Real-time Constrained Cycle Detection in Large Dynamic Graphs

Xiafei Qiu[1], Wubin Cen[1], Zhengping Qian[1], You Peng[2], Ying Zhang[3], Xuemin Lin[2], Jingren Zhou[1]

[1]Alibaba Group

[2]University of New South Wales

[3]University of Technology Sydney

[1]{*xiafei.qiuxf,wubin.cwb,zhengping.qzp,jingren.zhou*}*@alibaba-inc.com*

[2]*unswpy@gmail.com,lxue@cse.unsw.edu.au*

[3]*ying.zhang@uts.edu.au*

ABSTRACT

As graph data is prevalent for an increasing number of Internet applications, continuously monitoring structural patterns in dynamic graphs in order to generate real-time alerts and trigger prompt actions becomes critical for many applications. In this paper, we present a new system `GraphS` to efficiently detect constrained cycles in a dynamic graph, which is changing constantly, and return the satisfying cycles in real-time. A hot point based index is built and efficiently maintained for each query so as to greatly speed-up query time and achieve high system throughput. The `GraphS` system is developed at Alibaba to actively monitor various online fraudulent activities based on cycle detection. For a dynamic graph with hundreds of millions of edges and vertices, the system is capable to cope with a peak rate of tens of thousands of edge updates per second and find all the cycles with predefined constraints with a 99.9% latency of 20 milliseconds.

PVLDB Reference Format:
Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *PVLDB*, 11 (12): 1876-1888, 2018.
DOI: https://doi.org/10.14778/3229863.3229874

1. INTRODUCTION

With the rapid development of information technology, data generated by an increasing number of applications is being modeled as graphs. On one hand, the graph structure is able to encode complex relationships among entities. Examples include social networks, e-commerce transactions, and electronic payments, etc. It is typical for such graphs to contain hundreds of millions of edges and vertices. Sophisticated analytics over such large-scale graphs provides valuable insights to the underlying dataset and interactions among different entities. On the other hand, the structural changes to such graphs are constant in nature, which makes them very dynamic, together with continuous stream of information produced by the entities. It is imperative and challenging to design an efficient graph system to store and manage dynamic graphs at scale and provide real-time analytics to explore and mine fast-evolving structural patterns.

In this paper, we study a problem of continuous constrained cycle detection in large dynamic graphs. Specifically, for each incoming edge of the dynamic graph, the goal is to identify the newly generated cycles and return them for a set of continuous queries, respectively. Each query can ask for cycles satisfying some predefined constraints, such as length and attribute constraints. Detecting constrained cycles in real-time turns out to be valuable for many real-world applications. We use two simplified real-world examples in the context of e-commerce and personal finance to illustrate the importance of such operations.

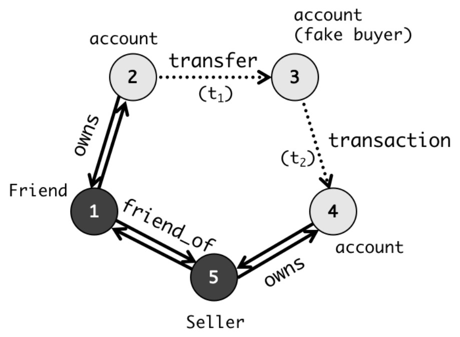


Figure 1: A graph of a merchant-fraud example.

Figure 1 represents a simplified graph among buyers and sellers in an e-commerce platform. We denote individual users (buyers or sellers) and their accounts as vertices in the graph. There are two types of edges. One type of *static* edges (in solid lines) models the association of accounts to

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229874

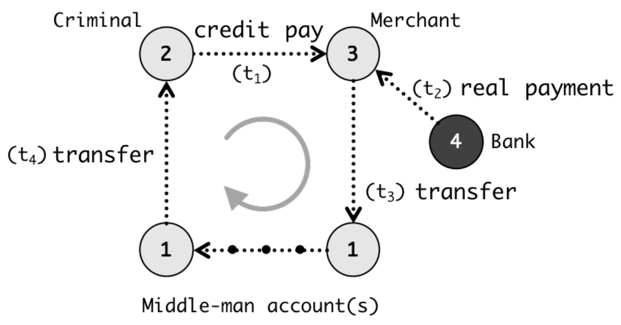


Figure 2: A graph of a credit-card-fraud example.

users and the relationships among different users, while online transactions including payment activities are denoted as *dynamic* edges (in dotted lines) for the corresponding vertices. In order to increase the popularity for a merchandise so as to improve future sales, *fake transactions* are placed to artificially bump up the number of past transactions. In this example, this is achieved through a third-party account (vertex 3) from which a normal order is placed and its payment (edge $3 \rightarrow 4$) is completed at time t_2. However, the merchandise is never shipped by the seller (vertex 5) and the money used for the payment by the fake buyer (vertex 3) was previously transfered to him/her via the seller's friend (vertex 1) at time t_1 using his or her own account (vertex 2). The entire process is rather complicated involving multiple entities. Interestingly, it generates a cycle ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$) in the graph, which can be served as strong indication that a fraud may exist.

Figure 2 represents a series of credit card transactions using graph and describes an interesting case of credit card fraud. By using fake IDs, a "criminal" (vertex 2) may obtain a short-term credit from a bank (vertex 4). He tries to illegally cash out money by faking a purchase (edge $2 \rightarrow 3$) at time t_1 with the help of a merchant (vertex 3). Once the merchant receives payment (edge $4 \rightarrow 3$) from the bank (vertex 4), he tries to send the money back (edges $3 \rightarrow 1$ and $1 \rightarrow 2$) to the "criminal" via a middle man (vertex 1) at time t_3 and t_4, respectively. If the system can detect the cycle ($2 \rightarrow 3 \rightarrow 1 \rightarrow 2$) in real time, it becomes possible to stop such fraud in time.

In both examples, such fraudulent activities become one of the major issues for the e-commerce platform, like Alibaba's Taobao and TMall, where the graph could contain hundreds of millions of vertices (users) and billions of edges (payments, transactions). In reality, the entire fraudulent process can involve a complex chain of transactions, through many entities, which requires complex cycle detection with various constraints such as the length of the cycles and the amount of a transaction (edge) to identify and monitor. In addition, the graph is being updated at a tremendous speed of tens of thousands of transactions per second. How to quickly detect various constrained cycles in such large dynamic graph becomes critical in stopping fraud in time and preventing potential financial loss.

Previous proposed graph processing frameworks such as Pregel [17] and GraphX [11] perform off-line analytics by leveraging graph partitioning and data parallelism to achieve high throughput and good scalability. However, we are confronting a task asking for a significantly lower latency. The quantitative difference is big enough to require a qualitative change in the architecture. Streaming systems have been a hot research topics. Twitter Storm [4] and Apache Flink [3] have recently been focusing on high-level programming abstractions and resiliency for streaming at scale. While fault tolerance is important, we need to perform complex graph traversals over streaming data which, to our knowledge, is beyond the expressiveness of today's popular stream processors. It very likely also needs further specialization to meet the latency goals.

Creating an index for finding *shortest paths* in a static graph has been studied extensively. However, it is not directly applicable to our case, which required continuously locating all possible cycles in highly dynamic graphs. In addition, the space and maintenance overhead of such an index is often too high to be scalable for dynamic graphs. Alternatively, a straightforward approach uses a breadth or depth-first search to traverse the graph whenever it is changed. The approach incurs significant overhead for repetitive computation. Moreover, the skewness of real-world graphs often leads to extremely unbalanced response times. As described later, when a query encounters some vertices with a high degree of connectivities, the approach may introduce exponentially more possible paths to explore and cause significant delays in response.

