Morphoses: A General Graph Stream Fine-Grained Tracking System

Litao Lin
National Engineering Research Center for Big Data Technology and System
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, China

Abstract

Graph streams produce real-time dynamic graphs, which are becoming increasingly important. Many applications need tracking of the update transactions that produce changes in the results of graph algorithms. However, there are two issues with limiting the tracking granularity to a single graph update: (1) The incremental computation pace is substantially slower than the graph stream update speed. (2) Tracking essential transactions necessitates computing a large number of duplicate results in a sequential manner. The complete materialization of iterative results restricts the computing performance of present incremental solutions in the graph stream tracking issue. Computing the results of transactions one by one leads to duplication for some objectives. We propose Morphoses, a general graph stream tracking system. We present a *Vertex State Set* that allows for completely incremental data structure and iterative algorithm processing. We offer a divide-and-conquer strategy based on the graph stream's micro-batch to swiftly track the related key transactions.

1 Introduction

Graph computing based on real-time dynamic graphs generated by graph streams is playing an important role in exploring time series and associated data, and is widely used in finance, telecommunications, and public healthcare. Due to the large scale and high rate of graph streams, it is difficult to run a complete iterative algorithm on real-time dynamic graphs, so recent research has focused on incremental analysis (or evolutionary graph analysis).

In many cases, the application needs to monitor the changes brought by the graph update to the iterative results of the algorithm. When a change in the result that meets the conditions is found, it is necessary to further track which update transaction caused the change. For example, in financial anti-fraud, risky transactions [1] generate temporal edges between account vertices. These updates affect the risk control program based on the Single Source Shortest Path (SSSP) algorithm. When the distance between an account and a known risky account is shortened, the program reports the change in the algorithm result and tracks which transaction caused the distance change. Related applications include community relationship analysis [3, 9], web page popularity ranking [11], etc.

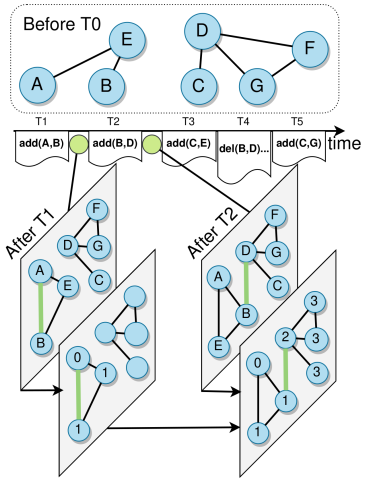

Figure 1: An Example of Graph Stream Tracking.

In such applications, a graph stream is defined as an infinite sequence of graph update transactions, which may contain temporal edges arriving at the same moment, or specify simultaneous interactions. As shown in the Figure 1, a transaction updates the real-time dynamic graph aggregated within a window, and the associated graph algorithm results also change. Before transaction T_2 occurs, the vertices on the left and right sides of the graph are not connected, so the distance from some points on the right to vertex A is ∞. After T_2, the right vertex is connected to vertex A, and the distance is shortened to a constant. The graph stream tracking system needs to passively calculate the difference between the latest result and the previous result, and track the cause of the result change

[1] https://github.com/CGCL-codes/Morphoses

when the difference meets the conditions.

Existing graph incremental processing systems can also be used for graph stream tracking. The straightforward approach is to compute a new result after each update, and then compared to the previous one. However, existing systems face two difficulties when the tracking granularity is reduced to a single update. First, the speed of calculating algorithm results is much lower than the update speed in the graph stream. In most cases, the system cannot complete the incremental calculation of a new result within 1 second, and the generation speed of common graph stream transactions is much higher than this (up to millions per second). Second, updates in graph streams are frequent, but it is difficult to report changes caused by each edge update. The real-time nature of the graph stream tracking system requires reporting as soon as possible after the transaction occurs, and it is processed completely online.

We propose Morphoses, a general graph stream tracking system. Morphoses performs incremental processing based on iterative process retention. Unlike previous methods, it proposes *Vertex State Set* to implement full incremental processing of data structures and iterative algorithms. The vertex state set avoids the complete materialization of the iterative results of the algorithm and unnecessary vertex state scanning. Morphoses can flexibly reuse the history of the previous iteration or the current iteration in the previous frame in both directions. The results are retained to narrow the scope of incremental recalculation. At the same time, Morphoses organizes multiple consecutive graph stream transactions into micro-batches for processing, and uses timing multi-version graph storage [19] to parallel computations within micro-batches. Based on this, we propose a divide-and-conquer method to quickly track key transactions that cause results to change.

In summary, we make the following contributions:

(1) Define the graph stream tracking problem and point out the challenges under different tracking granularity.

(2) We design and implement the vertex state set, use it to completely increment the iterative algorithm and data structure, avoid the materialization of the complete result, and improve the computational efficiency;

(3) We propose and implement a divide-and-conquer method for graph stream tracking based on the micro-batch method, which effectively achieves scaling.

