Towards Window Analytics over Large-scale Graphs

Qi Fan
NUS Graduate School for Integrative Sciences and Engineering
National University of Singapore
Singapore
fan.qi@nus.edu.sg

Kian-Lee Tan
Department of Computer Science
School of Computing, NUS
Singapore
tankl@comp.nus.edu.sg

ABSTRACT

In relational DBMS, window functions have been widely used to facilitate data analytics. Surprisingly, while similar concepts have been employed for graph analytics, there has been no explicit notions of graph window analytic functions. In this paper, we formally introduce window queries for graph analytics. In such queries, for each vertex, the analysis is performed on a window of vertices defined based on the graph structure. In particular, we identify three instantiations, namely the *unified window*, the *k-hop window* and the *topological window*. We focus on processing the latter two window queries and develop two novel indices, *Dense Block* index (*DBIndex*) and *Inheritance* index (*I-Index*), to facilitate efficient processing of these two types of windows respectively.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing; H.2, E.5 [**Database**]: Optimization

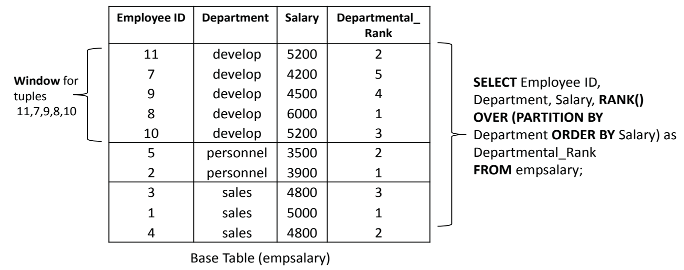
General Terms

Graph Database, Query Processing, Large Network

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. With increasingly larger network sizes, it is becoming significantly challenging to query, analyze and process these graph data. Therefore, there is an urgent need to develop effective and efficient mechanisms over graph data to draw out information from such data resources.

Traditionally, in relational DBMS, window functions have been commonly used for data analytics [3]. Instead of performing analysis (e.g. ranking, aggregate) over the entire data set, a window function returns for each input tuple a value derived from applying the function over a window of neighboring tuples. For instance, users may be interested in finding each employee's salary ranking within the department as shown in Figure 1. In such a window function, each tuple's neighbors are the tuples from the same department. Generally, a tuple's window is the set of tuples which are *related* to it. Thus, performing analytics over a tuple's window reflects its *personalized* evaluation.



Employee ID	Department	Salary	Departmental_ Rank
11	develop	5200	2
7	develop	4200	5
9	develop	4500	4
8	develop	6000	1
10	develop	5200	3
5	personnel	3500	2
2	personnel	3900	1
3	sales	4800	3
1	sales	5000	1
4	sales	4800	2

```
SELECT Employee ID,
Department, Salary, RANK()
OVER (PARTITION BY
Department ORDER BY Salary) as
Departmental_Rank
FROM empsalary;
```

Window for tuples 11,7,9,8,10
Base Table (empsalary)

Figure 1: Window function in RDBMS. The employee salary table is partitioned based on department and each employee's salary is ranked based on the window query on the right-hand side.

Interestingly, this notion of *window* turns out to be not uncommon in graph scenario. For instance, in a social network, it is important to detect a person's social position and influence among his/her social community. The "social community" of the person is essentially his/her *window* comprising neighbors derived from his/her friends. Surprisingly, though such a concept of window functions has been widely used, the notion has not been explicitly formulated. In this paper, we are motivated to bring window query from relational table to graph. As compared to relational scenario, the structure of graph plays an important role in determining *relatedness* of vertices. Thus, based on various metrics of *relatedness*, we identified three types of the windows , namely *unified*, k*-hop*, and *topological* windows. We first demonstrate these window semantics with the following examples:

EXAMPLE 1. (Unified window) *In a web graph, web pages are modeled as vertices and the hyperlinks are the edges. In order to find the importance of webpages, some measures (e.g. PageRank, centrality etc.) need to be performed. Usually, the entire web graph is partitioned based on the webpage category (e.g. news, advertising, and personal blogs etc.). In such scenarios, collecting vertices based on* attribute *is necessary.*

EXAMPLE 2. (K-hop window) *In a social network (e.g. LinkedIn and Facebook etc.), users are normally modeled as vertices and connectivity relationships are modeled as edges. In social network*

SIGMOD'15 PhD Symposium, May 31, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-3529-4/15/05 ...$15.00.
http://dx.doi.org/10.1145/2744680.2744689.

