGraph and Ligra demonstrate similar parallelism because they process the same graph data, and the degree distribution of vertices is consistent, providing comparable opportunities for parallelization. As the number of threads increases, both MergeGraph and Ligra demonstrate a decrease in execution time, indicating their strong parallelism capabilities. The speedup of MergeGraph, compared to Ligra, initially increases with an increase in the number of threads and then decreases. The reason for the increase in speedup in cases with relatively fewer threads is that, for Ligra's approach, the main overhead lies in merging graph structures. Increasing the number of threads does not significantly improve this aspect compared to the performance gains achieved in the graph computation part. After reaching the inflection point, MergeGraph has a lower computational workload compared to Ligra, which results in insufficient utilization of idle threads. As a result, the speedup decreases.

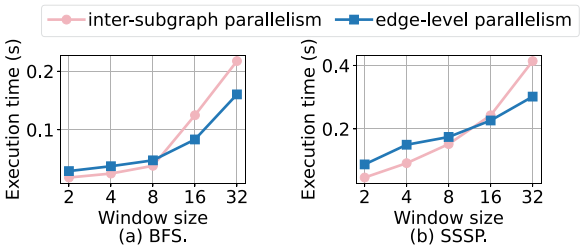


Figure 12: Performance of different parallel strategies.

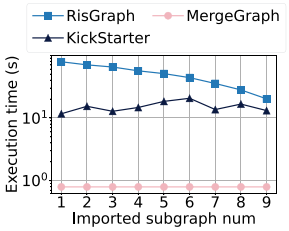


Figure 13: Performance of different initial proportions for streaming graph system.

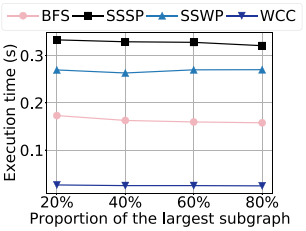


Figure 14: Performance of MergeGraph under different data skews.

Intra-node access optimization. We merge the neighbors of all vertices within each slice whose total degree is less than $cacheline\_size/sizeof(vertex)$ (which is 8 in our configuration) into a new contiguous storage area. The results show that we achieve a performance improvement of 18.73% by merging the neighbors of vertices with small degrees. The merge cost accounts for only 32.54% of the total time.

Parallelism optimization. We evaluate the benefits brought by different parallel strategies. Figure 12 illustrates the performance of BFS and SSSP algorithms on the WT dataset under different parallel strategies. The subgraph-level parallel strategy and edge-level parallel strategy have their respective advantages in different scenarios, highlighting the necessity of dynamically selecting the strategy. For both algorithms, when the window size is small, the subgraph-level parallel strategy performs better. The edge-level parallel strategy outperforms when the window size is large. This is because when the window size is large, the degree distribution among different slices becomes more uneven, allowing the edge-level parallelism to leverage its benefits fully. Overall, the parallelism optimization brings a performance of 72.65% compared to only performing inter-subgraph parallelism and 77.73% compared to edge-level parallelism.

6.4 Detailed Analysis

Different initial proportions in stream graph system. In the streaming graph system, the proportion of initial data significantly impacts performance. We evaluate the performance of BFS algorithm of Kickstarter and RisGraph under different initial data proportions on the Enwiki dataset, which has 10 subgraphs, as shown in Figure 13. We evaluate the performance with initial parts ranging from 1 subgraph to 9 subgraphs. As the proportion of initial parts increases, the execution time of RisGraph gradually decreases by 3.90×. However, even in the case of 9 subgraphs, that is, when the initial proportion is about 90%, the performance of RisGraph is still 25.41× slower than MergeGraph. Kickstarter outperforms RisGraph in large batch updates, whereas RisGraph emphasizes the performance of individual updates. On average, our method is 14.68× to 25.94× faster than Kickstarter, demonstrating the superiority of our approach in handling window queries.

Performance under different data skews. We explore the performance of different subgraphs with varying skewness in a single window query, as shown in Figure 14. Taking the SO dataset with 8 subgraphs as an example, we first set the maximum proportion of different subgraphs in all subgraphs and then randomly divide the size of the remaining subgraphs. Under different data skewness, the performance of MergeGraph does not change significantly. The average difference between maximum and minimum performance on four algorithms is only 5.60%. This indicates that our method has high adaptability in different scenarios.