In this paper, we present a new system, called `GraphS`, to take on the above challenges of continuous constrained cycle detection over fast changing graphs. It leverages a dynamic index structure to optimize trade-offs between memory usage and query efficiency, with minimized maintenance cost. Specifically, for each query, we propose *hot point based index*, which can be selectively applied to a certain portion of the graph and exploits various heuristics (e.g., memory usage and vertex connectivities) to balance cost and efficiency. As the graph evolves, the system automatically adapts itself to capture new hot points and update the index accordingly. In order to cope with a high rate of edge updates without sacrificing query performance, the index is efficiently maintained as a by-product of search evaluation. Additional optimization techniques are designed to evaluate constrained cycles concurrently and optimistically to increase system throughput while guaranteeing correctness. `GraphS` is a real production system deployed at Alibaba. It has served as the foundation for several key businesses to detect constrained cycles in real-time and prevent fraud in a very dynamic environment. The system can handle a dynamic graph with hundreds of millions of edges and vertices, and is capable to cope with a peak rate of tens of thousands of edge updates per second and find all the cycles with predefined constraints within a few milliseconds.

The rest of the paper is organized as followed. We formally define the problem, describe straightforward solutions, and demonstrate the challenges in Section 2. We then present a novel hot point based index in Section 3. In Section 4, we describe the `GraphS` system and its internals systematically. We present experiments and demonstrate the performance improvement in Section 5. Finally, we survey the related work in Section 6 and conclude in Section 7.

2. PRELIMINARY

In this section, we first formally introduce the problem of constrained cycle detection in dynamic graphs, then present a baseline solution based on the depth-first search.

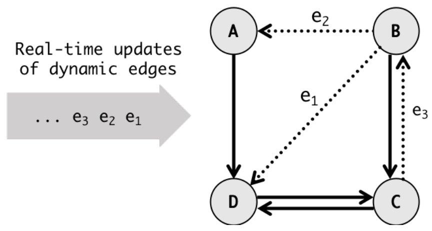


Figure 3: Real-time constrained cycle detection on dynamic graphs.

2.1 Problem Definition

Let $G = (V, E, A_V, A_E)$ denote a directed graph, where (1) V is a set of vertices; (2) $E \subseteq V \times V$, in which $e(u, v) \in E$ denotes an edge from vertex u to v; and (3) A_V (A_E) denotes the set of attributes of the vertices (edges). For instance, we use $t(e)$ to denote the timestamp ($t \in A_E$) of the edge e. In the system, we have two types of edges: *static* and *dynamic*. For every static edge e (e.g., friend relation between two users), we set $t(e) = \infty$ where the edge will never expire. For dynamic edges (e.g., money transfers among users), we assume they have logical/application timestamps, indicating such as transaction time, and they arrive in order. In this paper, we use the popular sliding window model to capture the dynamic of a graph G. Particularly, we maintain a system clock c to indicate current time, which is updated upon the arrival of every edge. A dynamic edge e with $t(e) + W \leq c$ will be expired in graph G, where W is the length of the sliding window. Whenever the context is clear, we use G to denote the *dynamic graph* under the sliding window model. In the system, the dynamic graph G is initialized by a base graph G_0 with all vertices and static edges. Then it is continuously updated upon the arrival and expiration of the dynamic edges.

We shall use the following notations for a dynamic graph G. A *path* p from vertex v to v' is a sequence of vertices $v = v_0, v_1, \ldots, v_n = v'$ such that $(v_{i-1}, v_i) \in E$ for every $i \in [1, n]$. We may also use $p(u, v)$ to denote a path from vertex u to v. The *length* of path (cycle) p, denoted by $len(p)$, is the number of edges in the path (cycle). We say a path is *simple* path if there is no repetitions of vertices and edges. A *cycle* is a path p with $v_0 = v_n$ and $len(p) \geq 3$. A cycle is a *simple cycle* if there is no repetitions of vertices and edges, other than the repetition of the starting and ending vertex.

Definition 1 (LENGTH CONSTRAINT). *We say a path or cycle p satisfy the length constraint if $len(p) \leq k$ where k is a predefined number.*

In the real applications, users may also impose other constraints based on the attribute values of the edges and vertices for different business logics. In this paper, we consider the simple predicate on individual edge and vertex as follows.

Definition 2 (ATTRIBUTE CONSTRAINT). *Let $f_A()$ denote a user-defined boolean function against the attribute values of the edges or vertices, we say an edge e (resp. a node u) satisfies the attribute constraint if $f_A(e)$ (resp. $f_A(u)$) is true. And a path or cycle satisfies the attribute constraint if every edge and node satisfies the attribute constraint.*

Definition 3 (CONTINUOUS CYCLE QUERY). *A continuous cycle query $q = (k, f_A(.))$ will incrementally report the new simple cycles resulting from the arrival of each new edge on the dynamic graph G, where each cycle satisfies both length and attribute constraints.*

Problem Statement. Given the dynamic graph G and a set of continuous (constrained) cycle queries Q, for every incoming edge e, we aim to develop a system to support every query $q \in Q$ in a real-time manner.

Example 1. *Figure 3 illustrates an example of continuous constrained cycle detection where three dynamic edges $\{e_1, e_2, e_3\}$ arrive in order. Suppose the length constraint of q is 3, and there is no attribute constraint. It is shown that the cycle (B, D, C) will be reported upon the arrival of $e_3(C, B)$. Note that the cycle (B, A, D, C) does not satisfy the length constraint and hence will not be reported.*

2.2 A Baseline Solution

Given the dynamic graph G and the attribute constraint of a query q, we can easily come up with a graph G_q by filtering the unsatisfied edges and vertices. Clearly we do not need to materialize G_q for each individual query q since the attribute constraint can be easily integrated into the cycle detection algorithm by discarding the unsatisfied vertices and edges during the search. Thus, in the following, we focus on the length constrained cycle detection.

An incoming edge $e(d, s)$ will result in a simple cycle with length constrained $k + 1$ for every simple path $p(s, d)$ with $len(p) \leq k$. So the key is to find all simple paths $\{p(s, d)\}$ under length constraint. The following algorithm uses a depth-first search (DFS) to enumerate all simple paths from a source vertex s to a destination vertex d. Upon each update $e(v, u)$, the algorithm takes as input the current snapshot of the directed graph G, a source vertex $s = e.u$ and a destination vertex $d = e.v$ of G, and the parameter, $q.k$, for length constraint. It outputs all simple paths from s to d in G, up to length k. The search starts from the source vertex, along the directed edges. Once it reaches the destination, a leaf vertex or the current length is already k, it backtracks to continue the search for other possible paths.

Algorithm 1 Baseline Solution (graph G, edge $e(d, s)$, length $k + 1$)

```
 1: Stack $S = \emptyset$                    ▷ keeps track of current path.
 2: bool visited[] = false
 3: FindAllSimplePaths($G$,$s$,$d$,$k$)

 4: procedure FINDALLSIMPLEPATHS(graph $G$, vertex $s$,
    vertex $d$, length $k$))
 5:     visited[s] = true;S.push(s)
 6:     if s==d then
 7:         report the simple path, i.e., a simple cycle in-
    curred by $e(d, s)$
 8:     else
 9:         if S.size() $\leq k$ then
10:             for every $v$ in $G.adj(s)$ do
11:                 if !visited[v] then
12:                     FindAllSimplePaths(G, v, d, k)
13:     S.pop(); visited[s] = false
```

Analysis. In Figure 4, we report the response time for a continuous constrained cycle query with length constraint

$k = 6$ against the arrival of new edges (See Section 5 for detailed data description). It is shown that there are many spikes, where the incoming edges consume significantly longer time than the average. These spikes will seriously deteriorate the throughput of the system, and hence are unacceptable in a real-time system. The key reason of the phenomenon is the existence of some hot points, i.e., points with very high out-going degrees. The number of edges visited in Algorithm 1 grows greatly whenever a hot point is encountered. Figure 5 shows a clear correlation between the number of the edges visited by the query and the response time. This motivates us to develop a hot points based indexing technique to alleviate this issue. To further explain why the existence of hot points hurts the system performance, Figure 6 reports the number of paths with length k in our real-life graph evaluated in the experiments as well as a random graph with the same number of vertices (518, 959, 261) and edges (2, 088, 968, 416), traversing from 10, 000 randomly selected vertices. It is shown that the number of k-length paths (i.e., edges visited) grows very quickly on real-life graph because the existence of hot points. Particularly, when $k = 3$ and $k = 5$, around 93% and 99% of the paths encounters at least one hot point (with out-going degree not less than 40) respectively. Note that, intuitively, the *total* number of paths in the random graph should be larger than that of the real-life graph due to the fact that the total number of edges is the same. However, the number of paths grows more quickly in real-life network when the length constraint k is not large. Note that, the k values used in our applications are not large.