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.*

EXAMPLE 3. (Topological window) *In biological networks (such as Argocyc, Ecocyc etc.[5]), genes, enzymes and proteins are vertices and their dependency in a pathway are edges. Because these networks are directed and acyclic, in order to study the protein regulating process, one may be interested to find out the statistics of molecules in each protein production pathway. For each protein, we can traverse the graph to find every other molecule that is in the upstream of its pathway. Then we can group and count the number of genes and enzymes among those molecules.*

A common feature among these examples is that data aggregation *for each vertex* is based on *a set of related vertices* (which is the *graph window*). To illustrate, in E.g. 1, every webpage needs to gather other pages with the same category, which forms its window. Similarly, in E.g. 2, every user needs to gather data from its friends and friends-of-friends. The *2-hop neighbors* form its window. Likewise, in E.g. 3, every protein needs to count the number of particular type of genes preceding it in the regulating pathway. For every protein, the set of *preceding molecules* forms its window.

To support analyses in the above-mentioned examples, we propose a new type of query, *Graph Window Query* (GWQ in short), over a data graph. Unlike the relational window query, we identify three types of useful graph windows, namely Unified Window W_u, k-hop Window W_{kh} and Topological Window W_t. W_u forms a window for one vertex by looking at the vertices that of the same type. which represents the *attribute relatedness* as shown in E.g 1. W_{kh} forms a window for one vertex by using its k-hop neighbors, which demonstrates the *structure closeness* as shown in E.g. 2. W_t, on the other hand, forms a window for one vertex by using all its preceding vertices in a directed acyclic graph. The preceding vertices of one vertex are normally those which *structurally influence* the vertex in a network as illustrated in E.g 3.

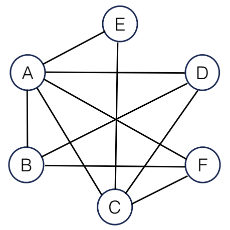
To the best of our knowledge, existing graph databases or graph query languages do not directly support our proposed *GWQ*. There are two major challenges in processing *GWQ*. First, we need efficient schemes to calculate the *window* of each vertex. Second, we need efficient solutions to process the aggregation over a large number of windows that may overlap. This offers opportunities to share the computation. However, it is non-trivial to address these two challenges. In this paper, we focus on efficiently processing *k-hop* and *topological* window as they show *structural relatedness* and are more challenging than *unified* window.

2. PROBLEM FORMULATION

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 node/edge is associated with a (possibly empty) set of attribute-value pairs.

Figure 2 shows an undirected graph representing a social network. The table shows the values of the five attributes (User, Age, Gender, Industry, and Number of posts) associated with each vertex. For convenience, each node 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.

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



(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)

Figure 2: Example Social Graph. (a) provides the graph structure; (b) provides the attributes associated with the vertices of (a).

$W(v)$ denotes a *window specification* for a vertex $v \in V$ that determines the set of vertices in some subgraph of G, Σ denotes an *aggregation 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 over $W(v)$, which we denote by $\Sigma_{v' \in W(v)} v'$.

Then, we formally define the following three useful types of window specifications(i.e. Ws):

DEFINITION 1 (UNIFIED WINDOW). *Given a vertex v in a data graph G, the unified window of v wrt a classifier θ, denoted by $W_u(v, \theta)$ (or $W_u(v)$ when there is no ambiguity) is the set of vertices whose class is the same as v's. More specifically, $W_u(v, \theta) = \{t | t \in G.V \cap \theta(t) = \theta(u)\}$*

Note that if there is no classifier exists, a vertex's window is every vertex in the graph. An example query of W_u could be *compute the centrality of each female in the female community*. Such a window can be expressed as $W_u(v, gender = F)$. As shown in Figure 2, $W_u(B, gender = F)$ is $\{B, C, F\}$. In fact, every member in the same community shares the same window, i.e. $W_u(C) = W_u(B) = W_u(F)$.

DEFINITION 2 (K-HOP WINDOW). *Given a vertex v in a data 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* [1] *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 with regard to their structural relationship in a graph. Thus, k-hop windows provide meaningful specifications for many applications, such as customer behavior analysis [1] etc.