Performance for very wide time windows. Large window ranges can lead to decreased performance because the proportion from which we can directly benefit from intermediate results diminishes, and accessing too many discrete subgraphs increases the overhead. To tackle this problem, we employ a recursive divide-and-conquer strategy. For a large window range of length W, we recursively partition the window into several sub-windows, continuing this process until the window size meets our operational requirements. Subsequently, we merge the results of each window layer by layer to produce the final result. Experiments show that for large-range queries (on the order of one year), our approach significantly outperforms the baseline, with an average speed improvement ranging between 1.54× and 31.93×.

Space overhead for storing transitional results. Consistent pattern in our paper refers to the practice of querying different windows using the same pattern. In such cases, we store the local results of the query for each slice. For scenarios involving numerous patterns, we save the local results of each query in each slice, which may result in a certain space overhead. However, these results are typically stored according to vertices, meaning their space complexity is $O(V)$, where V is the number of vertices, which is generally much smaller than the number of edges in graph structures. In all datasets, storing local results on average accounts for only 5.48% of the total storage of the slices. Therefore, this cost is still considered acceptable.

6.5 Discussion

Offline precompute overhead. We precompute the transitional results within each slice during the data import phase, which incurs a certain level of system overhead. However, we only compute the transitional result once during data import, which does not affect the latency of online queries. Moreover, we can reuse the transitional result in subsequent window-based queries, effectively amortizing this cost over multiple queries.

Application scope. This work focuses on supporting window-based monotonic graph analytics in a data warehouse. We derive performance gains primarily from two key aspects: First, We save on computation by directly merging results from different data

slices, effectively reducing the need for repeated computations. Second, we minimize the overhead associated with merging graph structures, thereby streamlining the overall process. We cannot consistently guarantee computation savings for standard graph tasks that do not exhibit monotonic changes with subgraph merging. However, even for these standard tasks, we can still capitalize on the benefits of performing computations directly on the discrete graph storage architecture. This approach reduces the costs associated with merging graph structures, thus improving overall performance. We evaluate MergeGraph with the PageRank algorithm, showing that it speeds up by 1.62-1.93× compared to the state-of-the-art baseline. This demonstrates that MergeGraph can also deliver performance improvement on general tasks.

Assumption of sharing vertices. This paper assumes that all slices share the same vertex space, which facilitates the accurate retrieval of local results and neighbors for the same vertex across different slices. In practice, this assumption may not always hold, and typically, a mapping from the actual vertex identifier to a compact vertex ID is employed to align the vertex spaces of different slices. Implementing this vertex ID mapping efficiently is another crucial task [62]. However, such work is orthogonal to our research and does not impact the optimizations we have developed.

Furture work. Although our approach successfully avoids the overhead of merging graph structures, it remains unclear how to combine the local results of individual slices to achieve accurate outcomes while saving computation for non-monotonic graph tasks. Furthermore, this paper assumes that the subgraph in each slice is stored using a standard graph format (i.e., adjacency list). In real world, they might exist in various forms, such as tables or key-value pairs. Extracting graph data from different data stores and integrating it with the work presented in this paper represents a promising direction for future research.

7 RELATED WORK

Static graph system. Static graph systems [14, 16, 27, 28, 52, 55, 68, 70, 74, 82, 84, 88] are specialized for computation on unchanging graph data. Some systems [27, 28, 52] are designed for parallel and distributed computing, while others [68, 74, 82] target specific storage and computation architectures. For example, PowerGraph [27] introduces a sequential greedy heuristic approach to distributed graph placement and representation that exploits the structure of power-law graphs. Ligra [74] is a framework for implementing graph traversal algorithms on shared-memory machines, which implements graph traversal algorithms that operate on subsets of vertices by mapping edges and vertices. Static graphs do not incur data reconstruction overhead, which allows static graph systems to achieve high efficiency through optimized data structures and algorithms. However, they are best suited for scenarios where the graph's structure remains constant, as they are not tailored for evolving graphs.