The other sections are organized as follows. Section 2 formally define the graph stream tracking problem. Section 3 presents fully incremental processing based on the vertex state set. Section 4 presents design of Morphoses. Section 5 introduces the related work.

2 Preliminaries and Problem Formulation

2.1 Preliminaries

The graph stream is an unbounded timing edge sequence, forming a large-scale real-time dynamic graph. Due to the unbounded nature of the graph stream, analysis often use the sliding window model to aggregate edges within a certain time range. The edges that meet the window requirements form a real-time dynamic graph, and the incoming edges or interactions are converted into updates to the graph. The continuous updates are composed of a series of transactions. In each transaction, the system processes simultaneous incoming edges or interactions that are designated to occur at the same time. Because each edge update may become a transaction, the large-scale nature of the graph stream determines it needs to process a large number of transactions in a short time.

Table 1: Notations.

Notations	Description
T_k	The k-th graph stream update transaction
$\mathcal{A}$	Graph iterative algorithm
$G_k, \Delta G_k$	The graph after T_k and corresponding update
$R_G^k, \Delta R_G^k$	Convergence result of the algorithm and update corresponding to T_k
$I_i^k, \Delta I_i^k$	The i-th iteration of the algorithm on G_k and update corresponding to T_k
$m_{(s,v)}^i$	The i-th iteration message from vertex s to v
(I_i^k, I_j^l)	The vector from left state to right
E_i^k	The state vector sum of T_k in iteration i
x_v^i	The i-th iteration state of vertex v
$\oplus, \ominus$	Vertes state set operator

Consider the real-time dynamic graph $G = (V, E)$ formed by the graph stream, where V is the vertex set and E is the edge set. The sequence of transactions T_k cause changes to the structure of G. The graph after each transaction can be denoted as snapshot G_k and so on. For any vertex-centered graph algorithm A, we compute $S^i(G_k, I_k)$ start from a set of initial values denoted as I_k, converging on $S^*(G_k, I_k)$. Table 2 lists the notations used in this paper.

2.2 Graph Stream Tracking Problem

Formally, we use $G = (V, E)$ to represent the real-time dynamic graph formed by the graph stream, where V is the vertex set and E is the edge set. The transaction number of the graph update is logically incremented.

Assume that the current moment τ is between the transaction T_k and the transaction T_{k+1}. Using the transaction number, we denote the current graph transient state as G_k, that is, the k-th frame. The graph algorithm of a vertex-centric model denoted $\mathcal{A}$ runs on G_k iteratively, and the convergence result obtained is R_G^k. At this time, the transaction to be executed is T_{k+1}, which may contain multiple simultaneous interactive operations or only one edge. The update of T_{k+1} will cause some changes in the graph, denoted as ΔG_{k+1}. Since the new frame G_{k+1} after the transaction update is different from the original frame G_k, for the same graph algorithm $\mathcal{A}$, different

convergence results R_G^{k+1} may be obtained. In order to describe the difference between convergence results, we define two operators $\oplus$ and $\ominus$. The difference between R_G^k and R_G^{k+1} can be expressed in the following form. Obviously, for graph algorithm $\mathcal{A}$, the impact of ΔG_{k+1} is ΔR_G^{k+1}.

$$R_G^{k+1} = R_G^k \oplus \Delta R_G^{k+1},\ \Delta R_G^{k+1} = R_G^{k+1} \ominus R_G^k. \quad (1)$$

In the graph stream tracking problem, for graph transient G_k and graph algorithm $\mathcal{A}$, we have the update sequence ΔG_{k+1}, ΔG_{k+2}, ΔG_{k+3}, $\cdots$, and obtain the corresponding $\Delta R_G^{k+1}, \Delta R_G^{k+2}, \Delta R_G^{k+3}, \cdots$ to track the impact of specific graph update on the convergence result of the algorithm. We can increase granularity of the graph stream tracking. For example, if we consider updates ΔG_{k+1} and ΔG_{k+2} together, the impact on the result will be $\Delta R_G^{k+1} \oplus \Delta R_G^{k+2}$ which can be computed as a whole rather than two items. However, the fine-grained graph stream tracing requires ΔR_G^k corresponding to ΔG_k caused by atomized update transaction T_k. So that, the tracking granularity is the smallest, because the transaction may only include the update of a single edge.

2.3 Tracking Granularity

A key parameter of the graph stream tracking problem is the *tracking granularity*. While graph streaming requires the system to complete tracking computations as soon as possible after a transaction update, tracking security will be lost if the moment is too far from the last computed frame. Tracking safety refers to calculating the results of two frames. From the results, the state change of all vertices during the period can be detected, so as to determine whether the tracking conditions are met, as Definition 2.1.

Definition 2.1 [Tracking Safety of Graph] In the graph stream sequence *Seq*, snapshot G_s with tracking safety relative to the base snapshot G_b holds:

$\forall v_i \in \bigcup_b^s V_{G_k}$, $\exists G_d$ in the sequence segment $[G_b, G_s]$, $\forall b \le m \le d, S_{G_m}^{*v_i} = S_{G_b}^{*v_i}$ and $\forall d < n \le s, S_{G_n}^{*v_i} = S_{G_s}^{*v_i}$. The absence of a vertex is a special state.