As an example, in Figure 2, the 1-hop window of vertex E is $\{A, C, E\}$ and the 2-hop window of vertex E is $\{A, B, C, D, E, F\}$.

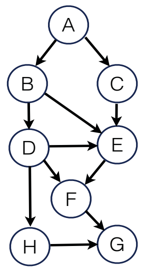
DEFINITION 3 (TOPOLOGICAL WINDOW). *Given a vertex v in a DAG G, the topological window of v, denoted by $W_t(v)$, refers to the set of ancestor vertices of v in G; i.e., a vertex u is in $W_t(v)$ iff there is directed path from u to v in G.*

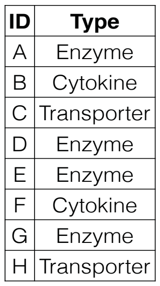
There are many directed acyclic graphs (DAGs) in real-world applications (such as biological networks, citation networks and dependency networks) where topological windows represent meaningful relationships that are of interest. For example, in a citation

[1]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 $\alpha \leqslant k$.

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 [2].

As an example, Figure 3 shows a small example of a Pathway Graph from a biological network. The topological window of E $W_t(E)$ is $\{A, B, C, D, E\}$ and $W_t(H)$ is $\{A, B, D, H\}$.





ID	Type
A	Enzyme
B	Cytokine
C	Transporter
D	Enzyme
E	Enzyme
F	Cytokine
G	Enzyme
H	Transporter

(a) (b)

Figure 3: Example Pathway DAG. (a) provides the DAG structure; (b) provides the attributes associated with the vertices of (a).

DEFINITION 4 (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.*

All three types of window could be uniformly presented in a SQL-like syntax as in Listing 1, where "compute" clause indicates aggregation functions and "over" clause indicates the window specification.

```
COMPUTE Σ1, Σ2...
ON G
OVER W(Wu | Wkh | Wt)
```

```
COMPUTE sum(Posts)
ON social-graph
OVER (2-hop)
```

Listing 1: GWQ Syntax

Listing 2: GWQ for E.g. 2

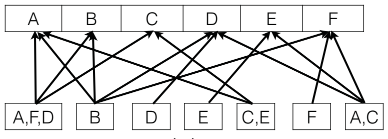
In this paper, we focus on graph window queries with a single window function that is either a k-hop or topological window. Furthermore, we focus on the attribute-based aggregation with distributive or algebraic aggregation functions at this stage. In other words, let $W(v)$ refer to a set of vertices, then aggregation function Σ operates on the values of attribute A over all the vertices in $W(v)$. Meanwhile, the aggregation function Σ is distributive or algebraic (e.g., sum, count, average), as these aggregation functions are widely used in practice. An sample query to answer E.g. 2 is shown as in Listing 2.

3. OUTLINE OF METHODOLOGY

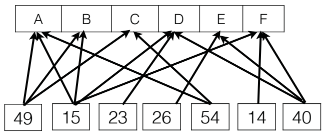
To boost window queries, we develop two indexed schemes for W_{kh} and W_t queries respectively.

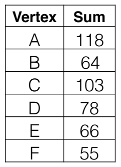
3.1 Dense Block Index for K-Hop Window

We first proposed an indexing scheme named *dense block index*(DBIndex), which is both space and query efficient. The main idea of DBIndex is to try to reduce the aggregation 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 Figure 2 using the 1-hop window function. 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



(a)





Vertex	Sum
A	118
B	64
C	103
D	78
E	66
F	55

(b) (c)

Figure 4: Window Query Processing using DBIndex. (a) provides the DBIndex for 1-hop window query in Figure 1; (b) shows the partial aggregate results based on the dense block; (c) provides the final aggregate value of each window.

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 the aggregation for $\Sigma_{v \in W(B)} v.A$ and $\Sigma_{v \in W(C)} v.A$.

Given a window function 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., there exists $v_1, v_2 \in V$, $v_1 \neq v_2$, $B \subseteq W(v_1)$, and $B \subseteq W(v_2)$), then B is referred to as 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 $W(X)$ is equal 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 $i \neq j$.

