Towards Neighborhood Window Analytics over Large-Scale Graphs

Qi Fan[1(✉)], Zhengkui Wang[2], Chee-Yong Chan[3], and Kian-Lee Tan[1,3]

[1] NUS Graduate School for Integrative Science and Engineering, Singapore, Singapore
fan.qi@nus.edu.sg
[2] Singapore Institute of Technology, Singapore, Singapore
zhengkui.wang@singaporetech.edu.sg
[3] School of Computing, National University of Singapore, Singapore, Singapore
{chancy,tankl}@comp.nus.edu.sg

Abstract. Information networks are often modeled as graphs, where the vertices are associated with attributes. In this paper, we study neighborhood window analytics, namely k-hop window query, that aims to capture the properties of a local community involving the k-hop neighbors (defined on the graph structures) of each vertex. We develop a novel index, *Dense Block Index (DBIndex)*, to facilitate efficient processing of k-hop window queries. Extensive experimental studies conducted over both real and synthetic datasets with hundreds of millions of vertices and edges show that our proposed solutions are four orders of magnitude faster in query performance than the non-index algorithm, and are superior over the state-of-the-art solution in terms of both scalability and efficiency.

Keywords: Graph analytics · Graph window · Neighborhood aggregation

1 Introduction

Information networks such as social networks, biological networks and phone-call networks are typically modeled as graphs [4] where the vertices correspond to objects and the edges capture the relationships between these objects. For instance, in social networks, every user is represented by a vertex and the friendship between two users is reflected by an edge between the vertices. In addition, a user's profile can be maintained as the vertex's attributes. Such graphs contain a wealth of valuable information which can be analyzed to discover interesting patterns. For example, we can find the top-k influential users who can reach the most number of friends within 2 hops. With increasingly larger network sizes, there is an urgent need to develop effective and efficient mechanisms over large-scale graph data.

Recent research on graph analytics focuses on discovering the global graph properties and characteristics. To name a few, graph summarization [14] aims

© Springer International Publishing Switzerland 2016
S.B. Navathe et al. (Eds.): DASFAA 2016, Part II, LNCS 9643, pp. 201–217, 2016.
DOI: 10.1007/978-3-319-32049-6_13

to provide a compressed representation of a given graph based on its structure and vertex/edge attributes, while graph aggregation [4,17,19] focuses on aggregating the graph based on its vertex/edge attributes to discover the underlying characteristics of large graphs.

In this paper, we study a new type of query that analyzes each vertex's local community (e.g., neighborhoods) in a graph. To each vertex, these local communities (also referred to as windows in this paper) carry the most important information that captures the vertex's social influence and relations in the graph. Unlike graph summarization and aggregation that discover the entire graph's property, graph window queries (GWQs) explore the underlying characteristics of a small window related to each individual vertex. We identify one instantiation of graph "windows", namely *k-hop* window. We first demonstrate the k-hop window semantics with the following example.

Example 1 (K-hop window). *In social network scenario, it is of great interest to summarize the most relevant connections to each user such as the neighbors within 2-hops. Some analytic queries such as summarizing the related connections' distribution among different companies, and computing age distribution of the related friends can be useful. In order to answer these queries, collecting data from every user's neighborhoods within 2-hop is necessary.*

A k-hop window forms a window for one vertex by using its k-hop neighbors. In the Example 1, every user needs to gather data from his/her friends and friends-of-friends. k-hop neighbors are important to one vertex, as these are the vertices showing structural closeness as in Example 1.

To the best of our knowledge, existing graph databases or graph query languages do not directly support our proposed GWQ. There are two major challenges in processing GWQ. First, we need an efficient scheme to calculate the neighborhood window of each vertex. Second, we need efficient solutions to process the aggregation over a large number of windows that may overlap. However, it is nontrivial to address these two challenges. The state-of-the-art algorithm for k-hop like query is EAGR [12]. EAGR builds an overlay graph to leverage the shared components of different windows through multiple iterations. However, EAGR requires all vertices' k-hop neighbors to be pre-computed and resides in memory during every iteration. This heavily limits the efficiency and scalability of EAGR.For instance, a LiveJournal social network graph[1] (4.8 M vertices, 69M edges) generates over 100GB neighborhood information for k = 2 in adjacency list representation. In addition, the overlay graph construction is not a one-time task, but is periodically performed after a certain number of structural updates in order to maintain the overlay quality. The high memory requirement renders EAGR impractical when k and the graph size increase.

In this paper, we propose Dense Block Index (DBIndex), which enables an efficient query processing by integrating the optimized query execution plan for shared aggregation computation. Additionally, for index construction, we apply a hash-based technique to cluster the vertices based on the window similarity,

[1] Available at http://snap.stanford.edu/data/index.html, which is used in [12].

which ensures memory efficiency. On the basis of the clusters, we further develop different optimizations to extract the shared components efficiently.

Our contributions are summarized as follows:

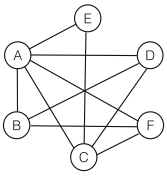
- We introduce a new type of graph analytic query, *Graph Window Query* and formally define the *k-hop* window. We illustrate how these window queries would help users better query and understand the graphs under different semantics.
- We propose *Dense Block Index (DBIndex)* to support the proposed k-hop window queries. The index integrates the window aggregation sharing techniques to salvage partial work done to enable efficient query processing over large-scale graphs.
- We perform extensive experiments over both real and synthetic datasets with hundreds of millions of vertices and edges on a single machine. Our experiments indicate that our proposed index-based algorithms outperform the naive non-index algorithm by up to four orders of magnitude. In addition, our experiments also show that DBIndex is superior over EAGR in terms of both scalability and efficiency.

The rest of the paper is organized as follows: Sect. 2 formulates the GWQ. In Sect. 3, we introduce the DBIndex for k-hop window query. Section 4 presents the experimental evaluations. In Sects. 5 and 6, we provide the related works and the conclusion respectively.