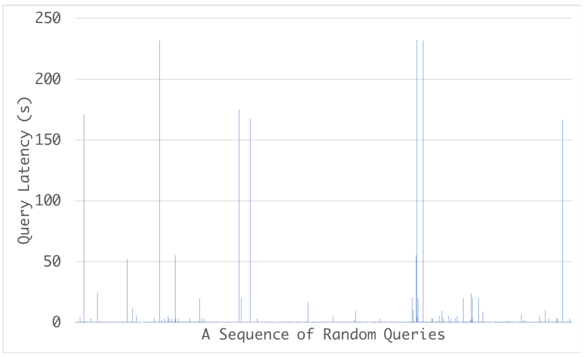


Figure 4: Latency spikes of the baseline solution.

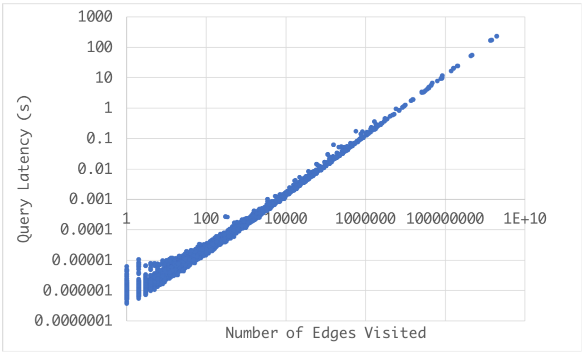


Figure 5: A clear correlation between query times and the number of edges visited.

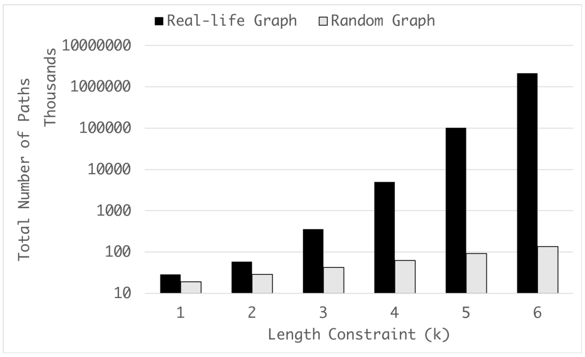


Figure 6: Number of paths on random and real-life graphs.

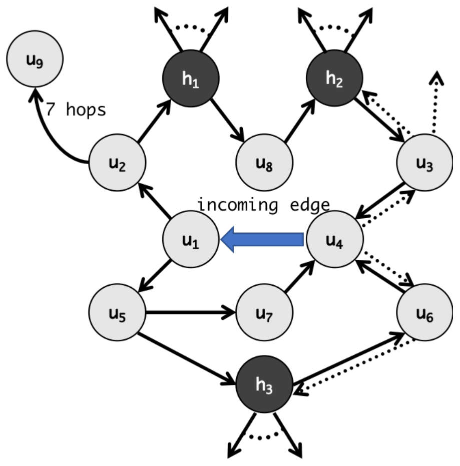


Figure 7: Motivation of hot points based index.

3. HOT POINT BASED INDEX (`HP-Index`)

As discussed in Section 2.2, the key technique for the continuous constrained cycle detection is the length constrained cycle detection and the key challenge is how to handle the hot points encountered during the search. In this section, we propose a novel hot points based indexing technique, namely `HP-Index`, to support length constrained cycle detection. Particularly, Section 3.1 introduces the motivation of `HP-Index`. Sections 3.2 and 3.3 present the indexing structure and corresponding searching algorithm. Section 3.4 shows the dynamic maintenance of the `HP-Index`. Section 3.5 discusses how to handle multiple continuous constrained cycle queries.

3.1 Motivation

As discussed in Section 2.2, the existence of hot points (i.e., vertices with large number of out-going edges) in real-life graph may seriously deteriorate the throughput of the system. As illustrated in Figure 7, when the edge (u_4, u_1) comes, we can enumerate the paths $\{p(u_1, u_4)\}$ with length constraint $k = 7$ by conducting a DFS search starting from u_1. A pitfall of this approach is that the search space may quickly blow up when encountering a vertex with many out-going edges. For instance, when the vertex h_1 in Figure 7 is visited, it may come up with a large number of paths and many of them will not reach u_4 within 7 hops.

This motivates us to build hot points based index such that we can avoid explicitly exploring the out-going edges of hot points when we traverse the graph. The key idea of the hot points based indexing technique is to continuously maintain the length constrained paths for each pair of hot points. By doing this, we can conduct DFS for u_1 on the graph G. A search branch will be suspended when a hot point (e.g., h_1 and h_3 in Figure 7) is encountered or the length constraint is violated. We also conduct a DFS from u_4 on the reverse graph of G, (i.e., all edges are reversed, denoted by dash lines in Figure 7), and suspend a search branch whenever we meet a hot point or the length constraint is violated. Then we can take advantage of the pre-computed length constrained paths among hot points to generate the paths with length constraint, which can significantly reduce the search space.

Particularly, to meet the real-time response time requirement for a large dynamic graph, our proposed indexing technique should have the following properties: (1) has small index size; (2) can be quickly updated; and (3) can efficiently enumerate all length constrained simple paths for any two vertices.

By analysing the structure characteristics of the graph (with $518,959,261$ vertices and $2,088,968,416$ edges) used in our case study (See Section 5 for detailed data description), we find it is a highly skewed scale-free graph. For instance, there are only $5,274$ vertices with degree larger than $4,000$. Moreover the number of paths among these vertices with length smaller than 6 is only $1,399,383$, which is much smaller than the graph size. These indicate that we can afford maintaining a considerable number of hot points and the length constraint paths among them.

Based on the index, we develop efficient cycle detection algorithm (i.e., path enumeration algorithm) for every incoming edges. Meanwhile, we also show that the index maintenance is by-product of the searching procedure. By doing this, we show our indexing technique meets three properties required. As demonstrated in our case study, our proposed `HP-Index` not only significantly reduces the number of spikes in the response times but also enhances the average processing time.

3.2 Index Structure

We introduce the `HP-Index` structure in this subsection.

Hot point. Let $d(v)$ be the degree for a vertex v, and t be the threshold. Let H be a set of vertices in G satisfying $\forall h \in H, d(v) \geq t$, called hot points. Note that we also use reverse edges of the vertices in the search processing, so we consider both in and out degrees of a vertex.

For every pair of hot points $h_i, h_j \in H$, the index contains all the (pre-computed) paths between h_i and h_j which has lengths less than or equal to k and each path does not go through any other hot point. The paths are organized into an *index graph*, denoted by G_{idx}. The index contains only the hot points $h \in H$. Each edge represents a path, weighted by its length.

In addition, we also maintain a reverse graph G' to support query processing. It has the same set of vertices as G. For the edges, $e'\langle v, u\rangle \in G'$ if and only if $e\langle u, v\rangle \in G$. Note that we will not immediately double the storage space because the static edges are usually bi-directed. Moreover, we do not need to duplicate the attribute values of the edges.

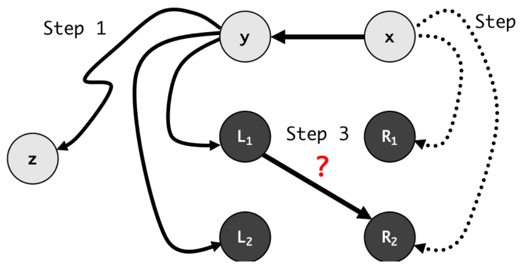


Figure 8: Length constrained paths enumeration using index.

