GAL: Topology-Aware Serialization for Graph Traversals

Zeynep Korkmaz
University of Waterloo
zkorkmaz@uwaterloo.ca

M. Tamer Özsu
University of Waterloo
tamer.ozsu@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
khuzaima.daudjee@uwaterloo.ca

ABSTRACT

Graph traversals are expensive and storage systems need to be optimized for them. Identifying the connectivity structure is important to estimate the likelihood of graph objects being accessed together. Developing a storage layout for graph objects in consideration of the connectivity structure is important to perform low latency graph operations. Irregular access patterns of the input graphs also contain valuable information that can be used for locality optimization. Although hard to capture, real-world graphs have well-connected community structures where the vertices share common neighbours. In this paper, we address the challenges in identifying the access likelihood and connectivity structure of graphs to increase the locality of accesses. We propose Graph-Aware Layout (GAL), a workload-agnostic approach that orders graphs vertices by exploiting the connectivity structure born out of graph traversals. GAL exploits graph structure, determines the vertices that are likely to be accessed together, and imposes a serialization layout. With respect to common locality metrics, GAL performs better than or is competitive with other approaches.

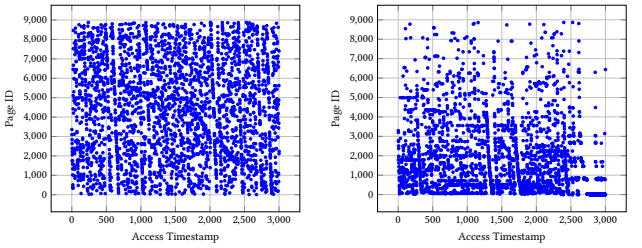
VLDB Workshop Reference Format:
Zeynep Korkmaz, M. Tamer Özsu, and Khuzaima Daudjee. GAL: Topology-Aware Serialization for Graph Traversals. VLDB 2025 Workshop: Proc. 4th International Workshop on Large-Scale Graph Data Analytics (LSGDA 2025).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zeynepsaka/graphs-GAL.

1 INTRODUCTION

Graphs are popular data structures to represent a variety of data where it is important to represent the relationships among data items as first-class objects [30]. Vertices in these graphs represent entities (e.g., people, processes, assets, and devices), and the relationships among these entities are well captured as edges. The volume of graph-structured data as well as the sizes of these graphs continue to grow, tracking the general data volume growth [15]. In response to this need, graph DBMSs (GDBMS) have been an active area of research and development – these are the fastest-growing NoSQL category [1] – and their performance and scalability have practical importance [29].

Proceedings of the VLDB Endowment. ISSN 2150-8097.



(a) The graph is serialized without structure awareness. (b) The graph is serialized with structure awareness.

Figure 1: Temporal data accesses of disk pages

Most graph processing applications need to perform expensive read-intensive traversals over large graphs. Locality in graphs captures the degree of correlated data accesses in time and space. Graph applications are known to have poor locality[1] [23]. This usually causes random access to disk pages during navigation, increasing the IO overhead. The culprit for random access is that GDBMSs typically serialize vertex and edge data on disk in the order that is given by the user in the input file without any attention to the graph topology. Figure 1a shows random accesses to disk pages when the input graph is serialized on disk without topology awareness using the storage layout of one of the popular GDBMSs [2]. The figure shows the scattered access causing high IO overhead.

Poor locality and inattention to graph topology also make in-memory caching algorithms ineffective. Access probabilities of graph objects can vary, and this yields non-uniform activity levels [4]. Random accesses tend not to have much locality [23], and they are hard to predict ahead of time to exploit caching. In previous work, we proposed a new caching algorithm, called LAC, for graph traversals and demonstrated that caching is greatly helped if the graph is serialized with attention to graph topology [20].

In this paper, we address the complementary problem of serialization and propose a workload-agnostic approach to serialize real-world graphs on disk by exploiting the connectivity structure born out of graph traversals.

We have two objectives: the first is to improve the ordering quality as evaluated by commonly used locality metrics, and the second is to minimize the number of page IOs in servicing user requests. Identifying the connectivity structure is important to estimate the likelihood of vertices being accessed together. Developing a storage layout of graph objects in consideration of this estimation is essential to perform low latency graph operations. Figure 1b shows a plot of the same graph data and query as in Figure 1a but with a changed data layout that correlates with the graph topology, e.g., adjacent vertices and neighbourhoods in the graph are stored on the same page or on adjacent pages on disk by using a topology-aware

[1]Poor locality in graph data is when vertices or edges which are frequently accessed together are not stored or laid out close to each other in memory or on disk.

serialization approach [35]. The resulting improved locality of data accesses generates more more clustered and consecutive disk page accesses.

A Small World Network graph consists of tightly-knit groups of vertices, closed triangles, and a number of local bridges [34]. As noted earlier, GDBMSs typically read vertices in ID-order from the input file, and those with consecutive IDs are stored consecutively. Usually, no attention is paid to the connectivity structure of the graph (if that structure is not represented in the order of items within the input file), which potentially leads to divergence of the physical data order on disk from the connectivity structure. We propose Graph-Aware Layout (GAL), which determines the vertices that would be frequently accessed together from the graph structure and generates a serialization order for graph objects on disk with the objective of increasing page access locality.