Thus, given a window function 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 nodes (i.e., V); the second component is a collection of blocks; i.e., $\mathcal{B} = \{B_1, \cdots, B_n\}$ where each $B_i \subseteq V$; and the third component is a collection of links from blocks to nodes such that if a set of blocks $B(v) \subseteq \mathcal{B}$ is linked to a node $v \in V$, then $W(v)$ is covered by $B(v)$. Note that a DBIndex is independent of both the aggregation function (i.e., Σ) and the attribute to be aggregated (i.e., A).

Figure 4(a) shows an example of a DBIndex wrt the social graph in Figure 2 and the 1-hop window function. Note that the index consists of a total of seven blocks of which three of them are dense blocks.

3.2 Query Processing using DBIndex

Processing k-hop window query with DBIndex consists of 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 $\Sigma_{B_i \in B(v)} T_i$.

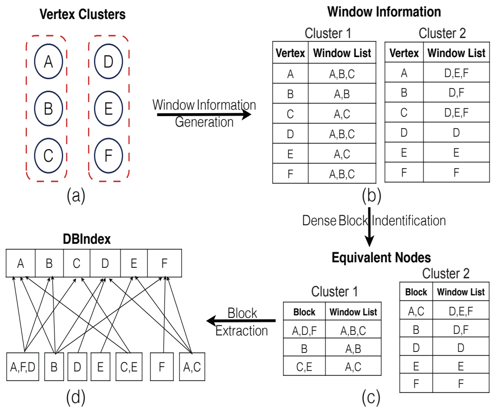
3.3 DBIndex Construction

Constructing DBIndex has two key challenges: First is the time complexity of the index construction. From above discussion of query processing, we note that the number of aggregations 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, 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 is NP-hard[2]. Therefore, efficient heuristics are needed to construct the DBIndex.

Second is the space complexity of the index construction. In order to identify large dense blocks, a straightforward approach is to first collect the window $W(v)$ for each node $v \in V$ and then use this 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 handle large graphs.

Our idea to reduce indexing complexity is by clustering vertices using their window values, and then within each cluster (which is much smaller than original graph) we discover the dense blocks among the member vertices. We refer this approach as *MinHash Clustering(MC)* heuristic, which works as follows: The vertices in graph is first clustered based on the MinHash signature of its windows (which can be collected by Breadth First Traversal). For each cluster, the nodes are grouped based on the window equivalence. Each grouped nodes is a *dense block*. Then, every nodes refine its window information based on the generated *dense block*s. An example of MC heuristic is shown as in Figure 5.

Based on the property of W_{kh}, we notice that, given a vertex$v$, its higher hop window always contains its lower hop window, i.e. *if* $i \leq j$, *then* $W_{ih}(v) \subseteq W_{jh}(v)$. Based on this observation, we proposed another index construction method *Estimated MinHash Clustering (EMC)*, which clusters vertices based on its lower hop window information. Since the lower hop window is easier to generate, the indexing time for *EMC* is much shorter than *MC* algorithm. On the other hand, since the estimation scheme imposes the performance penalty as lower hop window does fully reflect higher hop window, the query performance for *EMC* is affected. However, as shown in our experiments, the drop on performance is minimal.



Cluster 1

Vertex	Window List
A	A,B,C
B	A,B
C	A,C
D	A,B,C
E	A,C
F	A,B,C

Cluster 2

Vertex	Window List
A	D,E,F
B	D,F
C	D,E,F
D	D
E	E
F	F

Cluster 1

Block	Window List
A,D,F	A,B,C
B	A,B
C,E	A,C

Cluster 2

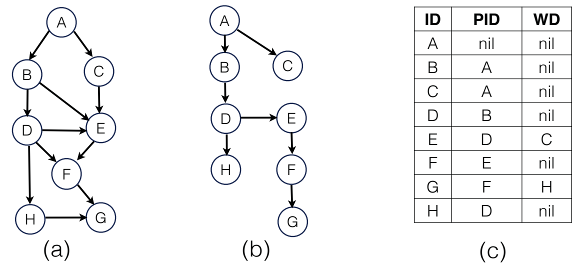
Block	Window List
A,C	D,E,F
B	D,F
D	D
E	E
F	F

Figure 5: DBIndex Construction over Social Graph in Figure 2. (a) shows two clusters after MinHash clustering; (b) shows the window information of involved vertices within each cluster; (c) shows the dense blocks within each cluster; (d) provides the final DBIndex.