3.3 Search Algorithm

For each incoming edge $\langle u, v\rangle$, we search for all simple paths from $s = v$ to $d = u$ in G with lengths less than or equal to k (abbr. as k-paths[1]), with the following three steps as illustrated in Figure 8.

Step 1. Conduct a DFS search starting from the source vertex s. A search branch stops when: i) d is reached (e.g., path $u_1u_5u_7u_4$ in Figure 7). ii) already has length k (e.g., path $u_1u_2 \dots u_9$ in Figure 7). iii) meets a hot point (e.g., paths $u_1u_2h_1$ and $u_1u_5h_3$ in Figure 7).

Note that this is the same as straightforward approach except that we suspend a search branch whenever a hot point is encountered. If there is no hot point appearing during the search procedure, it is immediate that we have already identified all the k-paths as required. Otherwise, we need to consider the paths involving hot points.

By L, we denote the set of hot points encountered in Step 1. For each hot point (vertex) $h \in L$, we record all paths from source s to h, denoted by $p(h)$. Then we go to Step 2.

Step 2. Conduct another DFS traversal from the destination vertex d in the reversed graph G' and stop if: (i) s is reached. (ii) already reach length k.[2] (iii) meet a hot point (e.g., h_2 and h_3 in Figure 7), which will be kept in a set R.

For each hot point (vertex) $h \in R$, let $p'(h)$ records all paths with reverse edges ending up at h in Step 2. Let $S = R \cap L$, we can identify the new k-paths with only one hot point $h \in S$ by checking all possible valid combinations of the paths in $p(h)$ and $p'(h)$. For instance, we can immediately identify the path $u_1u_5h_3u_6u_4$ by considering paths $u_1u_5h_3$ in $p(h_3)$ and $u_4u_6h_3$ in $p'(h_3)$.

Step 3. This step will identify the remaining paths involving more than one hot point by utilizing the index structure. For each hot point $h_i \in L$, we conduct a depth-first search with length constraint k on the index graph G_{idx} to identify the hot points $\{h_j\} \subseteq R$. For each pair of hot points h_i and h_j, k-paths from h_i to h_j, denoted by $P_{i,j}$ can be identified in the above search. Then, the k-paths from s to d can be derived by concatenating the partial paths from $p(h_i)$, $P_{i,j}$, and $p'(h_j)$. Note that we need to reverse the paths from $p'(h_j)$. For instance, we have $u_1u_2h_1 \in p(h_1)$, $h_1u_8h_2 \in P_{1,2}$, and $u_4u_3h_2 \in p'(h_2)$ in Figure 7. Then we can come up with a valid path $u_1u_2h_1u_8h_2u_3u_4$.

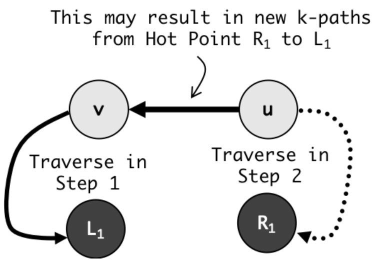
[1]For presentation simplicity, we assume the length constraint of the cycle is $k+1$

[2]An immediate optimization is to search for length $k - l_{min}$ only, where l_{min} is the shortest lengths of all paths in $\bigcup p(h), \forall h \in L$.

REMARK 1. *Note that we assume both s and d are not hot points in the above algorithm description. We may simply set $L = \{s\}$ (resp. $R = \{d\}$) if s (resp. d) is a hot point in Step 1 (resp. 2).*

3.4 Index Maintenance

In this subsection, we show how to continuously maintain the `HP-Index` upon the arrival and expiration of edges.



This may result in new k-paths from Hot Point R_1 to L_1
v
u
Traverse in Step 1
Traverse in Step 2
L_1
R_1

Figure 9: Index maintenance on edge insertion is a by-product of the search algorithm.

Edge insertion. When a dynamic edge $\langle u, v\rangle$ arrives and is inserted into the dynamic graph G, we need to add the new k-paths among the hot points resulting from the insertion of e. Assume that both u and v are not hot points [3]. As shown in Figure 9, let p be a new k-path from hot points R_1 to L_1. Clearly, p must contain the incoming edge $\langle u, v\rangle$. Since $len(p) \leq k$, its subpath from v to L_1 and the reversed subpath from u to R_1 must be retrieved in Steps 1 and 2, respectively. Thus, we can immediately identify p by checking the paths in L_1 and R_1 obtained in Steps 1 and 2. For instance, in Figure 7, we have a new path $h_2u_3u_4u_1u_2h_1$ from hot point h_2 to the hot point h_1 upon the arrival of $\langle u_4, u_1\rangle$. The subpath $u_1u_2h_1$ is retrieved in Step 1 w.r.t h_1, and the reverse of subpath $h_2u_3u_4$, i.e., $u_4u_3h_2$, is retrieved in Step 2 w.r.t h_2.

Edge deletion. Suppose the k-paths of the index are maintained by an inverted index based on edge Ids, we can immediately identify relevant paths and remove them when an edge expires. In our implementation, we adopt a lazy strategy where a path is set invalid if any of its edge is expired. The invalid paths can be deleted in batches later.

Adjusting the `HP-Index`. With the involving of the dynamic graph G, the degree distribution may change overtime and hence we may need to adjust the `HP-Index` to accommodate the change. We can easily tune the threshold t to increase or reduce the number of hot points. When a vertex u is included as a new hot point, we need to enumerate the k-paths to existing hot points by applying DFS (Algorithm 1) on both G (for k-paths starting from u) and reverse graph G' (for k-paths ending up at u) without exploring any hot point. Recall that a hot point can only be the start or end vertex of the k-paths in `HP-Index`. Thus, we also need to remove the k-paths $x \rightsquigarrow u \rightsquigarrow y$ from the `HP-Index`. Note that both $x \rightsquigarrow u$ and $u \rightsquigarrow y$ are detected in the above computation. When a hot point v is removed from `HP-Index`, all its relevant paths will be deleted. In addition, v may be included in the k-paths of the hot points as a non-hot point (i.e., not the begin or end vertex in the k-paths of `HP-Index`). Let $L = \{x\}$ denote the hot points which have

[3]The solution is trivial if one or both of u or v are hot points.

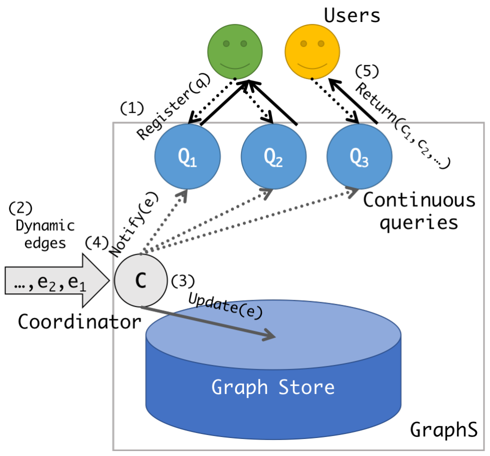


Figure 10: Architecture overview of the `GraphS` system.

k-paths $x \rightsquigarrow v$ in `HP-Index`. Similarly, let $R = \{y\}$ denote the hot points with k-paths $v \rightsquigarrow y$. We concatenate every possible paths $\{x \rightsquigarrow v \rightsquigarrow y\}$ and include the ones satisfying the length constraint to `HP-Index`.

3.5 Coping with Multiple Continuous Constrained Cycle Queries

In the system, we may need to support multiple continuous queries with different length or attribute constraints. For the queries with the same attribute constraint, we only need to take care of the query with the largest length constraint k_{max} because its outputs include all valid cycles for other queries. Regarding the attribute constraint of a query q, we only need to keep the valid vertices and edges , which may result in a much smaller dynamic graph G_q and the corresponding `HP-Index`. To facilitate the concurrency of the system and support different attribute constraints, we maintain a `HP-Index` for each individual query. More details will be discussed in the next Section.