Some of the existing approaches that address similar problems force access locality in graph objects by performing community extraction or partitioning the graphs to preserve locality in subgraphs [16, 36]. In real graphs with power-law distribution of vertex degrees, edge-cut-based partitioning does not perform well [21]. A recent approach finds a permutation of graph vertices to minimize the CPU cache miss ratio [35]. It uses a cost function to maximize the weights that are assigned to the direct neighbour and sibling relations of vertices in the serialization order. Although identifying these direct relations gives some information about the access likelihood of the algorithms, it does not capture the locality in access patterns that are beyond immediate neighbourhoods.

GAL, by contrast, generates a serialization order of graph objects without relying on a partitioning algorithm; it examines the connections beyond the immediate neighbours and siblings by considering vertices that may be farther with respect to number of hops, but are in the same access sequence in graph traversals. For this purpose, GAL uses random walks to explore the connections farther than the immediate neighbours. These beyond-neighbourhood connections play an important role in the execution of reachability queries and have been shown to be important [14, 17, 21, 24, 27]. Small World Networks require both closely knit communities (strong ties) and a sufficient number of long-range neighbours (weak ties) [34]. Weak ties represent the local bridges where the vertices at the end points of such edges have no common immediate neighbours, such as acquaintances. Although there are many closed triangles in Small World Networks, branching structure appears as a result of local bridges that enable reaching many nodes in a few steps, and their endpoint vertices can access the parts of the graph that are otherwise hard to reach [7, 9].

Identifying tightly knit communities motivates many graph partitioning approaches, which define some level of cohesion for each vertex in a partition by assigning a fraction of its immediate neighbourhood in the same partition. However, a pair of vertices in a partition do not necessarily have much in common due to the existence of densely connected vertices as well as some number of weak edges [35]. Therefore, GAL stores hub vertices that are not immediate neighbours of each other, endpoints of local bridges, and their important neighbours that are likely to appear together in graph traversal. GAL may not serialize a vertex with all of its immediate neighbours on the same page. This potentially results in accessing more than one disk page to answer a single hop neighbour query. However, frequent accesses to disk pages consisting of the graph objects that appear in most of the traversals reduces the overall amount of IO.

In summary, our major contributions are the following:

- We propose GAL that generates a serialization order for graph objects by considering their structural roles in their placement and increasing the distance to look ahead, greater than that of the immediate successor in the access sequence of traversals.
- GAL captures the access likelihood of graph objects by looking at the graph topology; therefore, its disk layout is not tailored to a specific query type.
- GAL outperforms its competitors in various widely used locality metrics.

2 RELATED WORK

Efficient placement of graph objects to minimize IO is a challenging problem due to graph connectivity and unpredictable access patterns of graph workloads. Collocating vertices/edges together in a page is fundamentally a clustering problem. There are several existing studies that use graph partitioning methods and community extraction algorithms for clustering graph objects into pages. Xie et al. [36] propose a block-level abstraction of vertex-centric graph processing with the objective of achieving good cache performance. It uses a graph partitioner [19] to divide a graph into partitions in a manner that minimizes the number of cross-partition edges (i.e., edge-cuts) and each partition is stored as a block. However, power-law degree distribution of real graphs affects the partitioning quality [21, 23]. Steinhaus et al. [33] propose a storage manager for graph data that tries to minimize the number of adjacent vertices that are placed on different disk pages as well as the distance between disk pages with adjacent vertices. Another heuristic serializes graph data on disk by extracting communities based on modularity-optimized-driven algorithm [22]. However, this requires solving NP-hard minimum linear arrangement problem [16]. These approaches do not scale to large graphs.

Although community extraction and graph partitioning approaches explore groups of vertices that are likely to communicate, they cannot capture access likelihood of vertices in navigating through communities that follow a pattern unique to the graph. This is important for efficient placement of graph objects on pages in order to exploit the access locality in traversals and therefore, reduce the number of page accesses to answer a query. Vertex/edge reordering that is based on the statistical characteristics of the graph or cost functions that try to optimize accesses with the likelihood of vertices (similar to what we do) has also been studied [3, 6, 11, 35, 39]. Some of these target main memory placement, while others target on-disk placement. However, there is no intrinsic obstacle to using main memory placement techniques for on-disk placement; therefore, in the remainder, we assume they are used as on-disk approaches.

Reverse Cuthill-McKee (RCM) [8] graph ordering algorithm minimizes the maximum gap distance, which is known as graph bandwidth. RCM traverses the vertices at each level by following a specialized breadth-first search where vertices are ordered by their

degrees and vertex identifiers of their parents. Some of the ordering approaches rely on vertex degree [11] [39], placing higher degree vertices together on the same page. Degree-based Grouping (DBG) [11] has multiple bins for vertices and places them into the bins according to their "hotness level", which is defined based on their degrees. These techniques assume that the vertices have already assigned vertex identifiers consistent with the graph topology in the input file. This assumption holds for some input graphs, but not all. In contrast to these, GAL exploits the access likelihood of graph objects by considering the degree property of vertices. Gorder (GO) [35] is similar to GAL in that it reorders graph vertices prior to loading. It defines access locality of vertex pairs by a score function that considers their immediate neighbourhood and the number of common friends. Although identifying the immediate neighbourhood gives important information about the access likelihood of graph objects in the graph traversals, it does not explore the connections at further depths that are not easy to predict.