2 Problem Formulation

In this section, we provide the formal definition of graph window query. We use $G = (V, E)$ to denote a directed/undirected data graph, where V is its vertex set and E is its edge set. Each vertex/edge is associated with a (possibly empty) set of attributes.

Figure 1 shows an undirected graph representing a social network that we will use as our running example. The table shows the values of the five attributes (User, Age, Gender, Industry, and Number of posts) associated with each vertex. For convenience, each vertex is labeled with its user attribute value; and there is one edge between a user X and another user Y if X and Y are connected in the social network.



(a)

User	Age	Gender	Industry	Posts
A	21	M	IT	12
B	26	F	IT	15
C	30	F	Finance	28
D	22	M	Finance	23
E	28	M	Power	26
F	23	F	Power	14

(b)

Fig. 1. Example of Social Graph. (a) Graph structure; (b) Attributes associated with the vertices in (a).

Given a data graph $G = (V, E)$, a *Graph Window Function (GWF)* over G can be expressed as a quadruple (G, W, Σ, A), where $W(v)$ denotes a *window specification* (or *window* for short) for a vertex $v \in V$ that determines the set of vertices (refer to as *nodes*) in some subgraph of G, Σ denotes an *aggregate function*, and A denotes a *vertex attribute*. The evaluation of a GWF (G, W, Σ, A) on G

computes for each vertex v in G, the aggregation Σ on the values of attribute A over all the vertices in $W(v)$, which is denoted by $\Sigma_{v' \in W(v)} v'.A$. We consider all common aggregate functions (e.g., sum, count, average, max, min etc.) in this paper.

Definition 1 (*k-hop Window).* *Given a vertex v in a graph G, the k-hop window of v, denoted by $W_{kh}(v)$ (or $W(v)$ when there is no ambiguity), is the set of neighbors of v in G which can be reached within k hops. For an undirected graph G, a vertex u is in $W_{kh}(v)$ iff there is a α-hop path between u and v where $\alpha \leqslant k$. For a directed graph G, a vertex u is in $W_{kh}(v)$ iff there is a α-hop directed path from v to u*[2] *where $\alpha \leqslant k$.*

Intuitively, a k-hop window selects the neighboring vertices of a vertex within a k-hop distance. These neighboring vertices typically represent the most important vertices to a vertex wrt their structural relationship in a graph. Thus, k-hop windows provide meaningful specifications for many applications, such as customer behavior analysis [1,6] and digital marketing [10]. As an example, in Fig. 1, the 1-hop window of vertex E is $\{A, C, E\}$ and the 2-hop window of vertex E is $\{A, B, C, D, E, F\}$.

We emphasize that there are different types of windows which can be formalized under different application scenarios. For instance, we have identified another useful window, namely the *topological window*, which captures the set of ancestor vertices of each vertex in a directed acyclic graph (DAG). There are many DAGs in real-world applications (such as biological networks, citation networks and dependency networks) where topological windows represent meaningful relationships that are of interest. For example, in a citation network where (X,Y) is an edge iff paper X cites paper Y, the topological window of a paper represents the citation impact of that paper [3,11]. Based on the topological window, we have proposed another index, *inheritance index* as in our technical report [8] to facilitate an efficient topological window query processing and systematically evaluated the index. In general, a graph window query is defined as:

Definition 2 (Graph Window Query). *A graph window query on a data graph G is of the form $GWQ(G, W_1, \Sigma_1, A_1, \cdots, W_m, \Sigma_m, A_m)$, where $m \geq 1$ and each quadruple (G, W_i, Σ_i, A_i) is a graph window function on G.*

In this paper, we focus on efficiently processing k-hop window queries with indexes. Due to space constraint, we only present the static solution for illustration, and the strategy for handling updates are described in our technical report [8].

[2] Other variants of k-hop window for directed graphs are possible; e.g., a vertex u is in $W_{kh}(v)$ iff there is a α-hop directed path from u to v where $\alpha \leqslant k$.

3 Dense Block Index

A straightforward approach to process a graph window query $Q = (G, W, \Sigma, A)$, where $G = (V, E)$, is to dynamically compute the window $W(v)$ for each vertex $v \in V$ and its aggregation $\Sigma_{v' \in W(v)} v'$. A independently from other vertices. We refer to this approach as *Non-Indexed* method. Given that many of the windows would share many common nodes (e.g., the k-hop windows of two adjacent vertices), such a simple approach would be very inefficient due to the lack of sharing of the aggregation computations.

To efficiently evaluate graph window queries, we propose an indexing technique named *Dense Block Index* (*DBIndex*), which is both space and query efficient. The main idea of DBIndex is to try to reduce the aggregation computation cost by identifying subsets of nodes that are shared by more than one window so that the aggregation for the shared nodes could be computed only once instead of multiple times.

For example, consider a graph window query on the social graph in Fig. 1 using the 1-hop window. We have $W(B) = \{A, B, D, F\}$ and $W(C) = \{A, C, D, E, F\}$ sharing three common nodes A, D, and F. By identifying the set of common nodes $S = \{A, D, F\}$, its aggregation $\Sigma_{v \in S} v.A$ can be computed only once and then reuse to compute $\Sigma_{v \in W(B)} v.A$ and $\Sigma_{v \in W(C)} v.A$.

Given a window W and a graph $G = (V, E)$, we refer to a non-empty subset $B \subseteq V$ as a *block*. Moreover, if B contains at least two nodes and B is contained by at least two different windows (i.e., if $|B| \geq 2$, and $\exists v_1 \neq v_2 \in V$, $B \subseteq W(v_1)$, and $B \subseteq W(v_2)$), then B is a *dense block*. Thus, in the last example, $\{A, D, F\}$ is a dense block.

We say that a window $W(X)$ is *covered* by a collection of disjoint blocks $\{B_1, \cdots, B_n\}$ if the set of nodes in the window $W(X)$ equals to the union of all nodes in the collection of disjoint blocks; i.e., $W(X) = \bigcup_{i=1}^{n} B_i$ and $B_i \cap B_j = \emptyset$ if $i \neq j$.