Without tracking safety, the system may miss some important state changes. So safe tracking granularity is a small distance from the base frame, the smaller the granularity, the safer the tracking, as defined 2.2. When the granularity is 1 transaction, the graph stream tracking calculation is safe absolutely.

Definition 2.2 [Tracking Granularity] The tracking granularity of the base snapshot G_b is a segment $[G_b, G_e]$ on graph stream sequence *Seq*, $\forall b \le m \le e$, G_m satisfies tracking safety, and $\forall n > e$, G_n does not satisfy tracking safety.

Safe tracking also has a feature that the state change at any vertex can locate a specific transaction, such as Theorem 2.1.

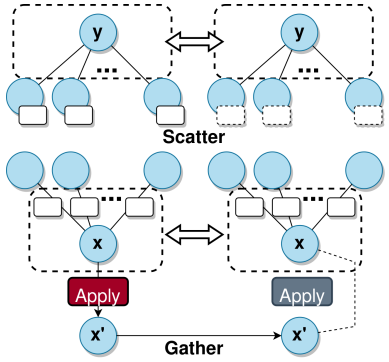

Figure 2: Gather and Scatter.

Theorem 2.1 In the tracking granularity, $\forall S_{G_k}^{*v_i} \neq S_{G_b}^{*v_i}$ can be located to a specific transaction T_d.

3 Incremental Fine-Grained Tracking

3.1 Fine-Grained Retention Replay

A simple form of graph parallel abstraction is a vertex-centric model that runs algorithm programs in parallel at each vertex. Using synchronous processing semantics, the development of vertex-centric parallel graph algorithms is not only more convenient, but also easy to ensure the correctness and convergence of the algorithm. The Gather-Apply-Scatter (GAS) model [13] is a popular vertex-centric model that has been used by real systems. In this model, the graph algorithm is divided into several iterations to run until the convergence result is obtained. Each iteration can be divided into three logical stages, that is, Gather collects neighbor information, Apply updates the vertex state, and Scatter transforms the message.

Based on the iterative retention technology, we save the complete process of the algorithm until convergence. Specifically, the system materialization and save the vertex state set of each iteration. Because adjacent frames are only slightly different due to transaction execution, ideally the next frame can reuse all the iterative results of the previous frame. But in order to ensure the correctness of the algorithm while reusing the state as much as possible, we must carefully consider the conditions of reuse and the calculation process of merging the results of reuse.

For GAS iterative parallel abstraction, a round of iterative calculation mainly includes a three-stage process of G/A/S for each vertex. In the gather phase, each vertex collects messages sent by neighbors and calculates a new state based on its own state. As shown in Figure 2, if the state of the vertex is the same as the previous frame, and the received message has not changed, it can be directly derived that its new state is the same as the state calculated in the previous frame. At this

time, the system can skip the gather stage of the vertex and reuse the result of the previous frame. Conversely, if there is any difference between the vertex state and the received message, the state should be recalculated. If the result of the previous frame is reused in the Gather stage, the system can ignore the status update in the Apply stage. In the Scatter stage, each vertex generates a propagation message based on its own state and neighbor list.

Without loss of generality, we discuss the most basic case of two consecutive iterations of two adjacent frames. Since the $k+1$ frame is only slightly different from the k frame in the structure of the graph, if a vertex is in the state of I_i^k (representing the i iteration of the k frame), The neighbor message is the same as I_i^{k+1}, then the vertex state of I_{i+1}^k can be multiplexed to I_{i+1}^{k+1}. For vertices with the same state of I_i^{k+1} and I_i^k, we can use the result of I_i^k materialization by default, and only distinguish between I_i^{k+1} and I_i^k differences.

The difference between I_i^{k+1} and I_i^k is mainly divided into the following three categories:

Graph structure difference. The difference in the structure of the graph between the $k+1$ frame and the k frame is caused by the transaction T_{k+1}. If T_{k+1} inserts vertices, then $k+1$ frame may have more points than k frames. If T_{k+1} inserts or deletes edges, the neighbors of some vertices may change.

Vertex state difference. Due to the difference in graph structure between the $k+1$ frame and the k frame, for the i round of a particular graph algorithm, the state of each vertex may be different. For example, different rank values are calculated in the PageRank algorithm, and different tags are obtained in the label propagation algorithm.

Messages difference. For a particular algorithm, different vertex states and different neighbors may cause differences in the propagation of messages. For example, the number of neighbors in the PageRank algorithm affects the value of propagation, and different messages are sent according to different vertex labels in label propagation. Among them, the status difference is the most important, because it will change every iteration and directly cause the message difference.