ICBL [38], considers neighbourhood at further depths. ICBL finds an ordering for graph data in order to serialize it on disk. It employs, similar to GAL, uniform random walks to extract diffusion sets and to explore the neighbourhood of a vertex in order to label their access likelihood. However, unlike GAL, it uses this information in k-means clustering to obtain k vertex-disjoint subgraphs, which may select local bridges as edge cuts that potentially carry most of the traffic between the different parts of the graph. In contrast to ICBL, GAL explores the global stability of the graph and the structural roles of graph objects, such as local bridges, in addition to identifying strong relations between the vertices and their access likelihood together.

3 GRAPH-AWARE LAYOUT

In this section, we describe GAL (Graph-Aware Layout), a topology-aware serialization technique that focuses on ordering graph vertices and projects this order on edges to obtain an edge ordering. GAL considers the roles vertices play in the graph structure, such as hubs or endpoints of local bridges. It goes beyond considering immediate neighbours and looks further for successor vertices in the access sequence of traversals to identify strong relations between the vertices and their likelihood of being accessed together. GAL orders hub vertices close to each other even if they are not immediate neighbours but are likely to be accessed together in graph traversal. The other ordering algorithms discussed in the previous section only consider vertices.

A graph $G = (V, E)$ consists of set of vertices V and edges $E = \{(u, v)|u, v \in V\}$. The number of edges linked to a given vertex v is its *degree*, $d(v)$. The reordering is based on the joint access likelihood of vertices, which is computed as follows. A set of random walks are started from each vertex v_i in the graph, during which virtual edges are created between the source vertex v_i and all the reachable vertices in the walk, and weights are assigned to each virtual edge based on their distance to the source vertex v_i – edges to vertices that are closer to the source (in terms of hop distance) have higher weights. The resulting weighted, denser graph $G_w = (V, E_w)$ captures the connection strength of vertices. G_w has edges that are not in G, representing the multi-hop connections between vertices that are captured through the traversals. While capturing the strength between important vertices that are not necessarily connected, it can sacrifice some of the immediate neighbours for not being ordered very closely.

Based on the edge weights in G_w and the number of common neighbours in G, GAL computes the access likelihood of all pairs of vertices (v_i, v_j) $(v_j \in V, v_i \neq v_j)$ using a closeness function that is explained in Section 3.2. Finally, GAL generates a new order of graph objects while maximizing the closeness of vertices.

Algorithm 1 shows the main steps of GAL. RandomWalk performs a number of random walks rooted at each vertex $v_i \in V(G)$ to generate G_w (Section 3.1. Then, VertexOrdering creates a new ordering of graph vertices by using the connection strength of vertices captured in G_w (Section 3.2).

Algorithm 1 GAL: Graph-Aware Layout Generation

Input: $V(G), E(G)$
Output: A new order of vertices in $V(G)$
1: **for all** $v_i \in V(G)$ **do**
2: RandomWalk(v_i) ▷ Generate and update G_w in every iteration
3: **end for**
4: VertexOrdering(G_w)

3.1 Random Walks

Exploring the structural distance between graph vertices is important to evaluate their similarity. GAL performs a number of random walks $R(v_i)$ rooted at each vertex $v_i \in V(G)$ to extract access sequences of vertices.

In the remainder, for ease of exposition, we denote each step of the random walk rooted at v_i as v_i^k, where k is the depth in the walk: $v_i^0 = v_i, v_i^1 = v_j, v_i^2 = v_k, \ldots, v_i^k = v_s$.

$$R(v_i) = \{v_i^0, v_i^1, v_i^2, \ldots, v_i^k\} \tag{1}$$

Each move of the random walk rooted at v_i is defined by the transition probability matrix P which has elements $p(v_i^t, v_i^{t+1}) = P[v_i^{t+1}|v_i^t]$ representing the probability of moving from a vertex at depth (step) t to a vertex at depth (step) $t + 1$ as shown in Equation 2.

$$P[v_i^{t+1}|v_i^t] = \begin{cases} p(v_i^t, v_i^{t+1}) & v_i^{t+1} \in N(v_i^t) \\ 0 & v_i^{t+1} \notin N(v_i^t) \end{cases} \tag{2}$$

The computation of $p(v_i^t, v_i^{t+1})$ depends on the type of random walk, which may be uniform or biased. In uniform biased walk, each vertex in $N(v_i^t)$ is equally likely to be chosen as the next visited vertex. The return time of vertex v_i, is defined as the expected time it takes a walk that starts at vertex v_i to return to vertex v_i. The return time of a vertex is independent of the structural properties of vertices that are visited at uniform random walk steps. However, the return time of a vertex is related to the degree property of vertices, $d(v_j)$, that are visited in the case of biased walks. When the walk reaches a high degree vertex, it decreases the return time of v_i which yields an exploration in the local neighbourhood. On the other hand when the walk reaches a low degree vertex, it increases the return time of v_i and, therefore, increases the probability of

traversing through the more sparse paths to a different part of the graph.