To maximize the sharing of aggregation computations for a graph window query, the objective of DBIndex is to identify a small set of blocks $\mathcal{B}$ such that for each $v \in V$, $W(v)$ is covered by a small subset of disjoint blocks in $\mathcal{B}$. Clearly, the cardinality of $\mathcal{B}$ is minimized if $\mathcal{B}$ contains a few large dense blocks.

Thus, given a window W and a graph $G = (V, E)$, a DBIndex to evaluate W on G consists of three components in the form of a bipartite graph. The first component is a collection of vertex (i.e., V); the second component is a collection of blocks $\mathcal{B} = \{B_1, \cdots, B_n\}$ where each $B_i \subseteq V$; and the third component is a collection of links from blocks to vertices such that if a set of blocks $B(v) \subseteq \mathcal{B}$ is linked to a vertex $v \in V$, then $W(v)$ is covered by $B(v)$. Note that a DBIndex is independent of both the aggregate function (i.e., Σ) and the attribute to be aggregated (i.e., A). Figure 2(d) shows an example of a DBIndex wrt the social graph in Fig. 1 and the 1-hop window. There are three dense blocks detected which are $\{A, F, D\}$, $\{C, E\}$, and $\{A, C\}$.

3.1 Query Processing Using DBIndex

Given a DBIndex wrt a graph G and a window W, a graph window query $Q = (G, W, \Sigma, A)$ is processed by the following two steps. First, for each block B_i in the index, we compute the aggregation (denoted by T_i) over all the nodes in B_i; i.e., $T_i = \Sigma_{v \in B_i} v.A$. Thus, each T_i is a partial aggregate value. Next, for each window $W(v)$, $v \in V$, the aggregation for the window is computed by aggregating over all the partial aggregates associated with the blocks linked to $W(v)$; i.e., if $B(v)$ is the collection of blocks linked to $W(v)$, then the aggregation for $W(v)$ is given by $\Sigma_{B_i \in B(v)} T_i$.

3.2 DBIndex Construction

In this section, we discuss the construction of the DBIndex (wrt a graph $G = (V, E)$ and window W) which has two key challenges.

The first challenge is the time complexity of the index construction. From our discussion of query processing using DBIndex, we note that the number of aggregation computations is determined by both the number of blocks as well as the number of links in the index; the former determines the number of partial aggregates to compute while the latter determines the number of aggregations of the partial aggregate values. Thus, to maximize the shared aggregation computations using DBIndex, both the number of blocks in the index as well as the number of blocks covering each window should be minimized. However, finding the optimal DBIndex to minimize this objective is NP-hard[3]. Therefore, efficient heuristics are needed to construct the DBIndex.

The second challenge is the space complexity of the index construction. In order to identify large dense blocks to optimize for query efficiency, a straightforward approach is to first derive the window $W(v)$ for each vertex $v \in V$ and then use this derived information to identify large dense blocks. However, this direct approach incurs a high space complexity of $O(|V|^2)$. Therefore, a more space-efficient approach is needed in order to scale to large graphs.

MinHash-based Index Construction (MC). To reduce both the time and space complexities for the index construction, instead of trying to identify large dense blocks among a large collection of windows, MC first partitions all the windows into a number of smaller clusters of similar windows and then identifies large dense blocks from each of the smaller clusters. Intuitively, two windows are considered to be highly similar if they share a larger subset of nodes. We apply the well-known *MinHash based Clustering* algorithm [2] to partition the windows into clusters of similar windows. The MinHash clustering algorithm uses *Jaccard Coefficient* to measure the similarity of two sets. Given two windows $W(v)$ and $W(u)$, $u, v \in V$, their *Jaccard Coefficient* is given by $J(u, v) = \frac{|W(u) \cap W(v)|}{|W(u) \cup W(v)|}$. The *Jaccard Coefficient* ranges from 0 to 1, where a larger value means that the windows are more similar.

[3] Note that a simpler variation of our optimization problem has been proven to be NP-hard [16].

Our heuristic approach to construct DBIndex I operates as in Algorithms 1 and 2. Let $vertices(I)$, $blocks(I)$, and $links(I)$ denote the collection of vertices, blocks, and links in I. Initially, we have $vertices(I) = V$, $blocks(I) = \emptyset$, and $links(I) = \emptyset$.

Algorithm 1. CreateDBIndex

Require: Graph $G = (V, E)$, window W
Ensure: DBIndex I
1: Initialize DBIndex I: $vertices(I) = V$, $blocks(I) = \emptyset$, $links(I) = \emptyset$
2: **for all** $v \in V$ **do**
3: Traverse G to determine $W(v)$
4: Compute the hash signature $H(v)$ for $W(v)$
5: **end for**
6: Partition V into clusters $\mathcal{C} = \{C_1, C_2, \cdots\}$ based on hash signatures $H(v)$
7: **for all** $C_i \in \mathcal{C}$ **do**
8: **for all** $v \in C_i$ **do**
9: Traverse G to determine $W(v)$
10: **end for**
11: `IdentifyDenseBlocks` (I, W, C_i)
12: **end for**
13: **return** I
14:

Algorithm 2. IdentifyDenseBlocks

Require: DBIndex I, window W, a cluster $C_i \subseteq V$
1: Return if C_i is empty.
2: Partition V into blocks wrt to C_i, $DenseNodes = \emptyset$
3: **for all** dense block B **do**
4: Insert B into $blocks(I)$ if $B \notin blocks(I)$
5: Insert (B, v) into $links(I)$ for each $v \in C_i$ where $B \subseteq W(v)$
6: $DenseNodes = DenseNodes \cup B$
7: **end for**
8: $C_n \leftarrow \emptyset$, $W_n \leftarrow \emptyset$
9: **for all** $v_i \in C_i$ **do**
10: **if** $(W(v_i) - DenseNodes \neq \emptyset)$ **then**
11: Insert v_i to C_n
12: Insert $(W(v_i) - DenseNodes)$ to W_n
13: **end if**
14: **end for**
15: `IdentifyDenseBlocks` (I, W_n, C_n)
16: **return**