Evolving & streaming graph system. In contrast to static graph systems, evolving and streaming graph systems analyze dynamic graph data. Evolving graph systems [23, 32, 33, 41, 54, 59, 65, 78, 80] store snapshots of different versions of the graph. They share analysis results among these snapshots to reduce redundant computations and enhance the locality of the analysis process. RisGraph [23] achieves impressive query speed to support fast addition and deletion of edges by designing a new data structure and using space for time. Tegra [34] provides support for performing ad-hoc queries on arbitrary time windows of the graph by introducing an in-memory intermediate state representation and compacting snapshots for arbitrary retrieval. Streaming graph systems [4, 13, 17, 37, 56, 57, 61, 66, 71, 73, 81] maintain a single version of the graph and the results of corresponding queries, which are incrementally updated when a batch of updates is applied. The focus of these works is on incremental computation, i.e., how to update query results efficiently while ensuring correctness. KickStarter[81], GraphBolt[57], and GraphFly[13] use techniques based on tracking dependencies and propagating impact to support efficient queries in the presence of deleted edges. CommonGraph [4] improves the efficiency of querying by converting deletions to additions, enabling shared additions among snapshots, and breaking sequential dependencies in the streaming approach.

General data warehouse. A general data warehouse is a centralized repository that integrates data from various sources and provides a unified view for data analysis and decision-making. It typically supports data storage, processing, and query capabilities for structured, semi-structured, and unstructured data [26, 51, 60, 83, 94]. General data warehouses are designed to handle large volumes of data, provide efficient data retrieval and analysis, and support complex queries and transformations. They are widely used in industry and academia for business intelligence, reporting, and advanced analytics tasks [5, 15, 25, 58, 75, 86]. However, applying general data warehouses to graph data and graph computations is still an emerging area of research. In this paper, we explore a class of important graph-based query workloads in data warehouses, specifically focusing on window-based monotonic graph analytics for pattern-consistent queries.

8 CONCLUSION

This paper introduces MergeGraph, an efficient window-based monotonic graph analytics system that minimizes computational costs during online analytics by reusing transitional results from each slice within the window. We began by carefully examining the limitations of existing systems in handling such workloads and then identified optimization opportunities based on real-world scenarios. Subsequently, we presented a *merge-continue-compute* framework to reduce computational overhead and a graph computation framework based on discrete storage to mitigate the high cost of graph merging. Our experimental results demonstrate that we achieve an average speedup of 11.30× across various monotonic algorithms compared to the state-of-the-art solutions.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 62322213 and 62172419) and Beijing Nova Program (No. 20230484397 and 20220484137). This work is also supported by Ant Group Research Fund. Z. Chen, F. Zhang, Y. Chen, X. Fang, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] 2020. China's 2020 Digital Payment Industry – WeChat Pay vs Alipay. https://thirdbridge.com/chinas-2020-digital-payment-industry-wechat-pay-vs-alipay/

[2] 2021. Top 5 enterprise graph analytics use cases. https://www.techtarget.com/searchbusinessanalytics/feature/Top-5-enterprise-graph-analytics-use-cases

[3] 2023. Alipay. https://www.alipay.com/

[4] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *ASPLOS (2)*. ACM, 133–145.

[5] António Lorvao Antunes, Elsa Cardoso, and José Barateiro. 2022. Incorporation of Ontologies in Data Warehouse/Business Intelligence Systems - A Systematic Literature Review. *Int. J. Inf. Manag. Data Insights* 2, 2 (2022), 100131.

[6] Naheed Anjum Arafat, Arijit Khan, Arpit Kumar Rai, and Bishwamittra Ghosh. 2023. Neighborhood-based Hypergraph Core Decomposition. *Proc. VLDB Endow.* 16, 9 (2023), 2061–2074.

[7] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *SIGMOD Conference*. ACM, 1185–1196.

[8] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (2023), 1548–1559.

[9] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[11] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[12] Lijun Chang, Mouyi Xu, and Darren Strash. 2022. Efficient Maximum k-Plex Computation over Large Sparse Graphs. *Proc. VLDB Endow.* 16, 2 (2022), 127–139.