Walks involving higher degree vertices can obtain the connection between hub vertices that are not immediate neighbours but connected to each other through their successor neighbours and frequently accessed together in multi-hop traversals, while walks involving lower degree vertices can reach farther parts of the graph. GAL can use either uniform or biased random walks in capturing connection strength between vertices in different input graphs.

Algorithm 2 shows GAL's random walk procedure. It takes a root vertex v_i, the number of walks rooted at v_i, *NWALKS*, and the length of the walk, *NSTEPS*, as input and produces the weighted graph G_w as output.

Algorithm 2 RandomWalk

Input: v_i, *NWALKS*, *NSTEPS*
Output: G_w

$G_w = \{\}$
N is an array of vertices $\forall v_j \in N(v_i)$.
if biased **then**
  N is sorted based on $d(v_j)$.
  N_{cum} is a cumulative growing array of $d(v_j)$.
end if
for $n = 0$ to *NWALKS* **do**
  for $k = 0$ to *NSTEPS* **do**
    $p = random()\%d(v_i)$
    if biased **then**
      $v_i^k \leftarrow N_{cum}[p]$
    else
      $v_i^k \leftarrow N[p]$
    end if
    insert v_i^k in $R(v_i)^n$ ▷ $R(v_i)^n$ is the n^{th} random walk rooted at v_i
  end for
  BuildWeightedGraph$(v_i, R(v_i)^n, G_w)$
end for

The number of walks (*NWALKS*) rooted at the same vertex is important to increase the probability of capturing different neighbours at different walks. However, when the number of walks increases, the parameter sensitivity experiments show that the size of G_w increases dramatically without improving the ordering quality. Therefore, *NWALKS* is set to the average degree of the input graph.

Random walks can also be performed biased. N_{cum} is the cumulative distribution over neighbouring vertices degrees. This cumulative degree array is constructed such that $N_{cum}[k] = \sum_{j=1}^{k} d(v_j)$. Selecting from N_{cum} biases the sampling toward neighbors with higher degrees. This results in transition probabilities proportional to neighbor degrees, effectively biasing the walk toward high-degree nodes.

The path length of a random walk rooted at a vertex is also important for capturing the connection strength of vertices. GAL defines this distance as number of steps, *NSTEPS*, of a random walk. Assume *NSTEPS* is 3, and the access sequence of three vertices rooted at v_i are v_i^1, v_i^2, and v_i^3. In this case, v_i^1 is the immediate neighbour of v_i, while the others may not be. GAL determines the connection strength between v_i and all the vertices in this access sequence of vertices. Thus G_w will have the following weighted edges: (v_i, v_i^1), (v_i, v_i^2) and (v_i, v_i^3) in G_w. The default value for *NSTEPS* represents the reachability distance between vertices; therefore, we set it as the effective diameter value of the input graph.

Edge weights in G_w represent the likelihood that two vertices might be accessed together – higher weights indicate a closer relationship (in terms of graph topology) between the vertices incident on the edge. GAL employs a linear decremental weight assignment approach, in which an increase in the successor distance decreases the corresponding edge weight.

A random walk rooted at v_i has an initial weight assigned to the edge (v_i, v_i^1). Starting each random walk with the same initial weight could hide valuable information. Relations between a high-degree vertex and its neighbourhood can be more important to capture than relations between a lower-degree vertex and its neighbourhood. Thus, the initial weight is set as the degree of the rooted vertex on each random walk. For subsequent edges, the weight decreases with increasing distance to v_i as shown in Algorithm 3.

Algorithm 3 BuildWeightedGraph

Input: $v_i, R(v_i)^n, G_w$

for all $v_i^k \in R(v_i)^n$ **do**
  if weight assignment is skewed **then**
    $G_w[v_i][v_i^k] += d(v_i) - k \times \frac{d(v_i)}{NSTEPS}$
  else
    $G_w[v_i][v_i^k] += NSTEPS - k$
  end if
end for

The skewed weight assignment can be useful when ordering the graphs whose degree distribution exhibits significant skewness in order to capture high locality pages with hub graph objects. Otherwise, a linearly decreasing weight assignment can be used.

3.2 Vertex Ordering

GAL uses G_w to compute the "closeness" of vertices in $V \in G$ and obtain an ordered vertex set V_o. It does this by considering the vertices that are already in V_o focusing on the more recent p insertions where p is the size of a window it maintains over V_o – this is denoted as V_o^p. The next vertex in $V \setminus V_o$ to be added to V_o is the one that is closest to those currently in V_o^p.