The first step (Lines 1–6 Algorithm 1) is to partition the vertices in V into clusters using MinHash algorithm such that vertices with similar windows belong to the same cluster. For each vertex $v \in V$, we first derive its window $W(v)$ by an appropriate traversal (e.g., k-hop BFS) of the graph G. Next, we compute hash signatures (denoted by $H(v)$) for each v by applying MinHash on $W(v)$. Vertices with identical hash signatures are considered to have highly similar windows and are grouped into the same cluster. To ensure that our approach is scalable, we do not retain $W(v)$ in memory after its hash signature $H(v)$ has been computed and used to cluster v; i.e., our approach does not materialize all the windows in the memory to avoid high space complexity. Let $\mathcal{C} = \{C_1, C_2, \cdots\}$ denotes the collection of clusters obtained from the first step, where each C_i is a subset of vertices.

The second step (Lines 7–12 Algorithm 1) is to identify dense blocks from each of the clusters computed in the first step. The identification of dense blocks in each cluster C_i is based on the notion of node equivalence defined as follows. Two distinct nodes $u, v \in V$ are defined to be equivalent (denoted by $u \equiv v$) wrt C_i iff u and v are both contained in the same set of windows wrt C_i; i.e., for every window $W(x), x \in C_i$, $u \in W(x)$ iff $v \in W(x)$. Based on this notion of node equivalence, V is partitioned into blocks of equivalent nodes. To perform

this partitioning, we need to again traverse the graph for each vertex $v \in C_i$ to determine its window $W(v)$[4].

However, since C_i is now a smaller cluster of vertices, we can now materialize all the windows for the vertices in C_i in memory without exceeding the memory space. In the event that a cluster C_i is still too large for all its windows to be materialized in main memory, we can further partition C_i into equal sized sub-clusters. This re-partition process can be recursively performed until the sub-clusters created are small enough such that the windows for all vertices in the sub-cluster fit in memory.

Recall that a block B is a dense block if B contains at least two nodes and B is contained in at least two windows. Thus, we can classify nodes in V as either dense or non-dense nodes: a node $v \in V$ is classified as a *dense node* if v is contained in a dense block; otherwise, v is a *non-dense node*.

For each dense block B in C_i, we update the blocks and links in the DBIndex I recursively as follows: If the current cluster or window only contains one element, then algorithm stops. Otherwise, we insert dense block B into $block(I)$; and we insert (B, v) into $links(I)$ for each $v \in C_i$ where $B \subseteq W(v)$ (Lines 3–7 Algorithm 2). For each vertex v in C_i, we remove dense nodes from its window $W(v)$. This forms the refined window $W_n(v)$. If $W_n(v)$ is not empty, we then add v to a refined cluster C_n. C_n and W_n are then processed recursively (Lines 8–15 Algorithm 2).

Figure 2 illustrates the construction of the DBIndex wrt the social graph in Fig. 1(a) and 1-hop window using the MC algorithm. First, the set of graph vertices are partitioned into clusters using MinHash clustering; Fig. 2(a) shows that the set of vertices $V = \{A, B, C, D, E, F\}$ are partitioned into two clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E, F\}$. Table 1 in Fig. 2(b) shows the node-vertex mapping in C_1, i.e. for each node $u \in V$, the corresponding row is the set $\{v \in C_1 | u \in W(v)\}$. Similarly, Table 2 in Fig. 2(b) shows the node-vertex mapping in C_2.

Consider the identification of dense blocks in cluster C_1. As shown in Fig. 2(c), based on the notion of equivalence nodes, cluster C_1 is partitioned into three blocks of equivalent nodes: $B_1 = \{A, D, F\}$, $B_2 = \{B\}$, and $B_3\{C, E\}$. Among these three blocks, only B_1 and B_3 are dense blocks. The MC algorithm then tries to repartition the window A, B, C using non-dense nodes in C_1, (i.e., B) as next window. Since B is the only node, it directly outputs. At the end of processing cluster C_1, the DBIndex I is updated as follows: $blocks(I) = \{B_1, B_2, B_3\}$ and $links(I) = \{(B_1, \{A, B, C\}), (B_2, \{A, B\}), (B_3, \{A, C\})\}$. The identification of dense blocks in cluster C_2 is of similar process.

[4] Note that although we could have avoided deriving $W(v)$ a second time if we had materialized all the derived windows the first time, our approach is designed to avoid the space complexity of materializing all the windows in memory at the cost of computing each $W(v)$ twice. We present an optimization later in this section to avoid the recomputation cost on k-hop window query.

Assume that, the average neighborhood size of each vertex is $\overline{w}$. The MinHash cost is thus $\overline{w}|V|$. The cost of traversal for all vertex is $\overline{w}|E|$. In Algorithm 1, Lines 1–10 have the cost of $\overline{w}(|V|+2|E|)$; In Algorithm 2, since we can simply partition nodes using hashing, the time cost is thus $\overline{w}|C_i|$. The recursive procedure runs at most $log(|C_i|)$ times, and the total cost for Algorithm 2 is $\Sigma(\overline{w}|C_i|log(|C_i|))$. Since $\Sigma(|C_i|) = |V|$, the total cost for Algorithm 2 is less than $\overline{w}|V|log(|V|)$. Therefore, the total cost for Algorithm 1 and Algorithm 2 is $\overline{w}(|V| + 2|E| + |V|log(|V|))$, thus the complexity is $O(\overline{w}(|E| + |V|log(|V|)))$.

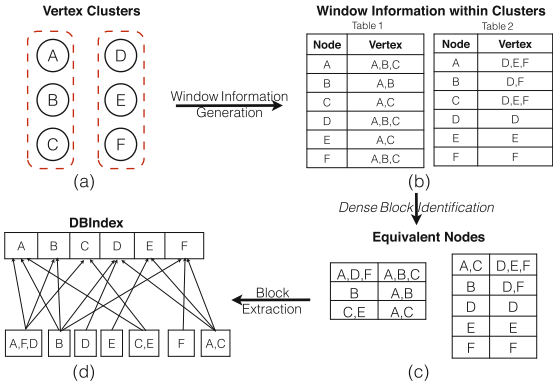


Fig. 2. DBIndex Construction over Social Graph in Fig. 1. (a) Two clusters after MinHash clustering; (b) Window information of involved vertices within each cluster; (c) Dense blocks within each cluster; (d) Final DBIndex.