3.2 Incremental Tracking

In Section 3.1, we explained that the differences between two adjacent frames are divided into three categories: graph structure differences, vertex state differences, and received message differences. Both vertex state recalculation and message recalculation are directly caused by these three types of differences. For state recalculation, the difference in graph structure will directly lead to the change of the received message, and the content of the message sent by the neighbor vertex will also be affected by the state of the neighbor vertex and its neighbors. For message recalculation, the difference in graph structure changes the neighbor list of the vertex. In some algorithms, the message needs to be recalculated (for example, the pagerank algorithm). If the state of the vertex changes, the message it sends must be recalculated. Between two adjacent frames, the difference in graph structure is caused by the transaction, which has an impact on all iterations of the algorithm. The status and message belong to a specific iteration round, and it only affects one iteration.

The process from I_i^k to I_{i+1}^k is the process of running one iteration of the k-th frame. Each vertex propagates the message according to the state of round i, and receives the message in round $i+1$, and calculates The new state of round $i+1$. The process of generating I_{i+1}^{k+1} is described in detail as follows.

In **Scatter**, if the state of a vertex at I_i^{k+1} is the same as the state of I_i^k, and the neighbors do not change, the messages it propagates are all the same, which is the same as the example in Figure 2. The state x and the set of adjacent edges must produce the same message M. For the same message, the system does not need to propagate when calculating I_{i+1}^{k+1}. The system will only send messages that are different from I_i^k, which includes message recalculations caused by different states of I_i^{k+1} and I_i^k and different neighbors.

In **Gather**, first, the vertex will only receive messages different from I_i^k in the neighbor nodes. As long as a different message is received, it means that the state recalculation may produce a state different from I_i^k. The vertex needs to actively collect all messages belonging to I_i^{k+1} from all its neighbor vertices.

In **Apply**, if the state of a vertex at I_i^{k+1} is the same as the state of I_i^k, and the number and content of messages received in the Gather stage are all the same as I_i^k, then the state of I_{i+1}^{k+1} is the same as I_{i+1}^k. Otherwise, the state of I_i^{k+1} is used to recalculate the new state with the messages collected in the Gather stage.

In the micro-batch computing model, a large number of frames are included in the micro-batch. For the same graph algorithm, the convergence rounds of different frames may be different. For example, in the label propagation algorithm, frames with extra (and vice versa) edges will converge faster (and vice versa). In order to judge whether the algorithm converges, the difference between the current iteration i of each frame and its previous iteration $i-1$ must be obtained. If the previous frame of the current frame k takes fewer rounds to reach convergence, when calculating the last iteration of frame k, there is no history of the previous frame as a reference, and the system has to materialize the complete result of each iteration. This also relies on iterative calculations from previous rounds.

In order to achieve the convergence of the judgment calculation and the iterative calculation when there is no history, the system needs to maintain the difference between the $i-1$ round and the i round. However, a round of iteration is likely to modify the state of a large number of nodes, making the modulus length of the state vector from R_{i-1}^k to R_i^k very large.

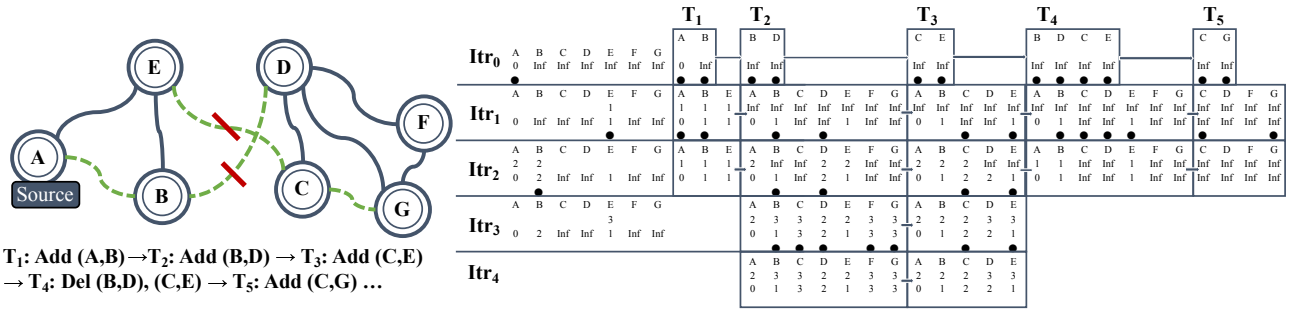

Figure 3: An example describing the state vector's frame-by-frame disparity. The real-time dynamic graph contains A-G vertices, with solid lines representing edges that already exist in the graph. $T_1 - T_5$ is a continuous update transaction, the Add edge represents the arrival of a new timing edge, and the Del edge represents the interactive transaction of the delete edge. The state of the real-time dynamic graph after each transaction ends is represented by $T_1 - T_5$. In the graph, the single-source shortest path algorithm iteration with A as the source is continuously performed, and each edge is regarded as undirected and has a weight of 1. The calculation process of 5 transactions is shown on the right side of the figure. The T_0 transaction is preceded by a fully materialized historical iterative process, including $Itr_0 - Itr_3$. Subsequent computations are based on the full historical iterative incremental representation. The implementation box represents the state vector, with four rows representing vertexId, recvMsg, newState and sendMsg.