[13] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. 2022. GraphFly: Efficient Asynchronous Streaming Graphs Processing via Dependency-Flow. In *SC*. IEEE, 45:1–45:14.

[14] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2018. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3 (2018), 13:1–13:39.

[15] Zhida Chen, Gao Cong, and Walid G. Aref. 2020. STAR: A Distributed Stream Warehouse System for Spatial Data. In *SIGMOD Conference*. ACM, 2761–2764.

[16] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–31.

[17] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*. ACM, 85–98.

[18] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, Hongzhi Chen, and Guoren Wang. 2022. Fast Maximal Clique Enumeration on Uncertain Graphs: A Pivot-based Approach. In *SIGMOD Conference*. ACM, 2034–2047.

[19] Yizhou Dai, Miao Qiao, and Lijun Chang. 2022. Anchored Densest Subgraph. In *SIGMOD Conference*. ACM, 1200–1213.

[20] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2022. A Hierarchical Contraction Scheme for Querying Big Graphs. In *SIGMOD Conference*. ACM, 1726–1740.

[21] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing Graph Algorithms. In *SIGMOD Conference*. ACM, 459–471.

[22] Muhammad Farhan, Qing Wang, and Henning Koehler. 2022. BatchHL: Answering Distance Queries on Batch-Dynamic Networks at Scale. In *SIGMOD Conference*. ACM, 2020–2033.

[23] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *SIGMOD Conference*. ACM, 513–527.

[24] Sen Gao, Hongchao Qin, Rong-Hua Li, and Bingsheng He. 2023. Parallel Colorful h-star Core Maintenance in Dynamic Graphs. *Proc. VLDB Endow.* 16, 10 (2023), 2538–2550.

[25] Georgia Garani, Andrey V. Chernov, Ilias K. Savvas, and Maria Butakova. 2019. A Data Warehouse Approach for Business Intelligence. In *WETICE*. IEEE, 70–75.

[26] Kamran Ghane. 2020. Big Data Pipeline with ML-Based and Crowd Sourced Dynamically Created and Maintained Columnar Data Warehouse for Structured and Unstructured Big Data. In *ICICT*. IEEE, 60–67.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. USENIX Association, 17–30.

[28] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. USENIX Association, 599–613.

[29] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD Conference*. ACM, 645–657.

[30] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. *ACM SIGPLAN Notices* 53, 1 (2018), 246–260.

[31] Wei Guo, Chang Meng, Enming Yuan, Zhicheng He, Huifeng Guo, Yingxue Zhang, Bo Chen, Yaochen Hu, Ruiming Tang, Xiu Li, and Rui Zhang. 2023. Compressed Interaction Graph based Framework for Multi-behavior Recommendation. In *WWW*. ACM, 960–970.

[32] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *EuroSys*. ACM, 1:1–1:14.

[33] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *GRADES*. ACM, 5.

[34] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *NSDI*. USENIX Association, 337–355.

[35] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* 16, 7 (2023), 1699–1711.

[36] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. 2022. Spade: A Real-Time Fraud Detection Framework on Evolving Graphs. *Proc. VLDB Endow.* 16, 3 (2022), 461–469.

[37] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *EuroSys*. ACM, 17–32.

[38] Zhiguo Jiang, Hanhua Chen, and Hai Jin. 2023. Auxo: A Scalable and Efficient Graph Stream Summarization Structure. *Proc. VLDB Endow.* 16, 6 (2023), 1386–1398.

[39] Junghoon Kim, Siqiang Luo, Gao Cong, and Wenyuan Yu. 2022. DMCS : Density Modularity based Community Search. In *SIGMOD Conference*. ACM, 889–903.

[40] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. iTurboGraph: Scaling and Automating Incremental Graph Analytics. In *SIGMOD Conference*. ACM, 977–990.

[41] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 29:1–29:40.

[42] Janet Layne, Justin Carpenter, Edoardo Serra, and Francesco Gullo. 2023. Temporal SIR-GN: Efficient and Effective Structural Representation Learning for Temporal Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2075–2089.