REMARK 2. *Similar to the length constraint, it is possible for us to consider computing sharing of the queries with similar attribute constraints. This will be an interesting future research direction, and is beyond the scope of this paper.*

4. SYSTEM IMPLEMENTATIONS

In this section, we first present the overall architecture for `GraphS`. We then describe system internal components which are able to efficiently store and update graph data, continuously monitor the graph structures, and return constrained cycles timely. Finally, we discuss additional system optimizations to improve system throughput and provide fault tolerance.

4.1 Architectural Overview

Figure 10 shows the architecture overview of the `GraphS` system. A graph store is responsible for persistently storing the graph data. For low-latency access, the system also maintains a copy of the graph data in main memory using an optimized compact representation as described below.

Users can define a query for constrained cycles and register it to the system as a continuous query (shown as Step (1)). For each query registered at time t, `GraphS` will return all

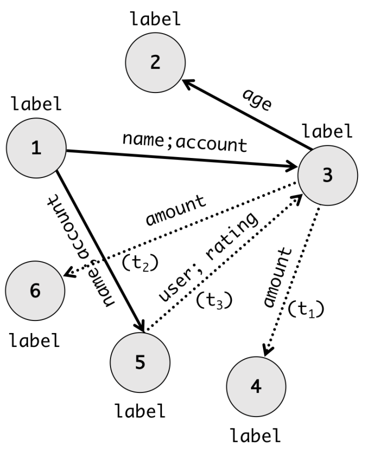


Figure 11: An example graph.

the cycles involving at least one dynamic edge with a timestamp large than t. The system detects cycles incrementally as follows. As edge updates come in continuously (shown as Step (2)), the system first applies them to the graph store (shown as Step (3)) and then notifies each query of the updates (shown as Step (4)). Each query calculates for new instances of satisfying cycles, evaluates additional constraints if needed, and returns the results to users (shown as Step (5)). At any time, users can stop or abort a continuous query.

For the rest of this section, we go through each component and describe the implementation in details.

4.2 Graph Store

As described in Section 2, there are two types of edges in the graph. The system stores them separately for optimized performance.

1. *Static* edges. They do not change in the query lifetime, and only get updated occasionally in a batch fashion.
2. *Dynamic* edges. Each of them has a timestamp and expires after a defined time interval.

Figure 11 shows an example graph. Edges with solid lines are *static* edges while the ones with dotted lines are *dynamic* edges. Conceptually, the graph store maintains the following two mappings.

- Graph topology. That is a mapping from a vertex to its edges.
- Properties. That is a mapping from either a vertex or an edge to its properties.

As a graph could contain hundreds of millions of edges and vertices, it is important to maximize memory efficiency and enable fast query performance.

Compact representation. Figures 12 and 13 show the internal representation for static and dynamic subgraphs, respectively, of the example graph.

For static graph, the list of edges are stored consecutively in an array, and their starting offsets are stored in the map. Further, we sort the edges according to their destination vertex in order that more sequential memory access is possible.

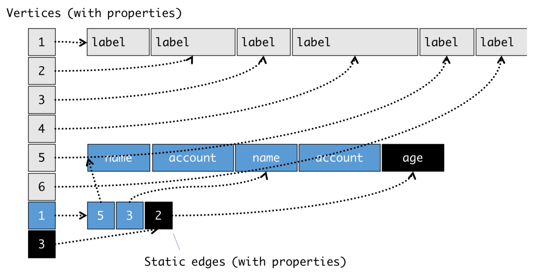


Figure 12: Static graph representation.

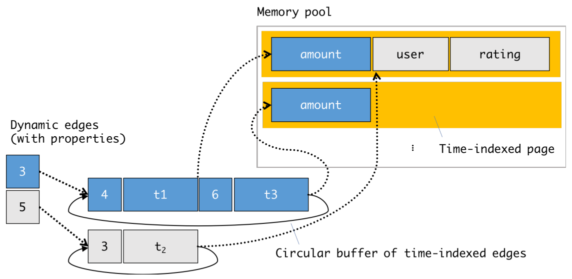

Memory pool
amount
user
rating
amount
Time-indexed page
Dynamic edges (with properties)
3
5
4
t1
6
t3
3
t2
Circular buffer of time-indexed edges

Figure 13: Dynamic graph representation.

We use a specialized encoding for vertices only having one edge, which has a large percentage in our dataset (nearly 67%), by storing the edge information directly in the map without indexing to a separate memory location. This gives about 17% of memory saving.

Each entry in the edge list contains its destination vertex, time stamp (for dynamic edges), and a pointer to its property values. For the property values, we introduce "schema" for each vertex or edge, which defines the meta-data information about their property names and value types. Then, a schema-specific encoding is used for different types.

We group together property values of a fixed length and keep them in consecutive memory without any gap. In this case, their offsets just need to be calculated per schema. For values of variable lengths, the size information has to be stored together with the value.

As shown in Figure 13, we use a circular buffer to maintain the list of dynamic edges, in order of their arrival (and expiration) times. On every edge insert, the circular buffer is updated immediately for that particular vertex. Timeout edges for that vertex is marked as deleted from the list, though we defer the actual deletion of their property values as well as other timeout edges from other vertex, as described below.

Deferred edge deletion. The removal of edge properties, as well as edges from other vertex is deferred to a later time using a garbage-collection mechanism running in the background when the system load is low. As every edge has a timestamp, the obsolete edges do not affect the query correctness.

Efficient memory management for variable-length properties is challenging. The cost of frequent allocation and deallocation is no longer negligible and can introduce memory fragmentation. Therefore, as shown in Figure 13, the system uses a memory pool of fixed-length pages for storing the properties of dynamic edges. Each page is tagged with the latest expiration time of the properties it contains and

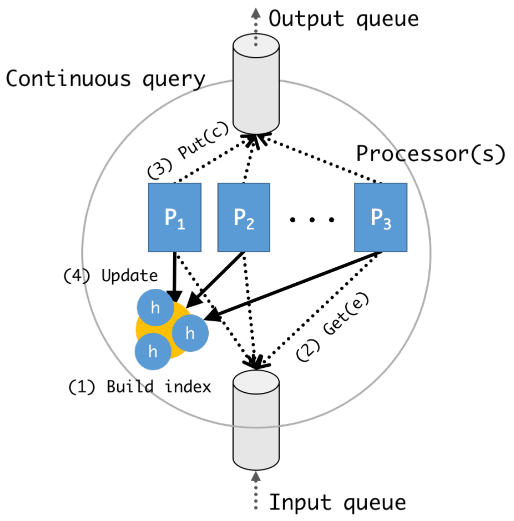


Figure 14: Continuous query processing.

is managed by a garbage-collection process. If its expiration time passes, the system can deallocate the page entirely, instead of deallocating properties one after another.

4.3 Query Evaluation

As described earlier, the system supports multiple concurrent queries in parallel. Each query is assigned with a dedicated process and the system provides its fault tolerance, independent from the others. Figure 14 uses Q_1 as an example to illustrate continuous query evaluation.

When a continuous query is submitted, the hot point index is built against the current snapshot (shown in Step (1)). Such index is relatively small compared to the original graph and its properties. We use simple hash maps, with paths ordered by their length. As a new edge update comes in (shown in Step (2)), the system evaluates if new satisfying cycles appear using the hot point index and output them immediately (shown in Step (3)). Lastly, it also maintains the hot point index as the last step of the process (shown in Step (4)).

As edge updates come in order, the system can conceptually consume each edge update one by one. However, the process is done in serial and may not be able to cope with a spike of incoming edge updates. In order to improve the overall throughput, the system maintains a pool of threads to handle different edge updates optimistically without waiting for previous edge updates to finish. The size of the thread pool can be dynamically adjusted as needed.

For most of the cases when two edge updates involves different parts of the graph and hot points, they can be processed independently and concurrently without affecting the result correctness.