Closeness of a vertex v_i to the vertices in V_o^p ($CL(v_i, V_o^p)$) is based on the edge weights and is computed by Equation 3 and has two components: edge weight score *es* and common neighbour score *ns*.

$$CL(v_i, V_o^p) = es(v_i, V_o^p) \times ns(v_i, V_o^p) \tag{3}$$

Edge weight score $es(v_i, V_o^p)$ is the sum of the edge weights between v_i and the vertices in V_o^p, $ew(v_i, v_j)$:

$$es(v_i, V_o^p) = \sum_{v_j \in V_o^p} ew(v_i, v_j) \tag{4}$$

The common neighbour score *ns* is the number of neighbour vertices of v_i in V_o^p:

$$ns(v_i, V_o^p) = |N(v_i) \cap V_o^p|. \tag{5}$$

GAL starts by computing the weight of each vertex as the sum of the weights of its incident edges in G_w (Algorithm 4) . That is the first vertex inserted into V_o – the first vertex in the order. This is followed by finding the "closest" vertices to those already ordered using the method described above. Naturally, it is not desirable to compute $CL(v_i, V_o^p)$ on the entire graph; therefore, a candidate list is generated for the vertices that should be considered for the next ordering. The candidate list consists of the neighbour vertices of those in V_o^p that are not already in V_o: $V_c^p = \bigcup N(V_o^p) \setminus V_o$.

Algorithm 4 VertexOrdering

Input: G_w, p
Output: V_o

```
1:  V_o = {}
2:  V_o^p = {}
3:  v_i ← the vertex with the highest sum of the weights of its
    incident edges in G_w.
4:  V_o ← V_o ∪ {v_i}                          ▷ insert v_i in V_o
5:  V_o^p = V_o^p ∪ {v_i}
6:  V ← V \ {v_i}                              ▷ remove v_i from V
7:  while V is not empty do
8:      V_c^p ← ⋃ N(V_o^p) \ V_o
9:      if V_c^p is not empty then
10:         for all v_n ∈ V_c^p do
11:             Calculate CL(v_n, V_o^p)
12:         end for
13:         v_next ← v_n with the maximum CL(v_n, V_o^p)
14:         V_o ← V_next ∪ {v_i}               ▷ insert v_next in V_o
15:         V ← V \ {v_next}                   ▷ remove v_next from V
16:         if |V_o^p| = p then
17:             V_o^p ← V_o^p \ V_o^p[0]
18:         end if
19:         V_o^p ← V_o^p ∪ {v_next}           ▷ insert v_next in V_o^p
20:     end if
21: end while
```

4 EXPERIMENTAL EVALUATION

This section describes the test environment, datasets and evaluation metrics used in the experiments.

4.1 Setup

The graph datasets are serialized into disk pages using Compact Sparse Row (CSR) format. CSR Offsets array has $|V|$ entries and each entry determines the alignment of the corresponding vertex's neighbourhood in the Edges array which is split into edge pages. Vertices are stored in the Vertices array which is split into vertex pages. When a query starts with a source vertex, its neighbouring vertices are found by following the offsets in the Edges array, and depending on the application, destination vertices' properties are visited in the Vertices array. We follow the data layout in [26] where each vertex page can store 512 vertices and each edge page can store 1024 edges.

4.2 Datasets

Our datasets are representative of three graph application domains: social networks, web graphs and road networks. SOC-LJ[2] and FLICKR[3] are mid-size online social networks where vertices and edges represent the friends and friendship relations, respectively. Online social networks are dense graphs with small diameters, meaning that vertices are reachable within a small number of hops. CIT-PATENT[4] and AMAZON[5] are web graphs whose vertices and edges represent web pages and hyperlinks between them, respectively. The average degree of web graphs is slightly smaller than online social networks, but their diameters are larger. Finally, US-ROAD[6] is a road network where road intersections and endpoints are represented by vertices and the roads connecting them are represented by undirected edges. Road networks are grid-type graphs where the average degree is very small, however the diameter is large.

4.3 Ordering Algorithms

We compare **GAL** with uniform random walk and **GAL-BW** with biased random walk against some of the ordering techniques given in Section 2: **RCM**, **DBG** and **GO**.

We also test the performance for the default layout (**DEF**) of the input graphs. Default layout refers to the order that vertices appear in the original input file. The ordering quality of the default layout is not always well-defined in the data source [32]. The default layouts of the input graphs have highly variable performance in practice, because some of the input files have already been ordered considering the graph topology, although the ordering techniques have not been mentioned. To show the impact of ordering, we shuffle the order of vertices in the input graphs and present its evaluation as **SHUFF** layout.

4.4 Evaluation Metrics

We evaluate GAL's ordering quality by widely-used and well-known locality metrics: conductance, cohesiveness, neighbourhood overlap, and gap distance. Graph objects in a cluster should display strong similarity among themselves. We use the terms "clusters" and "pages" interchangeably in the rest of the section.

The **conductance** metric is widely-used to evaluate the clustering quality and measures how well a page (P_i) is separated from the rest of the graph [5, 12, 18, 31]. The conductance of a page $\phi(P_i)$ measures the ratio of inter-page edges to all the edges in P_i: lower conductance, which represents fewer number of inter-cluster edges, is better.