In order to solve this problem, we can derive (R_{i-1}^k, R_i^k) from the state vector of $(R_{i-1}^{k-1}, R_i^{k-1})$ in the previous frame.

$$(R_{i-1}^k, R_i^k) = (R_{i-1}^{k-1}, R_i^{k-1}) \oplus (R_i^{k-1}, R_i^k) \ominus (R_{i-1}^{k-1}, R_{i-1}^k). \quad (2)$$

It is not difficult to see that the difference for the k-th frame of the fixed i round is accumulated from the difference of the $k-1$ round. We can define the cumulative sum of vectors starting from the first frame of the microbatch.

$$E_i^k = (R_{i-1}^0, R_i^0) \oplus \sum_{j=1}^{k} ((R_i^j, R_i^{j+1}) \oplus (R_{i-1}^{j+1}, R_{i-1}^j)). \quad (3)$$

In order to obtain the vector accumulation sum $Sum(i,k)$, the system needs to maintain a map with the same capacity as the maximum number of vertices in all frames in the microbatch, and save the state change record for each vertex. In the first frame of the micro-batch processing, $Sum(i,0) = (R_{i-1}^0, R_i^0)$, which comes from the difference between two consecutive iterations retained in history. With the frame-by-frame calculation in round i, the (R_i^{k-1}, R_i^k) obtained in each frame is added to $Sum(i,k)$. Since R_i^{k-1} is the result of frame $k-1$ in round i, for For any vertex v, either there is no state vector yet, or the right end state of the state vector is the same as the left end state of (R_i^{k-1}, R_i^k). The $\oplus$ operation of the vector only needs to add a new vector to the vertex v, or modify the right end state of the old vector to the state in R_i^k. After that, we need to accumulate $\ominus(R_{i-1}^{k-1}, R_{i-1}^k)$. The opposite state vector is obtained by swapping the left end state and the right end state of all vertices, and the $\oplus$ operation is the same. It can be seen from the calculation process that the system cannot determine the set of vertices with non-empty state vectors from the cumulative vector sum, unless the entire map is scanned. The cumulative vector sum is calculated strictly in one line and frame by frame. Frames can be missing, but there is no way to go back to the previous frame.

The cumulative vector sum in frame k represents the difference between the current iteration i and the previous iteration $i-1$, and the system can judge whether the frame has converged or not. The condition of convergence is that the modulus length of the cumulative vector sum is 0, that is, no vertex in the map has a state vector. Although we cannot determine the set of vertices of a non-empty state vector from the cumulative vector sum, the number of vertices with a non-empty state vector can be determined in real time. When we add a new vector to a vertex v, the modulus length increases by 1. Transforming the right end state of the existing vector will not change the modulus length, unless the modified right end state is the same as the left end state, the vector is removed from the vertex v, and the modulus length is reduced by 1.

Another function of cumulative vector sum is to switch to iterative calculation when there is no history. When the $k-1$ frame has converged in the i-th round of calculation, but the k frame has not converged, the complete materialization and accumulation vector of the first frame can be used to materialize the frame result.

$$R_i^k = R_i^0 \oplus Sum(i,k). \quad (4)$$

Due to the need to completely materialize the result of a frame, all vertices will inevitably be scanned. During the scanning process, the right end state of the current cumulative vector sum is merged with the state of the first frame and

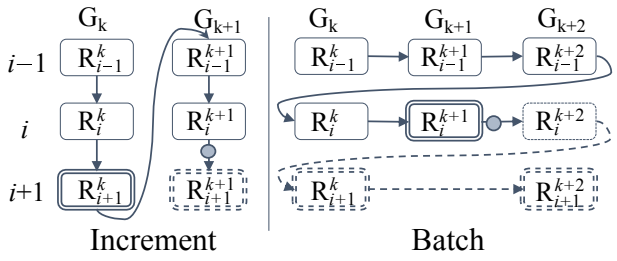

Figure 4: Micro-Batch Computation.

materialized into a new continuous storage space. The current cumulative vector sum is still valid, it skips the frame and continues to accumulate.

The micro-batch calculation ends after all the iterations of all frames have reached convergence. At this time, the state vectors generated by the multiple iterations of all frames are stored in the memory, and the system also saves the cumulative vector sum of all iterations of the last frame. After the system calculates and outputs the difference between the results of each frame and the previous frame according to the state vector, the micro-batch processing is completely finished. Two issues need to be considered at this time. The first is to release the intermediate results generated by this micro-batch calculation, and the second is to materialize the calculation results of the current micro-batch so that it can be used at the beginning of the next micro-batch processing.

3.3 Micro-Batch Parallel Model

In order to meet the high-throughput computing requirements of fine-grained tracking of graph streams while retaining the iterative process, we propose a micro-batch processing model. A micro-batch caches multiple consecutive update transactions, and the large-scale nature of the graph stream makes a short-term micro-batch contain a large number of frames to be calculated. This method is of practical significance, and multiple stream processing systems are designed based on this spirit. However, micro-batch processing has unique significance in the problem of graph stream tracking.