Occasionally, the subsequent edge update could rely on the index maintenance from the earlier edge update. If that happens, a *compensation* query is issued to evaluate possible missing cycles. For instance, for a given edge update sequence $..., e_1, e_2, ...$, before e_1 updates the hot point index, the system assigns a thread to evaluate e_2 using the current version of the hot point index and returns results without waiting. The system also records which hot points

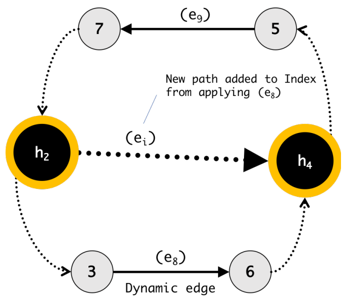


Figure 15: Concurrent query processing.

are involved when evaluating e_2. When e_1 is finally completed, the system checks if the modified hot point index intersects with the set of hot points that e_2 depends on. If so, a new compensation query is performed as follows: e_2 is re-evaluated and only satisfying cycles containing edges with a timestamp of e_1 are returned.

Figure 15 shows an example of applying two concurrent edge updates. We assume $..., e_8, e_9, ...$ as the sequence of added new edges. Before applying e_8 to the hot point index, the system issues new threads to applying update e_9. It uses an old version of the hot point index before the e_8 update and return satisfying cycles promptly. In this case, updating e_9 relies on the paths between the hot point h_2 and h_4. If applying e_8 does not add any new path between the two points, there is no potential conflicts and both the updates can be performed concurrently without missing any results. If applying e_8 does add a new path, for instance e_i in this example, applying e_9 could miss some potential cycles. Once e_8 updates the hot point index, a compensation query is issued for e_9 to calculate cycles with an edge whose timestamp is larger than the timestamp of the operated hot point index.

4.4 Fault Tolerance

The `GraphS` system achieves high availability by running multiple instances of the system with the graph replicated among them. Incoming edges are persisted using a reliable store before being consumed by all the instances. This ensures a deterministic order for events in order to generate consistent results in case of failures.

Within each instance, we separate different continuous queries and their HP-index into different processes. The graph is accessed through shared memory. If one continuous query fails, we take a snapshot of one of the remaining healthy instances, record the sequence number of the corresponding input, and use that information to recover the computation, possibly on a different server.

To detect a failure, `GraphS` monitors CPU, memory, and network resource utilization, as well as query response times, among many other metrics. The information is also used to tune performance-critical configurations such as the frequency of garbage collection to minimize impact to query latency while ensuring enough memory available.

4.5 Additional Optimizations

Hash map is used extensively throughout the implementation of `GraphS` to efficiently store and query vertex property, edge information and indexed paths. The default hash map implementation in our library uses linear probing to resolve

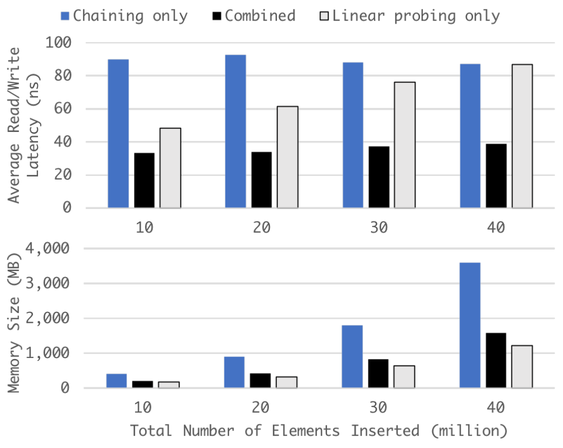


Figure 16: Hash map optimization.

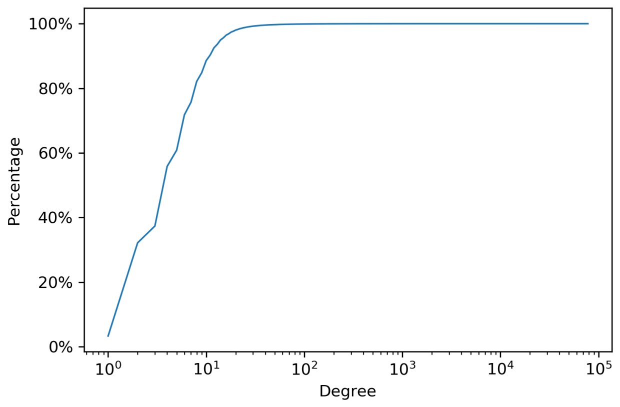


Figure 17: Degree distribution of graph vertices.

conflicts, which is to find the next available slot in an array. Such an implementation is very memory efficient as shown in Figure 16. But when we try to fit a very large graph into limited memory, the conflict rate increases significantly, which results in poor latency. Another common approach for hash conflict resolution is to chain the conflicted entries in a linked list. Since our vertex set remains stable (though the edges come and go on the fly), the overhead of linked list could be significant, as shown in the figure, because there are additional pointers to maintain and it is unlikely for consecutive accesses to fit in a CPU cache line.

We combine the two strategies in our implementation for hash map. We *first* try to resolve any conflicts using linear probing. If the conflict remains after m (a configurable parameter) probes, we switch to use a linked list for conflicts. Such design improves memory efficiency and access latency (as it is more friendly to CPU cache line than chaining), as shown in Figure 16.

5. EVALUATION

The `GraphS` is developed using Rust [24] at Alibaba. The system is deployed in production to actively monitor constrained cycles for dynamic graphs in different business scenarios. The detected cycles are streamed in real time to the control center, which classifies them into several groups based on other data analysis and estimated degree of severity. Further actions are then applied accordingly, including tagging problematic transactions for review, automatically canceling fraudulent transactions, blacklisting suspicious personnels, etc. The detailed description of various business logics and policies is out of the scope of this paper. Nevertheless, such constrained cycle detection has played a critical role in the entire process.

The following report uses constrained cycle queries chosen from a real workload and evaluated on a production cluster. All the queries are performed on an Intel(R) Xeon(R) E5-2650 server with 32 cores (at 2.60GHz) and 128GB of memory. In Section 5.1, we present the dynamic graph and outline its characteristics. We describe the chosen queries and their properties in Section 5.2. In Section 5.3, we systematically evaluate the performance of such queries in production, including the choice of `HP-Index`es and their impact on the query performance. Finally, we demonstrate the scalability of the system in Section 5.4 and discuss the impact of high frequency updates on the system performance in Section 5.5.

5.1 Dynamic Graph

The dynamic graph dataset is based on activities on the Alibaba's e-commerce platform, including different types of entities, real-time transaction and payments among them. The graph data contains both static and dynamic information. For instance, the vertices represent individual users and accounts. Static edges denote relationships among users (such as friends) as well as the mapping from accounts to their owners. Real-time transactions and payments are modeled by dynamic edges.

For a given day, this particular graph is being updated at an average rate of $3,000$ edges per second. At the peak, over 20,000 new edges are added per second. From a random snapshot, we observe $518,959,261$ vertices and $2,088,968,416$ edges. They consume about 73GB of memory, including vertex and edge properties.

Figure 17 shows the complete distribution of vertex degrees (including both in- and out-edges). 80% vertices have a degree less than 10 while the largest vertex degree is over $78,000$.

5.2 Cycle Detection Queries

A variety of queries with different constraints are performed continuously against the dynamic graph. Specifically, we chose a representative continuous query from production and study its performance. It detects 6-hop cycles in the above graph, over a sliding window of 48 hours. There is an additional constraint on the edge to filter out certain types of transactions.

The query is used to monitor fake transactions involving particular types of user activity. A subgraph satisfying the constraint has 12,243,538 vertices and 33,826,783 edges. By leveraging the constraint, the system builds `HP-Index` from this smaller graph. However for query processing, the filtering is still performed by traversing the original graph at runtime.

To repeat experiments under varying conditions, we collect a trace of 500,000 updates from real production and use that for the rest of the evaluation.