A well-identified community's members are expected to be closely connected to each other, so that it should be hard to split it into disconnected components. This internal structure of a page can be termed as the **cohesiveness** ($\psi(P_i)$) and characterized by its density, which is the fraction of the edges (out of all possible edges) that appear between a pair of vertices in a page [18, 37].

[2] http://konect.cc/networks/soc-LiveJournal1/
[3] http://konect.cc/networks/flickr-growth/
[4] http://konect.cc/networks/patentcite/
[5] https://snap.stanford.edu/data/amazon0601.html
[6] https://networkrepository.com/road-road-usa.php

To evaluate the contribution of each page in the ordering quality of the graph, the *average* conductance over all the cuts of the graph and the average cohesiveness of all k pages in the graph are used: $\phi(G) = avg(\phi(P_i)), 1 \leq i \leq k$ and $\psi(G) = avg(\psi(P_i)), 1 \leq i \leq k$.

Another metric that has been used in the literature is **neighbourhood overlap** that measures goodness of clustering based on the strength of the relationship between vertices in a cluster or page. Granovetter [13] first proposed the strength to be measured by the relative overlap of the neighbourhoods of two vertices in the graph. Two vertices are similar when they have many common neighbours. Based on this, neighbourhood overlap of vertices u and v ($no(u, v)$) is defined as the ratio of their common neighbours to the total number of neighbours [25]. The edge (u, v) is identified as strong if $no(u, v)$ is high. Based on this, the neighbourhood overlap of a page $no(P_i)$ is defined as the average of all the $no(u, v)$ for all pairs of vertices $u, v \in P_i$. A page with a higher $no(P_i)$ is a better cluster since that means vertices in P_i have more common neighbours. Finally, the neighbourhood overlap value of a graph, $no(G)$ is defined as the average of $no(C_i)$ for all identified clusters in G: $no(G) = avg(no(P_i)), 1 \leq i \leq k$

The last metric we introduce is **gap distance** [10, 28], that represents the difference in vertex identifier values of vertices that are immediate neighbours in the graph. Given an ordering over $V(G)$, the gap distance is defined on each edge; given two neighbour vertices u and v, $gap(u, v)$ is the absolute difference of their identifiers. When graph serialization is performed with the objective of minimizing the gap distance, vertices with smaller gap distance are considered to better maintain the locality of vertices. Gap distance of a graph is defined as the average of $gap(u, v)$ of all edges (u, v) in graph G.

$gap(u, v)$ is defined on the incident edge of two neighbouring vertices u and v, and is the absolute difference of their identifiers (recall that ordering schemes change the vertex identifiers, so these are semantically significant). Since it is defined on the existing edges in the pages, and it may hide the information about the pages where vertices without neighbouring relation are collocated. Since we are evaluating the collocation quality of vertices in the pages, we slightly modify gap distance and introduce **page gap distance** between vertices, $gap_p(u, v)$ to better represent the impact of distance between vertices on the performance: $gap_p(u, v) = |i - j|, u \in C_i, v \in C_j, 1 \leq i \leq j \leq k$.

Then page gap distance of a graph, $gap_p(G)$ is the average page gap distance of all edges (u, v) in graph G:

$$gap_p(G) = \frac{1}{|E(G)|} \sum_{(u,v) \in E(G)} gap_p(u, v) \tag{6}$$

$gap_p(G)$ is not only promoted by the well-clustered pages as $gap(G)$ is, but also demoted by the pages which consist of vertices whose neighbours are scattered throughout the other pages.