When there is no micro-batch processing, the iterative process retention technology needs to calculate each transaction separately, starting from the first iteration until convergence. Each iteration needs to calculate the difference with the retained part, and determine the range that needs to be recalculated in the next iteration. After the calculation of this frame is completed, the new iterative process and the remaining iterative process are replaced, and the new iterative process is completely materialized, so as to continue processing the next frame. As shown in Figure 4, from the overall point of view, the calculation process is carried out vertically one by one. By constantly comparing the differences, the incremental calculation of the column in the first frame is completed, and the iterative process of this column is completely materialized, replacing the previous retention. The calculations for each subsequent frame are performed in this way. The vertical calculation process is inconsistent with the inherent requirements of graph stream tracking. Graph stream tracking focuses on the changes of real-time dynamic graphs, while a column of calculations produces a complete result of one frame, and at the same time materializes a complete iterative process of one frame. Adding the micro-batch, as shown in Figure 4, we get a two-dimensional matrix, each column still represents each frame in the image stream, and each row still represents several rounds of iterations in one frame of calculation. Since the number of frames in the microbatch is determined in advance, we can convert the vertical calculation process to the horizontal calculation. First, calculate the first iteration of all frames sequentially, then calculate the second iteration, and so on until the iteration process of all frames converges. The horizontal calculation does not affect the correctness of the result. When we calculate to the i round of the k frame, we have already obtained the i (inclusive) iterations before the $k-1$ frame and the $i-1$ rounds before the k frame. The change of the real-time dynamic graph from the k frame to the $k-1$ frame is included in transaction k. These results are no different from the information when the k frame to the i round are calculated separately. Micro-batch processing changes the incremental calculation process after the iterative process, and pays more attention to the change part of each frame, which is consistent with the requirement of calculating the difference in graph stream tracking.

The micro-batch calculation model has the advantage of calculating the difference. Ideally, the system only needs to save the status of the i and $i-1$ rows to complete the calculation of the i row, instead of saving the complete result of each column. The information in one line represents the difference between the state of each frame in this round of iteration and the previous frame, and it is all caused by transaction updates. The information in one column is the complete information of a frame, including a large number of states that do not need to be recalculated.

The micro-batch computing model also has the advantage of state reuse. The calculation of each line produces the difference between two adjacent frames. Adding the difference on the basis of the previous frame can be directly stored as the state of the next frame calculation, which saves the complete materialization of the calculation result of the current frame. Ideally, all changes are saved in incremental form, naturally reusing the results of the previous frame.

4 Morphoses Design

In this section, we first introduce the efficient model designed for graph stream fine-grained tracking, and then describe the architecture and API of Morphoses designed based on this model.

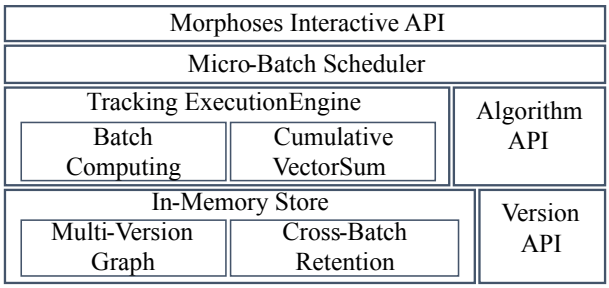

Figure 5: System architecture.

Table 2: Algorithm & Version APIs

initializeAggregationValue()	*void*
initializeVertexValue()	*bool* : *is active*
edgeFunction()	*void*
incrementalAggregationValue()	*bool* : *success*
computeFunction()	*bool* : *active next*
isVertexValueEqual()	*bool* : *is equal*
useIncrementalRecompute()	*bool* : *yes*
theSameVertexTheSameMessage()	*bool* : *yes*
insertEdgeDelta()	*void*
insertVertexDelta()	*void*
mergeVersion()	*void*

4.1 System Overview

Morphoses achieve cross-micro-batch reuse of the computing state as shown in Figure 5, it is necessary to save the iterative process of the last frame and discard all previous frames after the micro-batch processing ends. Intermediate results produced by frames other than the last frame in the microbatch can be safely deleted because they are all included in the cumulative vector sum saved for the last frame. The system then submits the transactions corresponding to all frames in the micro-batch to the real-time dynamic graph. At this time, there is no need to maintain the multi-version graph for the calculation of multiple frames.

The real-time dynamic graph merges all versions to align to the last frame. The iterative process of completely materializing the last frame needs to save the iterative process before the first frame of the current micro-batch and add the cumulative vector sum generated by the micro-batch calculation, which requires merging the states of all vertices in each iteration.