5.3 Performance Evaluation

We evaluate the performance of the `HP-Index` in this subsection. Figure 18 and Figure 19 show the number of hot points and indexed paths (with memory sizes) respectively, where the degree threshold t varying from 10 to $5,120$. As expected, the number of hot points decreases when the threshold t increases. As expected, the number of hot points and index size decrease when the threshold t increases. Moreover, the index size is very small compared to the graph size under our settings.

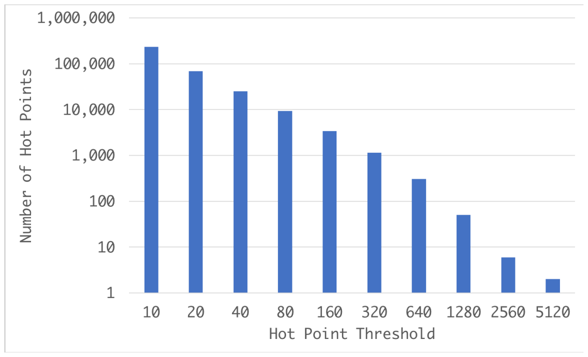


Figure 18: Number of hot points under different thresholds.

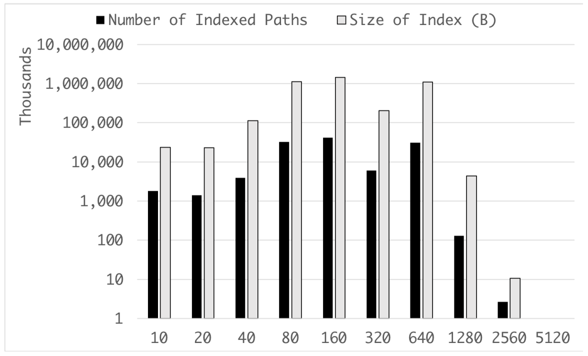


Figure 19: Number of indexed paths and memory sizes under different thresholds.

Figure 20 shows query response times using `HP-Index` under different thresholds t varying from 10 to 100. We report the 99.9% time, which is a commonly used metric for online system performance evaluation. Under each setting, we report the overall latency in Figure 20 and more detailed query response time (of Step 1, 2, and 3 as described in Section 3.3) and index maintenance time in Figure 21. Not surprisingly, the maintenance cost is close to zero and can hard seen from the figure. Surprisingly however, the index look-up time is quite significant and even dominate the total query time, especially when a search encounter more hot points (as it has to check all their combinations).

Even though, with the look-up cost in place, our algorithm can achieve a pretty good overall performance against the straightforward approach (in Algorithm 1) without index. Figure 22 details the performance comparison by showing a cumulative distribution of response times of the baseline algorithm and best numbers of using index (at $t = 40$). For

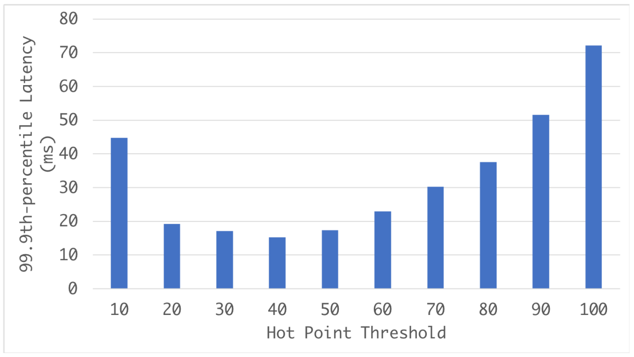


Figure 20: Overall latency using `HP-Index` under different thresholds.

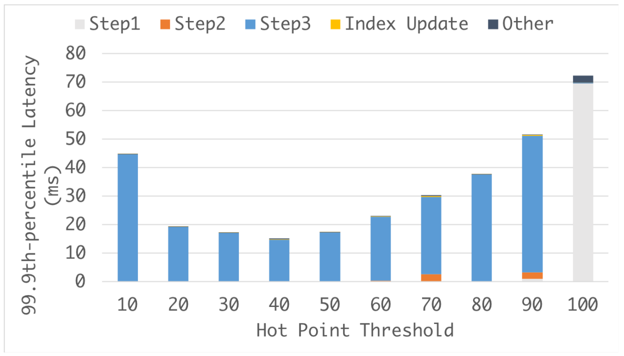


Figure 21: Detailed performance using `HP-Index` under different thresholds.

bad cases defined by 99.9% (and above) latency, `HP-Index` achieves an order of magnitude of improvement.

In Figure 23, we show the response times for the same update/query trace used in Figure 4. It is shown that our algorithm alleviates the latency spikes significantly.

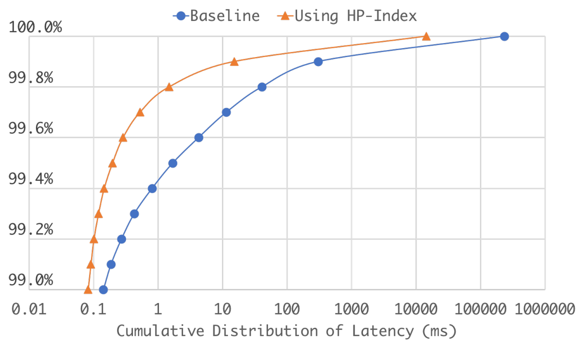


Figure 22: A comparison of query performance: baseline vs `HP-Index`.

5.4 Scalability

As described in Section 4, we leverage a thread pool to optimistically apply edge updates concurrently. Such design improves the system throughput and demonstrates good scalability. In Figure 24, we report the throughput of `GraphS` with the growth of the number of cores. It is shown that

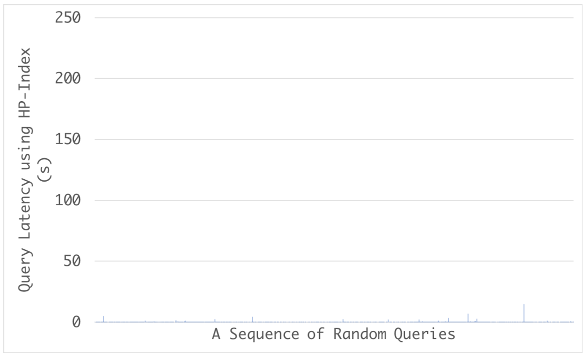


Figure 23: Latency spikes using using `HP-Index`.

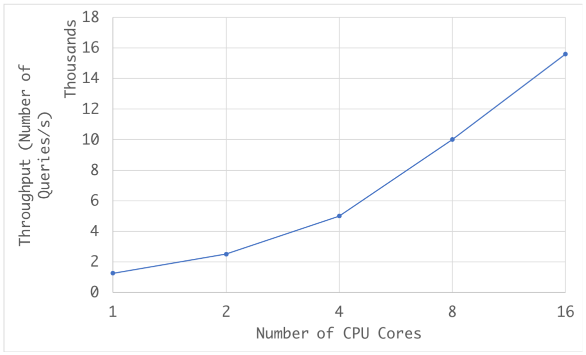


Figure 24: Dynamic scaling of input rate using multi-core.

`GraphS` scales almost linearly using multi-core to process incoming updates concurrently.

5.5 Coping with Graph Changes

Although we discuss the techniques to adjust the `HP-Index` against the involving of the graph in Section 3.4, the degree distribution of the network does not change much upon the incoming and expiration of the edges in our applications. Figure 25 confirms this by reporting the percent of hourly change of the number of hot points (including both addition and removal, at $t = 40$) over 24 hours.

6. RELATED WORK

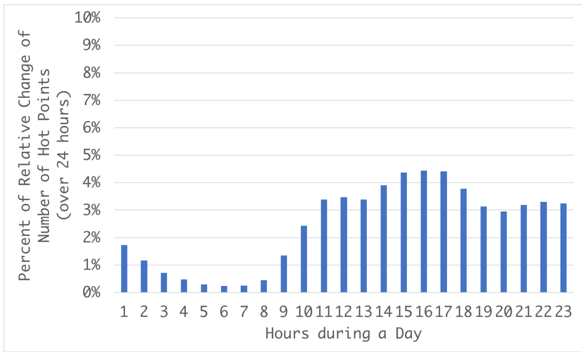


Figure 25: Percent of hourly change of number of hot points over 24 hours.