3.4 Inheritance Index for Topological Window

DBIndex is a general index that can support both k-hop as well as topological window queries. However, the evaluation of a topological window function, W_t, can be further optimized due to its window containment property. In other words, the window of a descendant vertex completely covers that of its ancestors. That is, *In DAG, if vertex* u *is the ancestor of vertex* v*, then* $W_t(u) \subset W_t(v)$. For example, in Figure 6 we can see that the window of D, $W_t(D)$ is $\{A, B, D\}$ and the window of E, $W_t(E)$ is $\{A, B, D, C, E\}$. It is easy to see that $W_t(D) \subset W_t(E)$.

[2]Note that a simpler variation of our optimization problem has been proven to be NP-complete [6].



ID	PID	WD
A	nil	nil
B	A	nil
C	A	nil
D	B	nil
E	D	C
F	E	nil
G	F	H
H	D	nil

Figure 6: I-Index Construction over the Pathway DAG in Figure 3. (a) shows the DAG structure; (b) provides the inheritance relationship discovered during the index construction; (c) shows the final I-Index.

Based on the containment property, given a vertex u and its parent v, since $W_t(u) \subseteq W_t(v)$, there is no need to maintain the full set of vertices in $W_t(v)$. Instead, we only need to keep the difference between $W_t(v)$ and $W_t(u)$. For instance, in Figure 6 (a), instead of maintaining $\{A, B, D, C\}$ for $W_t(E)$, one can simply maintain the difference to $W_t(D)$ which is $\{C\}$. This is clearly more space efficient.

Thus, we propose a new structure, called the **Inheritance Index**, $I\text{-}Index$, to support efficient processing of topological window queries. In $I\text{-}Index$, each vertex v maintains two information: (1)*PID*: the ID of its *closest* parent and (2)*WD*: the *difference* in window information with its closest parent. Figure 6(c) shows the I-index of Figure 6(a), where I-Index is represented in a table format; the second column is the PID and the third is the WD.

Building an I-Index for a DAG can be done efficiently via a topological scan. During the scan, each vertex v find its parent with smallest window cardinality. Then, v pass the window information $W_t(v)$ to all its children.

With the I-Index, window aggregation can be processed efficiently in topological order. For vertex v, its window aggregation can be calculated as $\Sigma(W_t(v)) = \Sigma(W_t(v.PID), \Sigma(v.WD))$,where Σ is the aggregate function. As the vertex is processed according to the topological order, $W_t(v.PID)$ would have already been calculated before computation of v.

4. PRELIMINARY RESULTS

We have conducted a series of experimental evaluations and gained some preliminary results. We use 2 real information networks for *k-hop* query, which are available at the Stanford *SNAP* website [3]: Amazon and Stanford-web. The detail description of these datasets is provided in Table 1. We use the widely used DAGGER [8] to generate synthetic DAGs for *Topological* query.

Name	Type	# of Vertices	# of Edges
Amazon	undirected	334,863	925,872
Stanford-web	directed	281,903	2,312,497

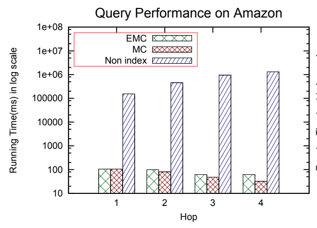
Table 1: Real Data Sets

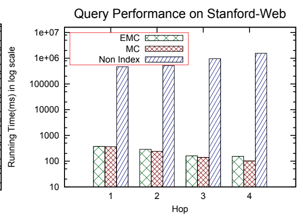
[3]http://snap.stanford.edu/snap/index.html

We use two non-indexed algorithms as baselines for W_{kh} and W_t respectively. The non-indexed algorithms presume no index exists and process the query from the raw graph. For W_{kh} query, the non-indexed algorithm is achieved by bounded breadth first search on each vertex; While for W_t query, the non-indexed algorithm is achieved by a topological scan.

4.1 Query Performance on W_{kh} Window

Figures 7 (a) and (b) present the query time of MC and EMC on the two datasets respectively as we vary the number of hops from 1 to 4. In the figures, 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 over the Amazon graph, 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 30ms to 300ms. 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 query time of EMC is slightly longer than that of MC when the number of hops is large. This is expected as EMC does not cluster based on the complete window information. However, the performance gap is quite small (20ms to 35ms) even for 4-hop queries.