Next, we provide two specific optimizations for constructing DB-Index for k-hop window queries.

Estimation Optimization. For k-hop window query with a large value of k, the cost of graph traversals to compute the k-hop windows is high. Moreover, the cost of initial MinHash in MC approach equals to the initial number of vertex-window mappings, which is of the same order as graph traversal.

To address the high computation issue, we make an observation that if the m-hop windows for two vertices are similar, the n-hop windows for them are also similar, where $m < n$. The intuition is that the shared component becomes larger via hop expansion. This observation is formally described as follows:

Theorem 1. *Let $\langle u, v\rangle$ be a randomly chosen vertex pair from a graph, let $J_k(u, v)$ be their Jaccard similarity wrt k-hop window. Then with high probability $J_m(u, v) \leq J_n(u, v)$, where $m < n$.*

We omit the theoretic proof here. Interested readers are referred to [8] for details. Based on Theorem 1, one optimization to improve the efficiency of Algorithm 1 with the tradeoff of a possible lower "quality" dense blocks (in terms of their sizes) is to use the m-hop window as an estimation for a n-hop index construction during the clustering, where $m < n$. In particular, for the first round of window computation (Lines 3–4 in Algorithm 1), we can use the hash signatures of the lower hop windows cluster the vertices in V to approximate k-hop windows. This approximation has the advantage of improved time-efficiency as traversal and MinHash clustering on lower-hop window is significantly faster. In particular, if the average number of neighbors for each vertex

in a n-hop window is denoted by $\overline{w}_n$, then the optimization reduces the index construction cost by $(|V|+|E|)(\overline{w}_n - \overline{w}_m)$ from $(|V|+2|E|+|V|log(|V|))\overline{w}_n$ to $(|V|+|E|)\overline{w}_m + (|E|+|V|(log|V|))\overline{w}_n$. This improvement is significant as $\overline{w}_n$ is exponentially greater than $\overline{w}_m$, where $m < n$, in k-hop windows. Our experimental results show with this optimization, the reduction in the quality of dense blocks is actually only marginal which makes this optimization a good tradeoff.

Batching Optimization. When multiple k-hop indexes are required, one applicable optimization is to batch the index constructions to share the graph traversal and clustering. Suppose there is a need to compute the DBIndex for 1-hop, 2-hop, ..., k-hop windows. Constructing each index independently would incur high overhead on both clustering and graph traversal which can be alleviated by batching their computations. The overall idea of the batching construction is to utilize the lower hop (e.g., 1-hop) traversal information to build the clustering and reuse it for all the h-higher hops. In addition, the second time graph traversal after obtaining the clustering can also be shared. Intuitively, while we expand the k-hop window, we can calculate the i-hop window as well. This can be achieved as the BFS is adopted where the $(i+1)$-hop window can be directly derived based on the i-hop windows, thus the traversal overhead can be shared.

4 Experimental Evaluation

In this section, we present the results of our experiments on both real-world networks and synthetic graphs. Due to space limitations, we can only present partial experimental results here and more results can be found in our technical report [8].

All experiments are conducted on an Amazon EC2 r3.2xlarge machine[5], with an 8-core 2.5 GHz CPU, 60 GB memory and 320 GB hard drive running with 64-bit Ubuntu 12.04. We implement EAGR algorithm as a reference in our comparative study. All algorithms are implemented in Java and run under JRE 1.6.

Name	Type	# of Vertices	# of Edges
LiveJournal	undirected	3,997,962	34,681,189
Pokec	directed	1,632,803	30,622,564
Orkut	undirected	3,072,441	117,185,083
DBLP	undirected	317,080	1,049,866
YouTube	undirected	1,134,890	2,987,624
Google	directed	875,713	5,105,039
Amazon	undirected	334,863	925,872
Stanford-web	directed	281,903	2,312,497

Fig. 3. Large-scale Real Datasets

Datasets. For real datasets, we use 8 information networks which are available at the Stanford *SNAP* website[6]: The detail description of these datasets is provided in Fig. 3. For synthetic datasets, we use *SNAP* graph generator to create graphs with various sizes.

[5] http://aws.amazon.com/ec2/pricing/.
[6] http://snap.stanford.edu/snap/index.html.

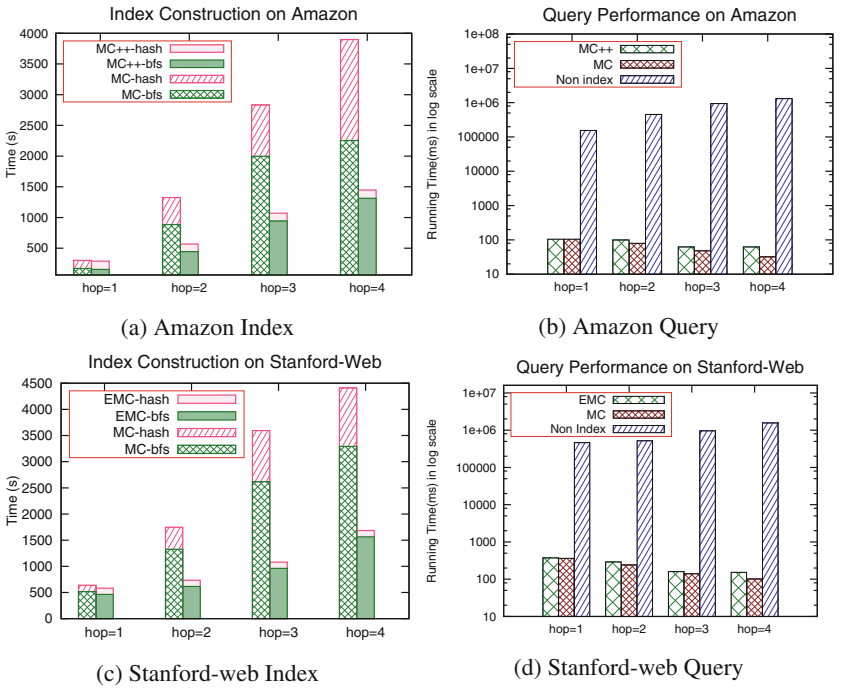
Query. In all the experiments, the window query is conducted by using SUM() as the aggregate function.