4.5 Ordering Quality Evaluation Results

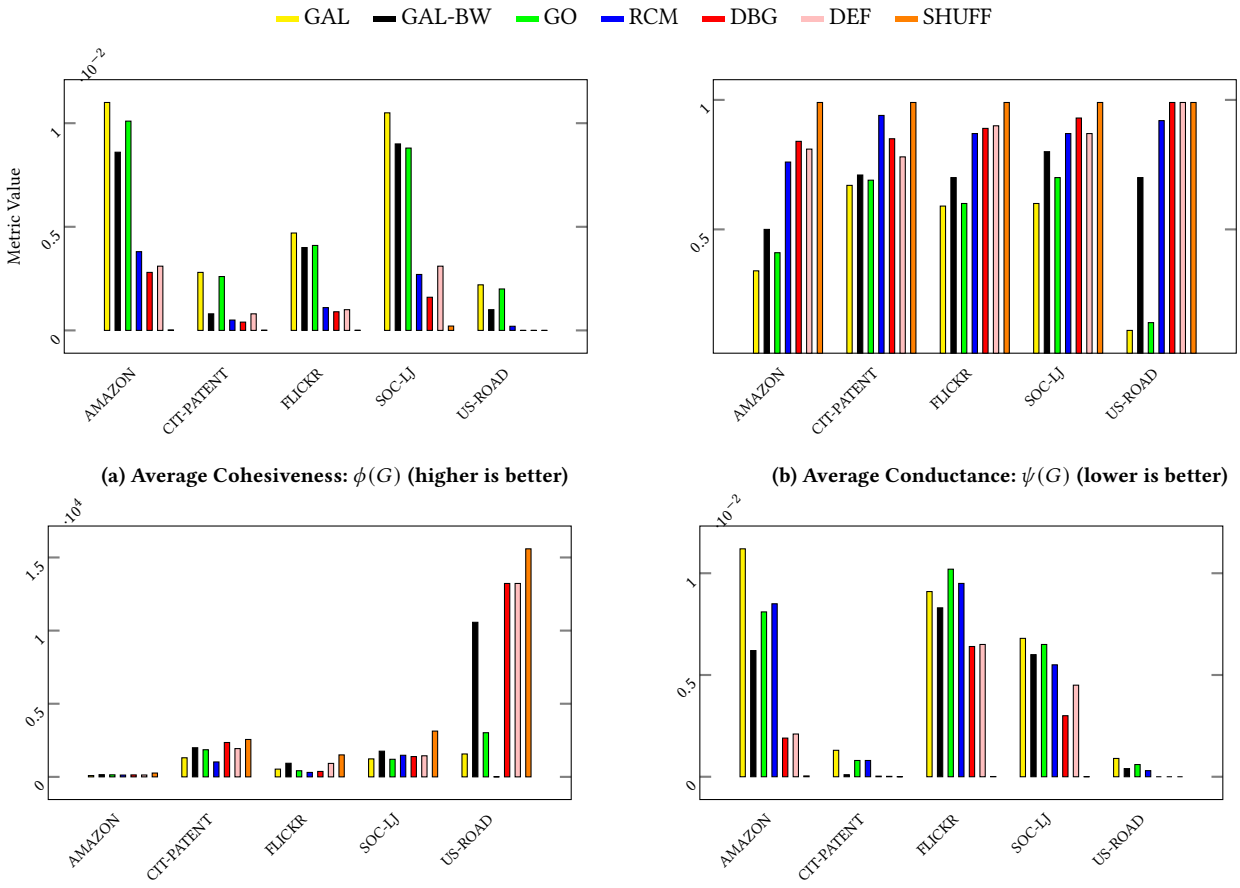
This section presents the results for ordering quality improvement achieved by different layout generation approaches. Figure 2 shows the ordering quality evaluation metrics of **cohesiveness** ($\phi(G)$), **conductance** ($\psi(G)$), **page gap distance** ($gap_p(G)$) and **neighbourhood overlap** ($no(G)$) results for the input graphs serialized by different algorithms.

The results show that ordering input graphs has an important positive impact relative to their default and shuffed layouts. As noted earlier the ordering quality of the default layout is not always well-defined in the data source [32]. The input files may already have some order, so the resulting default layout may reflect that; hence the default layout performance can vary highly.

A good cluster is usually identified in the literature as having low conductance and high cohesiveness. When calculating conductance, we exclude pages that contain vertices that have no incoming or no outgoing edges. Since lower conductance indicates better performance, this exclusion prevents such pages from artificially lowering the average conductance and distorting the results. Figures 2a and 2b show that GAL significantly reduces the inter-cluster dependency in all of the input graphs and has the highest cohesiveness and therefore increases the intra-cluster connectivity in all of the input graphs. GO has the second-best ordering quality in terms of these metrics, and its performance is close to GAL, especially in conductance when the input graph is very skewed. GAL-BW collocates the high-degree vertices that are a few hops of each other and tries to increase the ordering quality of these pages. However, it sacrifices low-degree vertices and their neighbourhoods more than GAL, and this situation reduces the overall quality in averages of the metrics. This property of GAL-BW is useful when graph serialization layout is targeted for processing hub graph objects. As DBG preserves the default layout while grouping vertices based on their degrees, its performance follows the default layout.

Smaller gap distance represents assigning closer vertex identifiers to the neighbouring vertices. RCM is explicitly designed to minimize the maximum distance between non-zero entries in rows (bandwidth) of a sparse matrix. This directly helps to decrease the gap distance that measures the average (page) distance between neighbours in a given vertex ordering. This leads to improved data locality, which is particularly beneficial in sparse graphs where most vertices have few neighbours, such as US-ROAD. In such cases, Figure 2c shows that RCM significantly reduces the distance between connected vertices and improves performance for gap distance metrics, although it performs less effectively for other evaluation criteria. In contrast, GAL demonstrates strong and consistent performance across different graph types. For example, GAL outperforms all other techniques in the AMAZON graph and matches RCM's performance in CIT-PATENT. GAL also outperforms GO in the US-ROAD network by considering deeper levels of neighbourhood structure, which is advantageous in graphs with high diameters like US-ROAD. GAL-BW (bandwidth-focused variant of GAL) performs closer to the default layout in some cases due to its biased random walks revisiting vertices and missing low-degree nodes. However, GAL overall shows that it is more adaptive. Its ability to leverage structural depth makes it a robust choice for improving vertex ordering across a range of graph topologies.