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

[44] Faming Li, Zhaonian Zou, Jianzhong Li, Xiaochun Yang, and Bin Wang. 2022. Evolving subgraph matching on temporal graphs. *Knowl. Based Syst.* 258 (2022), 109961.

[45] Faming Li, Zhaonian Zou, Xianmin Liu, Jianzhong Li, Xiaochun Yang, and Bin Wang. 2023. Detecting maximum k-durable structures on temporal graphs. *Knowl. Based Syst.* 271 (2023), 110561.

[46] Jia Li, Wenyue Zhao, Nikos Ntarmos, Yang Cao, and Peter Buneman. 2023. MITra: A Framework for Multi-Instance Graph Traversal. *Proc. VLDB Endow.* 16, 10 (2023), 2551–2564.

[47] Wentao Li, Miao Qiao, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2022. On Scalable Computation of Graph Eccentricities. In *SIGMOD Conference*. ACM, 904–916.

[48] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When Temporal Graph Neural Networks Meet Temporal Personalized PageRank. *Proc. VLDB Endow.* 16, 6 (2023), 1332–1345.

[49] Meihao Liao, Rong-Hua Li, Qiangqiang Dai, and Guoren Wang. 2022. Efficient Personalized PageRank Computation: A Spanning Forests Sampling Based Approach. In *SIGMOD Conference*. ACM, 2048–2061.

[50] Dandan Liu and Zhaonian Zou. 2023. gCore: Exploring Cross-layer Cohesiveness in Multi-layer Graphs. *Proc. VLDB Endow.* 16, 11 (2023), 3201–3213.

[51] Jiesong Liu, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li, Huanchen Zhang, Jidong Zhai, Hechen Zhang, et al. 2024. G-Learned Index: Enabling Efficient Learned Index on GPU. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[52] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2014. GraphLab: A New Framework For Parallel Machine Learning. *CoRR* abs/1408.2041 (2014).

[53] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *SIGMOD Conference*. ACM, 845–859.

[54] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *ICDE*.