A large number of previous work focused on cycle enumeration, pattern match, and shortest path, etc. In this section, we briefly review closely related work.

6.1 Path and Cycle Enumeration

There is a long history of study on enumerating all simple paths or cycles on a graph [20, 15, 18]. Another line of research (e.g., [19]) is to count or estimate the number of paths between two given vertices, which is a well-known $\#P$ hard problem. But these techniques cannot be trivially extended to large scale dynamic graph to solve our problem in this paper due to the matrix operations involved or the sampling techniques for approximate solution. The problem of incremental cycle detection has been studied in the literature (e.g., [16, 22, 5]). Nevertheless, the technique proposed cannot be trivially extended to detect cycles with length constraint in large scale real-life network due to some stringent assumptions such as acyclic graph [22] and low-dimensional graph [16].

6.2 Pattern Match

Pattern match has been widely studied in the literature (See [10] for a survey) in a variety of computing environments. Some research efforts have been devoted to the problem of continuous exact pattern matching which aims to incrementally identify desired subgraph patterns upon the update of the graph. In [9], IncIsoMat is proposed to continuously identify subgraph matching upon the update of the graph where a candidate subgraph region is computed to reduce the research space. Graphflow [14] applies a worst-case optimal join algorithm to incrementally evaluate subgraph matching for each update. SJ-Tree [8] constructs the left-deep tree for the query graph and continuously maintain the partial matching in the tree nodes.

By regarding the cycles with length not larger than k as a set of patterns, we can apply the continuous pattern detection algorithms to identify length constraint cycles (LCCs). Nevertheless, since every path p with $2 \leq len(p) \leq k - 1$ may be a *partial* solution, it is infeasible to use the partial solution based techniques as such SJ-Tree [8].

6.3 Reachability and Shortest Path

Given two vertices u and v in a graph, the point-to-point reachability and shortest path queries are two of the most important types of queries in graph (see [27] and [23] for a survey). Existing main-memory algorithms working on reachability queries can be divided into two categories: Label-Only and Online-Search. Label-only approaches focus on compressing the graph transitivity to get a smaller index size for fast query processing. Recent studies include TF-Label [6] and DL [13]. Online-Search methods answer reachability queries by performing DFS from source vertex at run-time with help of a set of pruning strategies. Representative works include GRAIL [26], FERRARI [21], and IP+ [25]. Recently, a new DAG reduction approach is proposed in [28] to further speed up the reachability computation. Regarding the shortest path computation, the existing main-memory algorithms can also be classified into Label-only and Online-search techniques. In scale-free networks, the most efficient method for a distance query is based on the 2-hop index. Pruned Landmark Labeling [1] and Hop-Doubling Labeling [12] are two state-of-the-art algorithms. Recently, the computation of reachability and shortest path on dynamic graph have been investigated in [29] and [2], respectively.

None of the above works considers the length constraint. As to our best knowledge, [7] is the only existing work which investigate the reachability problem with k-length constraint, where a set of vertices in a vertex cover of the graph are chosen as hubs and the reachability of each pair of hubs is pre-calculated. Note that, the indexing techniques for reachability and shortest path may be used for pruning purpose in the computation of length constraint cycles (LCCs). It is immediate that the incoming edge (u, v) cannot contribute a k-path if v cannot reach u with in (k-1) hops or the shortest path from v to u is larger than k-1. In addition to that fact that we still need to continue the computation if the pruning fails, the expensive index maintenance cost makes these solutions infeasible in a large dynamic graph environment.

7. CONCLUSION

In this paper, we present a graph analytical system **GraphS** developed at Alibaba to efficiently detect constrained cycles in dynamic graphs. The system supports large graphs with hundreds of millions of edges and vertices, and allows ad-hoc continuous queries to identify constrained cycles in a fast changing graph in real-time. For each query, a hot point based index is constructed and efficiently maintained to greatly speed up cycle evaluation. In addition, the system utilizes optimized data layout and efficient memory management to improve performance and uses optimistic query evaluation to increase the system throughput. The **GraphS** system is deployed in production at Alibaba to actively monitor fraudulent activities based on cycle detection for several businesses. For a large dynamic graph with hundreds of millions of edges and vertices and a peak rate of tens of thousands of edge updates per second, the system is able to find newly formed cycles under a millisecond.

Although we focus on cycle detection in this paper, the **GraphS** system is able to support querying various structural patterns with complex constraints, for instance, tree-like graph patterns, etc. Pattern-specific indexes are needed for each query to speed up pattern evaluation whenever the graph changes. Another future work is to share indexes among different queries to minimize index maintenance cost without sacrificing individual query performance.

8. ACKNOWLEDGMENTS

We are grateful to our shepherd Zbigniew Jerzak for his guidance in the revision process and to the anonymous reviewers for their insightful comments. Youyang Yao provided feedback that helped improve the paper. Zilin Lee implemented the first version of the GraphS system including an efficient graph store, on which our work on HP-Index is based. We would also like to thank the members of the Sesame Credit team at Ant Financial, Bingxin Jia, Renxin Mao, Jingchao Zuo, and Kun Zhu, for providing the use cases and benchmarks and their contribution to the real production deployment. Finally, we would like to thank Alibaba Computing Platform team members for their support and collaboration.

9. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *the proceeding of ACM SIGMOD conference*, pages 349–360, 2013.

[2] T. Akiba, Y. Yano, and N. Mizuno. Hierarchical and dynamic k-path covers. In *the proceeding of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1543–1552, 2016.

[3] Apache Flink. http://flink.apache.org/.

[4] Apache Storm. http://storm.apache.org/.

[5] A. Bernstein and S. Chechik. Incremental topological sort and cycle detection in expected total time. In *SODA*, pages 21–34, 2018.

[6] J. Cheng, S. Huang, H. Wu, and A. W. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *the proceeding of ACM SIGMOD conference*, pages 193–204, 2013.

[7] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.

[8] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *the proceeding of the International Conference on Extending Database Technology (EDBT)*, pages 157–168, 2015.

[9] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *the proceeding of ACM SIGMOD conference*, pages 925–936, 2011.

[10] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. 6, 01 2006.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *the proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[12] M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.

[13] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

[14] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *the proceeding of ACM SIGMOD conference*, pages 1695–1698, 2017.

[15] A. A. Khan and H. Singh. Petri net approach to enumerate all simple paths in a graph. *Electronics Letters*, 16(8):291–292, 1980.

[16] S. R. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In *STOC*, pages 398–406, 1988.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *the proceeding of ACM SIGMOD conference*, pages 135–146, 2010.

[18] S. RAI and A. KUMAR. On path enumeration. *International Journal of Electronics*, 60(3):421–425, 1986.

[19] B. Roberts and D. P. Kroese. Estimating the number of s-t paths in a graph. *J. Graph Algorithms Appl.*, 11(1):195–214, 2007.

[20] F. Rubin. Enumerating all simple paths in a graph. *IEEE Transactions on Circuits and Systems*, 25(8):641–642, 1978.

[21] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In *the proceeding of the IEEE International Conference on Data Engineering (ICDE)*, pages 1009–1020, 2013.

[22] O. Shmueli. Dynamic cycle detection. *Inf. Process. Lett.*, 17(4):185–188, 1983.

[23] C. Sommer. Shortest-path queries in static networks. *ACM Comput. Surv.*, 46(4):45:1–45:31, 2014.

[24] The Rust Programming Language. https://www.rust-lang.org/.

[25] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 7(12):1191–1202, 2014.

[26] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[27] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. 2010.

[28] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang. DAG reduction: Fast answering reachability queries. In *the proceeding of ACM SIGMOD conference*, pages 375–390, 2017.

[29] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *the proceeding of ACM SIGMOD conference*, pages 1323–1334, 2014.