In order to amortize this cost, Morphoses adopted a "passive" materialization method. The system puts the old materialized results and the cumulative vector sum simple together, and their combination represents the complete iterative process of the last frame. The next time one of the nodes is visited, the right-hand state of the non-empty cumulative vector sum is actually merged into the materialized result to vacate the new state vector. The logical number of the micro-batch is used to identify the new and old data, and the results of different micro-batch will not affect each other.

4.2 Morphoses APIs

Morphoses provides convenient APIs to help developers implement graph iteration algorithms. Morphoses API mainly includes algorithm API and multi-version graph API. The algorithm API needs to be implemented by the user so that the system can understand the logic of the algorithm. The multi-version graph API is built into the system and does not need to be modified in most cases.

The Morphoses algorithm API is designed based on the general GAS iteration process, mainly including initialization, edge calculation, vertex calculation, and discrimination. The initialization API includes **initializeAggregationValue**() and **initializeVertexValue**(), which are used to initialize the aggregate value of the vertex and the state value of the vertex, respectively. After initializing the state, it need to return whether the vertex was active in the initial iteration. The edge API is **edgeFunction**() and its incremental version **incrementalAggregationValue**() (optional). Edge computing combines the two processes of Scatter and Gather into one function. The vertex API is **computeFunction**(), which is equivalent to the Apply procedure and returns the new activation. The discrimination API includes **isVertexValueEqual**() to judge whether the state of the vertices is the same, **useIncrementalRecompute**() to enable incremental edge calculation, and **theSameVertexTheSameMessage**() to reflect the characteristics of algorithmic message propagation. Users need to implement algorithm API when developing graph iteration algorithm application. Algorithm 1 is the main APIs of PageRank application.

Algorithm 1: Development functions for PageRank

```
function edgeFunction(edge(u,v),tid):
    delta ⟵ u.value/getOutDegree(u,tid);
    v.aggregation ⟵ v.aggregation + delta;
    return;
end function
function computeFunction(vertex,tid,global):
    v.value ⟵ (1 − global.damping) +
      (global.damping ∗ vertex.aggregation);
    return true;
end function
function isVertexValueEqual(va,vb,tid,global):
    isEqual ⟵ ABS(va.value − vb.value) <
      global.ε;
    return isEqual;
end function
```

4.3 Optimization

The difference calculation of Morphoses is essentially based on the state change of the vertex. The system records the state change of each vertex in each iteration in the form of a vector. The same vertex state and the same neighbor will inevitably produce the same message propagation, which in turn will also calculate the same next-round state. However, this also means that any changes in the state and neighboring sides risk causing errors in the history retention, and the system has to recalculate all messages and states. This introduces a redundancy that often occurs: changing the situation does not affect the spread of the message.

By observing many algorithms, we find that graph algorithms often only propagate the only kind of message about the state of the vertex. Changes in node status will mostly affect the content of messages, but changes in neighbors are not necessarily. For example, in the minimum label propagation algorithm, if a vertex has a label X, the messages it sends to all neighbors are consistent with X. When the vertex label changes to X', all messages are changed to X'; when only vertex neighbors change, irrelevant messages are still X, and new (deleted) neighbors will receive one more (less) message X. In the PageRank algorithm, the vertex also only spreads one type of message $PR(v)/N(v)$, but the difference is that the change in the number of neighbors will affect the content of all messages.

Buffering a message for each vertex can reduce the occurrence of recalculation without affecting message propagation. The system retains a sample of all messages sent by that vertex while retaining the state changes of each iteration of each vertex. This brings additional $O(|V| * ni)$ memory space usage and $O(|SV|)$ calculation overhead for each iteration. If the sample is valid, that is, the message sent by the iterative calculation of the new frame is the same as the previous frame, the message can be omitted; otherwise, if the sample is invalid, that is, a message content different from the sample appears, a new message is sent. The main meaning of a message reuse is that when vertex neighbors increase but the propagation message has not changed, only the new neighbor vertex recalculation is activated by the newly added edge, and the old neighbor will not receive the activation message. In some cases, a message reuse reduces the number of activation points from the average degree of vertices in the graph to 1.

Message reuse can be understood as treating the message as a special state retention, so in addition to the state comparison function, the user needs to provide an additional message comparison function. In the scatter phase, the vertex still calculates all the messages sent to the neighbors based on the current state, but uses the message comparison function to compare it with the remaining messages, and if they are the same, the message is skipped and not sent. The message sampled by the system is the first in the order of all messages at the vertex. The unsent message is also regarded as a special "empty" message. If the message is not empty and the new calculation needs to propagate "empty".

5 Related Work

The state-of-the-art evolving graphs analysis technology [7, 12, 21, 22] make it possible to analyze the large-scale graph stream in real-time. But it is still challenging to explore the impact of each specific update on the results of the graph algorithm. The fine-grained tracking [20] of these changes places stringent requirements on the analysis granularity, processing latency, and throughput of the system. Existing solutions cannot meet these requirements.