4.1 Index Construction Optimization

To study the performance of index construction, we compare two indexing methods, namely MC and MC++. MC method uses the MinHash clustering as described in Algorithm 1 while MC++ adapted the estimation optimization as in Theorem 1. We then present the results on the Amazon and Stanford-web graphs for a series of k-hop queries.

Index Construction. Figure 4(a) and (c) compare the index construction time between MC and MC++ when we vary the windows from 1-hop to 4-hop under Amazon and Stanford-web datasets. To better understand the time difference, the construction time is split into two parts: the MinHash cost (MC++-hash or MC-hash) and the BFS traversal (to compute the k-hop window) cost (MC++-bfs or MC-bfs). The results show the same trend for the two datasets. We made several observations. First, as the number of hops increases, the indexing time increases as well. This is expected as a larger hop count results in a larger window size and the BFS and MinHash computation time increase correspondingly. Second, as the hop count increases, the difference between the index time of MC++ and that of MC widens. For instance, as shown in Fig. 4(a), for the 4-hop window queries, compared to MC, MC++ can save 62 % construction time. MC++ benefits from both the low MinHash cost and low BFS cost. From Fig. 4(a), we can see that the MinHash cost of MC increases as the number of hops increases, while that for MC++ remains almost the same as the 1-hop case. The similar pattern can be found in Fig. 4(c) as well. These show that the cost of MinHash becomes more significant for larger windows. Thus, using 1-hop clustering for larger hop counts reduces the MinHash cost in MC++. Similarly, as MC++ saves on BFS cost for k-hop queries where $k > 1$, the BFS cost of MC++ is much smaller than that of MC as well.

Query Performance. Figure 4(b) and (d) present the query time of MC and MC++ on Amazon and Stanford-web datasets as we vary the number of hops from 1 to 4. To appreciate the benefits of an index-based scheme, we also implemented a *Non-indexed* algorithm which computes window aggregate by performing k-hop breadth first search for each vertex individually in real time. In Fig. 4(b), the execution time shown on the y-axis is in log scale. The results show that the index-based schemes outperform the non-index approach by four orders of magnitude. For instance, for the 4-hop query, our algorithm is 13,000 times faster than the non-index approach. This confirms that it is necessary to have well-designed index support for efficient window query processing. By utilizing DBIndex, for these graphs with millions of edges, every aggregation query can be processed in just between 30 ms to 100 ms. In addition, we can see that as the number of hops increases, the query time decreases. This is the case because a larger hop count eventually results in a larger number of dense blocks where more (shared) computation can be salvaged. Furthermore, we can see that the



(a) Amazon Index (b) Amazon Query

(c) Stanford-web Index (d) Stanford-web Query

Fig. 4. The evaluation of the index construction optimization.

query time of MC++ is slightly longer than that of MC when the number of hops is large. This is expected as MC++ does not cluster based on the complete window information; instead, it uses only partial information derived from the 1-hop windows. However, the performance difference is quite small even for 4-hop queries - the difference is only 20 ms. For small number of hops, the time difference is even smaller. This performance penalty is acceptable as tens of milliseconds time difference will not affect user's experience. As MC++ is significantly more efficient than MC in index construction, MC++ may still be a promising solution for many applications. In addition, we also observe the same pattern in Fig. 4(d). As such, in the following sections, we adopt MC++ for DBIndex in our experimental evaluations.

4.2 Comparison Between DBIndex and EAGR

We then compare DBIndex and EAGR [12][7] using both real and synthetic datasets.

Real Datasets. We first study the index construction and query time performance of DBIndex and EAGR for 1-hop and 2-hop windows using 6 real

[7] As in [12], for each dataset, EAGR is run for 10 iterations in the index construction.

datasets: DBLP, YouTube, Livejournal, Google, Pokec and Orkut. The results for 1-hop window and 2-hop window are presented in Fig. 5(a)-(d). As shown in Fig. 5(a) and (c) both DBIndex and EAGR can build the index for all the real datasets for 1-hop but EAGR ran out of the memory for 2-hop window queries on LiveJournal and Orkut datasets. This further confirms that EAGR incurs high memory usage as it needs to maintain the vertex-window mapping information. We also observe that DBIndex is significantly faster than EAGR in index creation. For instance, for Orkut dataset, EAGR takes 4 hours to build the index while DBIndex only takes 33 min.