IEEE Computer Society, 363–374.
[55] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*. ACM, 135–146.
[56] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: sparsity-aware incremental processing of streaming graphs. In *EuroSys*. ACM, 83–98.
[57] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*. ACM, 25:1–25:16.
[58] Anthony Martins, Pedro Martins, Filipe Caldeira, and Filipe Sá. 2020. An Evaluation of How Big-Data and Data Warehouses Improve Business Intelligence Decision Making. In *WorldCIST (1) (Advances in Intelligent Systems and Computing)*, Vol. 1159. Springer, 609–619.
[59] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Trans. Storage* 11, 3 (2015), 14:1–14:34.
[60] Mukesh K. Mohania. 2001. Building web warehouse for semi-structured data. *Data Knowl. Eng.* 39, 2 (2001), 101–103.
[61] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.
[62] Zhenxuan Pan, Tao Wu, Qingwen Zhao, Qiang Zhou, Zhiwei Peng, Jiefeng Li, Qi Zhang, Guanyu Feng, and Xiaowei Zhu. 2023. GeaFlow: A Graph Extended and Accelerated Dataflow System. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[63] Serafeim Papadias, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Space-Efficient Random Walks on Streaming Graphs. *Proc. VLDB Endow.* 16, 2 (2022), 356–368.
[64] Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. 2023. Computing Graph Edit Distance via Neural Graph Matching. *Proc. VLDB Endow.* 16, 8 (2023), 1817–1829.
[65] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. 2012. Managing Large Graphs on Multi-Cores with Graph Awareness. In *USENIX Annual Technical Conference*. USENIX Association, 41–52.
[66] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.
[67] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. AAAI Press, 4292–4293. https://networkrepository.com
[68] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: scale-out graph processing from secondary storage. In *SOSP*. ACM, 410–424.
[69] Siddhartha Sahu and Semih Salihoglu. 2021. Graphsurge: Graph Analytics on View Collections Using Differential Computation. In *SIGMOD Conference*. ACM, 1518–1530.
[70] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *SSDBM*. ACM, 22:1–22:12.
[71] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Euro-Par (Lecture Notes in Computer Science)*, Vol. 9833. Springer, 319–333.
[72] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed Stream KNN Join. In *SIGMOD Conference*. ACM, 1597–1609.
[73] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *SIGMOD Conference*. ACM, 417–430.
[74] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*. ACM, 135–146.
[75] Dominik Slezak, Jakub Wroblewski, Victoria Eastwood, and Piotr Synak. 2008. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proc. VLDB Endow.* 1, 2 (2008), 1337–1345.
[76] Yahui Sun, Shuai Ma, and Bin Cui. 2022. Hunting Temporal Bumps in Graphs with Dynamic Vertex Properties. In *SIGMOD Conference*. ACM, 874–888.
[77] David Tench, Evan West, Victor Zhang, Michael A. Bender, Abiyaz Chowdhury, J. Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. 2022. GraphZeppelin: Storage-Friendly Sketching for Connected Components on Dynamic Graph Streams. In *SIGMOD Conference*. ACM, 325–339.
[78] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic Algorithm Transformation for Efficient Multi-Snapshot Analytics on Temporal Graphs. *Proc. VLDB Endow.* 10, 8 (2017), 877–888.
[79] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-truss Search in Dynamic Directed Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2199–2211.
[80] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic Analysis of Evolving Graphs. *ACM Trans. Archit. Code Optim.* 13, 4 (2016), 32:1–32:27.
[81] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*. ACM, 237–251.
[82] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX Annual Technical Conference*. USENIX Association, 507–522.
[83] Jingting Wang and Bao Liu. 2020. Design of ETL Tool for Structured Data Based on Data Warehouse. In *CSAE*. ACM, 119:1–119:5.
[84] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement - Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *USENIX Annual Technical Conference*. USENIX Association, 387–401.
[85] Zhigang WANG, Ning WANG, Jie NIE, Zhiqiang WEI, Yu GU, and Ge YU. 2023. A lock-free approach to parallelizing personalized PageRank computations on GPU. *Frontiers of Computer Science* 17, 1, Article 171602 (2023), 171602 pages. https://doi.org/10.1007/s11704-022-1546-2
[86] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing. In *SIGMOD Conference*. ACM, 2797–2800.
[87] JianXuan Wu, Xiangnan He, Xiang Wang, Qifan Wang, Weijian Chen, Jianxun Lian, and Xing Xie. 2022. Graph convolution machine for context-aware recommender system. *Frontiers of Computer Science* 16, 6, Article 166614 (2022), 166614 pages. https://doi.org/10.1007/s11704-021-0261-8
[88] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: scaling graph computation to the trillions. In *SoCC*. ACM, 408–421.
[89] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient Triangle-Connected Truss Community Search In Dynamic Graphs. *Proc. VLDB Endow.* 16, 3 (2022), 519–531.
[90] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2023. Scalable Time-Range k-Core Query on Temporal Graphs. *Proc. VLDB Endow.* 16, 5 (2023), 1168–1180.
[91] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *SIGMOD Conference*. ACM, 2348–2356.
[92] Haoteng Yin, Muhan Zhang, Jianguo Wang, and Pan Li. 2023. SUREL+: Moving from Walks to Sets for Scalable Subgraph-based Graph Representation Learning. *Proc. VLDB Endow.* 16, 11 (2023), 2939–2948.
[93] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *SIGMOD Conference*. ACM, 860–873.
[94] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. POCLib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE transactions on Parallel and Distributed Systems* 33, 2 (2021), 459–475.
[95] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines. *Proc. VLDB Endow.* 16, 11 (2023), 2728–2741.
[96] Ziwei Zhao, Xi Zhu, Tong Xu, Aakas Lizhiyu, Yu Yu, Xueying Li, Zikai Yin, and Enhong Chen. 2023. Time-interval Aware Share Recommendation via Bi-directional Continuous Time Dynamic Graphs. In *SIGIR*. ACM, 822–831.
[97] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2239–2247.
[98] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *Proceedings of the VLDB Endowment* 17, 4 (2023), 891–903.
[99] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.
[100] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. {GridGraph}:{Large-Scale} Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.
[101] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proc. VLDB Endow.* 16, 9 (2023), 2172–2185.
[102] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *Proc. VLDB Endow.* 16, 10 (2023), 2645–2658.