The state-of-the-art research has made considerable progress in the area of incremental processing of evolving graphs. If the incremental system analyzes the graph after each transaction and reports the difference between the graph algorithm results before and after the transaction, graph stream fine-grained tracking can be achieved. The current incremental analysis technology can be divided into three categories named by Dependency Retention, Iterative Process Retention and Other.

Dependency Retention requires users to define dependency functions of the graph algorithm, and the system tracks the dependency actively. The state-of-the-art representatives are RisGraph [5] and GraphBolt [14, 15]. RisGraph is designed for monotonic algorithms and optimized deeply, it uses Dependence Tree proposed by KickStarter [18] to track the propagation of vertex values. By providing a list of vertex values modified in each transaction, RisGraph can achieve fine-grained tracking in graph stream. However, the dependency tree is based on the properties of the algorithm and is only applicable to specific monotonic graph algorithms. GraphBolt proposes a more general dependency-driven model for incremental computing. GraphBolt tracks the dependency information by aggregated values on vertices, and uses dependency to merge the influence of graph structure changes. The dependency-driven model can handle some non-monotonic algorithms (e.g., PageRank, Belief Propagation), but involves much more overheads than Dependency Tree for monotonic algorithms thus can only be batch processing. The refinement function required by incremental processing also limits the use, which requires the user to design a non-trivial incremental operation of a specific algorithm manually. In conclusion, Dependency Retention is not universally applicable in the problem of graph stream tracking, since fine-grained tracking can be achieved only on monotonic algorithms. It is either impossible or cannot achieve fine-grained tracking on other algorithms. [6, 17]

Iterative Process Retention saves the intermediate results of the graph completely in each iteration, and uses this history to compute the iterations after the update incrementally, so that trivial graph-parallel abstraction can be used. The state-of-the-art representative is Tegra [8]. Tegra is an evolving

graph ad-hoc analysis system that supports effective ad-hoc window operations. The system materializes each iteration for the preceding query, and takes window switching as a graph update, so as to compute within only a limited range. However, Tegra materializes the complete iteration and revisits it to compare the differences when processing the next query, which greatly limits processing of a large number of transactions in the graph stream. Differential Dataflow [16] is a generalized streaming system. But for graph workloads, the system materializes the messages of each iteration thus complexity up to O(|E|), which makes it difficult to apply to graph stream tracking.

Other incremental analysis techniques are designed for specific graph analysis requirements [10], or not for graph algorithms but can be transplanted and applied [2,4]. Many limitations and the inapplicability led to the difficulty of processing graph stream tracking workloads.

References

[1] Yukuo Cen, Jing Zhang, Gaofei Wang, Yujie Qian, Chuizheng Meng, Zonghong Dai, Hongxia Yang, and Jie Tang. Trust relationship prediction in alibaba e-commerce platform. *IEEE Trans. Knowl. Data Eng.*, 32(5):1024–1035, 2020.

[2] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 918–934. ACM, 2019.

[3] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 619–630. ACM, 2015.

[4] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. Parallelizing sequential graph computations. *ACM Trans. Database Syst.*, 43(4):18:1–18:39, 2018.

[5] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 513–527. ACM, 2021.

[6] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. Automating incremental graph processing with flexible memoization. *Proc. VLDB Endow.*, 14(9):1613–1625, 2021.

[7] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30. USENIX Association, 2012.

[8] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: efficient ad-hoc analytics on evolving graphs. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 337–355. USENIX Association, 2021.

[9] Wenjun Jiang, Guojun Wang, Md. Zakirul Alam Bhuiyan, and Jie Wu. Understanding graph-based trust evaluation in online social networks: Methodologies and challenges. *ACM Comput. Surv.*, 49(1):10:1–10:35, 2016.

[10] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. VRGQ: evaluating a stream of iterative graph queries via value reuse. *ACM SIGOPS Oper. Syst. Rev.*, 55(1):11–20, 2021.

[11] Issa Khalil, Ting Yu, and Bei Guan. Discovering malicious domains through passive DNS data graph analysis. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 663–674. ACM, 2016.

[12] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4):29:1–29:40, 2020.

[13] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data,*

SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, pages 135–146. ACM, 2010.

[14] Mugilan Mariappan, Joanna Che, and Keval Vora. Dzig: sparsity-aware incremental processing of streaming graphs. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 83–98. ACM, 2021.

[15] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 25:1–25:16. ACM, 2019.

[16] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455. ACM, 2013.

[17] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 417–430. ACM, 2016.

[18] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 237–251. ACM, 2017.

[19] Tangwei Ying, Hanhua Chen, and Hai Jin. Pensieve: Skewness-aware version switching for efficient graph processing. In David Maier, Rachel Pottinger, An-Hai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 699–713. ACM, 2020.

[20] Fan Zhang, Lei Zou, Li Zeng, and Xiangyang Gou. Dolha - an efficient and exact data structure for streaming graphs. *World Wide Web*, 23(2):873–903, 2020.

[21] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. Depgraph: A dependency-driven accelerator for efficient iterative graph processing. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 371–384. IEEE, 2021.

[22] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.*, 13(7):1020–1034, 2020.