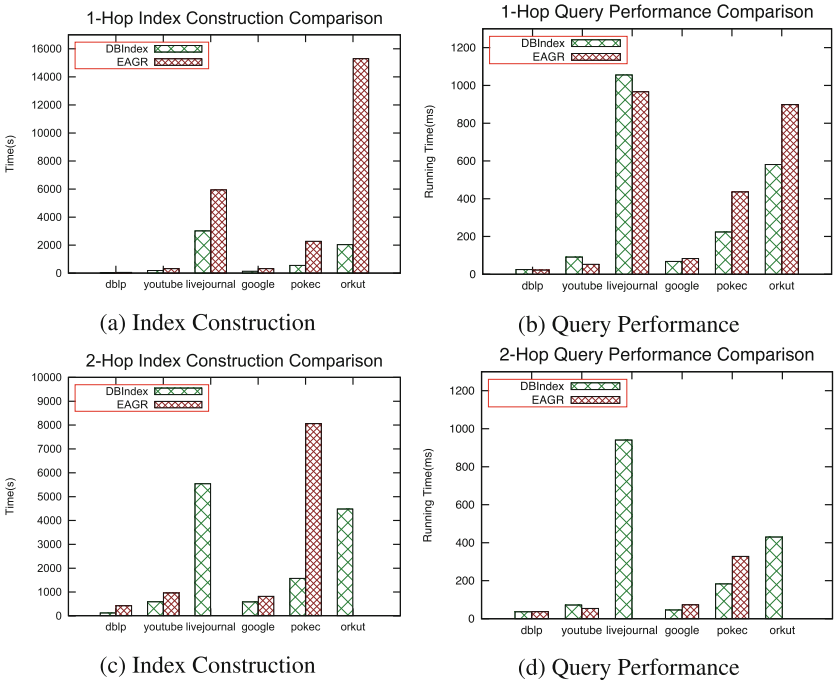


Fig. 5. DBIndex VS. EAGR (a)(b) are for 1-hop queries;(c)(d) are for 2-hop queries

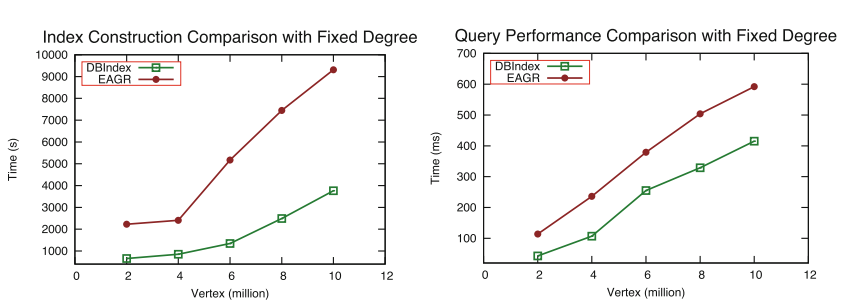
Figure 5(b) and (d) show the query performance for 1-hop and 2-hop queries respectively. The results indicate that the query performance is comparable. For most of the datasets, DBIndex is faster than EAGR. In some datasets (e.g., Orkut and Pokec), DBIndex performs 30 % faster than EAGR. We see that, for 1-hop queries on YouTube and LiveJournal datasets and 2-hop queries on YouTube dataset, DBIndex is slightly slower than EAGR. We observe that these datasets are very sparse graphs where the intersections among windows are naturally small. For very sparse graphs, both DBIndex and EAGR are unable to find much computation sharing. In this case, the performance of DBIndex and EAGR is very close. For instance, in the worst case, as in the livejournal dataset,

DBIndex is 9 % slower than EAGR where the actual time difference remains tens of milliseconds.

We also observe that, both algorithms process 2-hop queries faster than 1-hop queries. This is because there is more computation sharing for 2-hop window query. In summary, DBIndex takes much shorter time to build but offers comparable, if not much faster, query performance than EAGR.

Synthetic Datasets. We generated synthetic datasets using the SNAP generator to study the scalability of DBIndex.

Impact of Number of Vertices. First, we study how the performance changes when we fix the degree [8] at 10 and vary the number of vertices from 2 M to 10 M. Figure 6(a) and (b) show the execution time for index construction and query performance respectively. From the results, we can see that DBIndex outperforms EAGR in both index construction and query performance. For the graph with 10 M vertices and 100 M edges, the DBIndex query time is less than 450 milliseconds. Moreover, when the number of vertices changes from 2M to 10M, the query performance only increases 3 times. This shows that DBIndex is not only scalable, but offers acceptable performance.

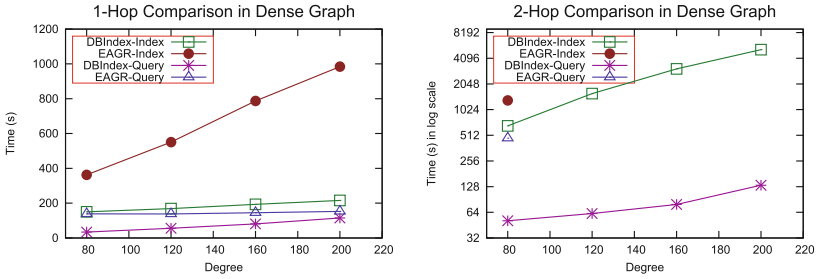


(a) Index Construction (b) Query Performance

Fig. 6. Impact of number of vertices

Impact of Degree. Our proposed DBIndex is effective when there are significant overlaps between windows of neighboring vertices. As such, it is interesting to study how it performs under various graph degrees. Here, we report the results from dense graphs. More results of sparse graphs can be found in the technical report [8]. We fix the number of vertices in a graph to be 200k. and then vary the degree from 80 to 200. Figure 7(a) shows the index construction and query time for 1-hop query. We can see that DBIndex outperforms EAGR significantly. As the degree increases, EAGR's performance degrades much faster than DBIndex.

[8] Degree means average degree of the graph. The generated graph is of Erdos-Renyi model .



(a) 1-Hop Comparison (b) 2-Hop Comparison

Fig. 7. Impact of Degree on Dense Graphs

It is notable that DBIndex indexing time almost matches EAGR's query time. Figure 7(b) shows the comparison under 2-hop queries. EAGR is only able to work on the dataset with degree 80 due to the memory issue. This is because the number of edges is large (e.g., 40 M edges for a graph of degree 200).

In summary, the insight we obtain is that the scalability of EAGR is highly limited by two factors: the graph size and the number of hops. DBIndex achieves better scalability as it does not need to create a large amount of intermediate data in memory.

5 Related Work

GWFs are different from graph aggregation [4,17,19] in graph OLAP. In graph OLAP, information in a graph are summarized by partitioning the graph's vertices/edges (based on some attribute values) and computing aggregate values for each partition. GWFs, on the other hand, compute aggregate values for each graph vertex wrt the subgraph associated with the vertex. Indeed, such differences also arise in the relational context, where different techniques are developed to evaluate OLAP and window function queries.