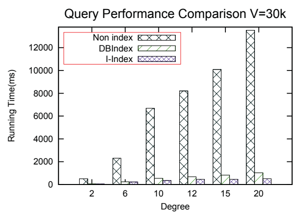
(a) Amazon Graph (b) Stanford-web Graph

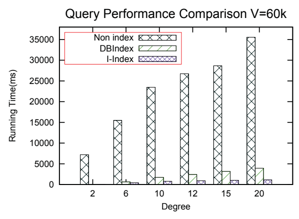
Figure 7: Query Performance Comparison of MC and EMC

4.2 Query Performance on W_t Window

Impact of Degree. First, we evaluate the impact of degree changes when we fix the number of vertex as 30k and 60k. We compare *DBIndex* with I-Index and a non-indexed algorithm. For query performance, as shown in Figures 8 (a) and (b), the non-index approach is, on average, 20 times slower than the index-based schemes. I-Index outperforms DBIndex by 20% to 30%. window.

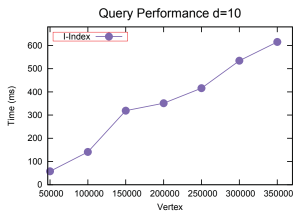
Impact of Number of Vertices. Next, we study how the performance of I-Index is affected when we fix the degree and vary the number of vertices. Figures 9 (a) and (d) show the query time when we fix the degree to 10 and 20 respectively. As shown, the degree affects the query processing time - when the degree increases, the query time increases as well. We also observe that the query time is increasing linearly when the number of vertices increases. This shows the I-Index has good scalability.

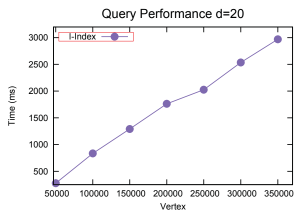




(a) Query Performance (b) Query Performance

Figure 8: Impact of Degree. (a) is for graph with 30K vertices; (b)is for graph with 60K vertices.





(a) Query Performance (b) Query Performance

Figure 9: Impact of the number of vertices with a fixed degree. (a) is for graphs with degree 10; (b) is for graphs with degree 20.

5. CONCLUSION AND FUTURE WORK

In this paper, we propose a new type of graph analytic query, *Graph Window Query*, with three instantiations: uniform window W_u, k-hop window W_{kh} and topological window W_t. We develop the Dense Block Index (DBIndex) to facilitate efficient processing for both types of graph windows. In addition, we propose the Inheritance Index (I-Index) that improves the performance of W_t queries. Both indices integrate window aggregation sharing techniques to salvage partial work done, which is both space and query efficient.

Our current work on window queries forms the basis of my dissertation. We then wish to further address several problems related to the window queries. First, we aim to extend window function processing to dynamic graphs, (i.e. graph may incur edge insertions and deletions). Second, we want to include structural aggregations (e.g. centrality, PageRank, and Graph Aggregation [7] etc.) which would enrich the semantic of graph window queries.

6. REFERENCES

[1] E. J. Briscoe, D. S. Appling, R. L. Mappus, IV, and H. Hayes. Determining credibility from social network structure. In *ASONAM'13*.

[2] J. M. Campanario. Empirical study of journal impact factors obtained using the classical two-year citation window versus a five-year citation window. In *Scientometrics'11*.

[3] Y. Cao, C.-Y. Chan, J. Li, and K.-L. Tan. Optimization of analytic window functions. In *VLDB'12*.

[4] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph olap: Towards online analytical processing on graphs. In *ICDM'08*.

[5] I. M. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I. T. Paulsen, M. Peralta-Gil, and P. D. Karp. Ecocyc: a comprehensive database resource for escherichia coli. In *Nucleic acids research'05*.

[6] V. Vassilevska and A. Pinar. Finding nonoverlapping dense blocks of a sparse matrix. In *Lawrence Berkeley National Laboratory'04*.

[7] Z. Wang, Q. Fan, H. Wang, K.-l. Tan, D. Agrawal, and A. El Abbadi. Pagrol: Parallel graph olap over large-scale attributed graphs. In *ICDE'14*.

[8] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. In *arXiv preprint arXiv:1301.0977*, 2013.