In [18], the authors investigated the problem of finding the vertices that have top-k highest aggregate values over their h-hop neighbors. They proposed mechanisms to prune the computation by using two properties: First, the locality between vertices is used to propagate the upper bound of aggregation; Second, the upper bound of aggregates can be estimated from the distribution of attribute values. However, all these pruning techniques are not applicable in our work, as we need to compute the aggregation for every vertex. In such a scenario, techniques in [18] degrade to the non-indexed approach as described in Sect. 4.

Indexing techniques have been proposed to efficiently determine whether a pair of vertices is within a distance of k-hops (e.g., k-reach index [5]). However, such techniques are not suitable for k-hop window queries due to the time complexity of $O(n^2)$ in determining each vertex's window. Moreover, such techniques do not leverage shared components among windows to boost query processing.

In distributed databases community, some works considered utilizing partial aggregates to facilitate efficient aggregate computation (e.g., [9,15]). However, their primary goal is to optimize the communication cost between sites, hence the optimization problem is fundamentally different from ours.

In network science community, *egocentric* analysis is emerging in recent years. However, their main focus is on structural analyses of a vertex's k-hop neighborhood. For example, Everett et al. [7] looked at finding the betweenness centrality among vertices' k-hop neighbors; and Moustafa et al. [13] developed techniques for matching specialized patterns among k-hop neighborhoods. These works are different from ours as they do not consider attribute aggregation.

The work that is most related to ours is [12] - referred to as EAGR which examined the evaluation of egocentric (similar to our k-hop window) aggregation queries. EAGR and our DBIndex share the similar spirit in terms of discovering the shared components among different windows to speed up the query processing. However, as elaborated in Sect. 1, our techniques are more memory-efficient, as well as more scalable than those in EAGR.

6 Conclusion

In this paper, we have proposed *Graph Window Query* to facilitate analytics over a local community of each graph vertex, and studied one instantiations, namely k-hop window in detail. We proposed the Dense Block Index (DBIndex) to facilitate efficient processing of k-hop window query. DBIndex integrates window aggregation sharing techniques to salvage partial work done, which is both space and query efficient. Results of an extensive experimental study on both large-scale real and synthetic datasets showed the efficiency and scalability of our proposed index.

Acknowledgment. Qi Fan is supported by NGS Scholarship. This work is supported by the MOE/NUS grant R-252-000-500-112 and AWS in Education Grant award.

References

1. Briscoe, E.J., Appling, D.S., Mappus IV, R.L., Hayes, H.: Determining credibility from social network structure. In: ICASNAM 2013, pp. 1418–1424. ACM (2013)
2. Broder, A.Z., Glassman, S.C., Manasse, M.S., Zweig, G.: Syntactic clustering of the web. Comput. Netw. ISDN Syst. **29**(8), 1157–1166 (1997)
3. Campanario, J.M.: Empirical study of journal impact factors obtained using the classical two-year citation window versus a five-year citation window. Scientometrics **87**(1), 189–204 (2011)
4. Chen, C., Yan, X., Zhu, F., Han, J., Yu, P.S.: Graph OLAP: towards online analytical processing on graphs. In: ICDM 2008, pp. 103–112 (2008)
5. Cheng, J., Shang, Z., Cheng, H., Wang, H., Yu, J.X.: K-reach: who is in your small world. VLDB **5**(11), 1292–1303 (2012)
6. Dai, L., Luo, J.-D., Fu, X., Li, Z.: Predicting offline behaviors from online features: an ego-centric dynamical network approach. In: HotSocial 2012, pp. 17–24 (2012)

7. Everett, M., Borgatti, S.P.: Ego network betweenness. Soc. Netw. **27**(1), 31–38 (2005)
8. Fan, Q., Wang, Z., Chan, C.Y., Tan, K.L.: Supporting window analytics over large-scale dynamic graphs, CORR (2015). arxiv:1510.07104
9. Huebsch, R., Garofalakis, M., Hellerstein, J.M., Stoica, I.: Sharing aggregate computation for distributed queries. In: SIGMOD 2007, pp. 485–496 (2007)
10. Ma, H.H., Gustafson, S., Moitra, A., Bracewell, D.: Ego-centric network sampling in viral marketing applications. In: Ting, I.-H., Wu, H.-J., Ho, T.-H. (eds.) Mining and Analyzing Social Networks. SCI, vol. 288, pp. 35–51. Springer, Heidelberg (2010)
11. Ma, N., Guan, J., Zhao, Y.: Bringing pagerank to the citation analysis. Inf. Process. Manage. **44**(2), 800–810 (2008)
12. Mondal, J., Deshpande, A.: Eagr: supporting continuous ego-centric aggregate queries over large dynamic graphs. In: SIGMOD 2014, pp. 1335–1346 (2014)
13. Moustafa, W.E., Deshpande, A., Getoor, L.: Ego-centric graph pattern census. In: ICDE 2012, pp. 234–245 (2012)
14. Navlakha, S., Rastogi, R., Shrivastava, N.: Graph summarization with bounded error. In: SIGMOD 2008, pp. 419–432 (2008)
15. Trigoni, N., Yao, Y., Demers, A., Gehrke, J., Rajaraman, R.: Multi-query optimization for sensor networks. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 307–321. Springer, Heidelberg (2005)
16. Vassilevska, V., Pinar, A.: Finding nonoverlapping dense blocks of a sparse matrix. Lawrence Berkeley National Laboratory, Livermore (2004)
17. Wang, Z., Fan, Q., Wang, H., Tan, K.-L., Agrawal, D., El Abbadi, A.: Pagrol: parallel graph OLAP over large-scale attributed graphs. In: ICDE 2014, pp. 496–507 (2014)
18. Yan, X., He, B., Zhu, F., Han, J.: Top-k aggregation queries over large networks. In: ICDE 2010, pp. 377–380 (2010)
19. Zhao, P., Li, X., Xin, D., Han, J.: Graph cube: on warehousing and OLAP multidimensional networks. In: SIGMOD 2011, pp. 853–864 (2011)