Higher neighbourhood overlap represents better collocation of vertices with their neighbourhood within a page. Figure 2d shows that GAL outperforms others in neighbourhood overlap for all input graphs except FLICKR. Although GAL can perform better in SOC-LJ and it has a very competitive performance, GO has the



(a) Average Cohesiveness: $\phi(G)$ (higher is better)

(b) Average Conductance: $\psi(G)$ (lower is better)

(c) Average Page Gap Distance: $gap_p(G)$ (lower is better)

(d) Average Neighbourhood Overlap: $no(G)$ (higher is better)

Figure 2: Comparison of ordering algorithms according to locality metrics.

best performance in neighbourhood overlap when the input graph has many closed triangles, such as FLICKR. This is expected as GO's objective function tries to maximize the number of common neighbours that form the closed triangles. When GAL sacrificing the lower degree vertices, it decreases the number of common neighbours of low degree vertices in a page. In web graphs and road networks, the average degree is lower, therefore, this situation rarely happens.

5 CONCLUSION

In this work, we study the irregular access pattern of graphs that comes from the structural properties. We propose a serialization algorithm, GAL, that considers the graph topology and the roles vertices play in this topology. GAL uses this information in the placement decisions, looking beyond the immediate successor of a vertex in the access sequence. We evaluate GAL with respect to existing ordering algorithms using well-known clustering quality/ordering locality metrics such as conductance, coheseiveness, neighbourhood overlap and gap distance. The results show that GAL outperforms the existing algorithms in most cases while demonstrating that overall GAL is more adaptive than its competitors. GAL's ability to leverage structural depth makes it a robust choice for improving vertex ordering across a range of graph topologies.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grants RGPIN-2024-03993 and RGPIN-2024-04657. We thank Lori Paniak for supporting the computing infrastructure used for this work.

REFERENCES

[1] [n.d.]. DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. https://db-engines.com/en/
[2] [n.d.]. Graph Data Platform | Graph Database Management System | Neo4j. https://neo4j.com/
[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *Proc. 30th IEEE Int. Parallel & Distributed Processing Symp.* 22–31. https://doi.org/10.1109/IPDPS.2016.110
[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proc. ACM SIGMOD Int. Conf. on Management of Data.* 1185–1196. https://doi.org/10.1145/2463676.2465296
[5] Yuchen Bian, Jingchao Ni, Wei Cheng, and Xiang Zhang. 2019. The multi-walker chain and its application in local community detection. *Knowl. and Information Syst.* 60 (2019), 1663–1691. https://doi.org/10.1007/s10115-018-1247-1

[6] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. 2022. Graph reordering for cache-efficient near neighbor search. In *Proc. 36th Int. Conf. on Neural Information Processing Systems* (New Orleans, LA, USA). Curran Associates Inc., Article 2789, 13 pages.

[7] Peter. Csermely. 2009. *Weak links : the universal key to the stability of networks and complex systems.* Springer.

[8] E. Cuthill and J. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th National Conference*. 157–172. https://doi.org/10.1145/800195.805928

[9] David Easley and Jon Kleinberg. 2010. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World.* Cambridge University Press.

[10] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Locality analysis of graph reordering algorithms. *Proc. Int. Symp. on Workload Characterization* (2021), 101–112. https://doi.org/10.1109/IISWC53511.2021.00020

[11] Priyank Faldu, Jeff DIamond, and Boris Grot. 2020. A closer look at lightweight graph reordering. *Proc. Int. Symp. on on Workload Characterization* (2020), 1–13. https://doi.org/10.48550/arxiv.2001.08448

[12] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* (2010), 75–174. https://doi.org/10.1016/j.physrep.2009.11.002

[13] Mark S. Granovetter. 1973. The Strength of Weak Ties. *Amer. J. Sociology* 78, 6 (1973), 1360–1380. http://www.jstor.org/stable/2776392

[14] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. 855–864. https://doi.org/10.1145/2939672.2939754

[15] Nicolaus Henke, Jacques Bughin, Michael Chui, J. Manyika, Tamim Saleh, and Bill Wiseman. 2016. The Age of Analytics: Competing in a data-driven world. https://api.semanticscholar.org/CorpusID:196173558

[16] Imranul Hoque and Indranil Gupta. 2012. Disk layout techniques for online social network data. *IEEE Internet Comput.* 16, 3 (2012), 24–36. https://doi.org/10.1109/MIC.2012.40

[17] Jinhong Jung. 2016. Random walk with restart on large graphs using block elimination. *ACM Trans. Database Syst.* 41, 2 (2016). https://doi.org/10.1145/2901736

[18] Ravi Kannan, Santosh Vempala, and Adrian Vetta. 2004. On clusterings: Good, bad and spectral. *J. ACM* 51, 3 (2004), 497–515. https://doi.org/10.1145/990308.990313

[19] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[20] Zeynep Korkmaz, M. Tamer Özsu, and Khuzaima Daudjee. 2025. Locality-Aware Cache Replacement Policy for Graph Traversals (to appear). *Proc. VLDB Endowment* 18 (2025).

[21] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Statistical properties of community structure in large social and information networks. In *Proc. 17th Int. World Wide Web Conf.* Association for Computing Machinery, 695–704. https://doi.org/10.1145/1367497.1367591

[22] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. [n.d.]. C-Miner: Mining block correlations in storage systems. In *Proc. 3rd USENIX Conf. on File and Storage Technologies*. USENIX Association, 13.

[23] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17 (2007), 5–20. https://doi.org/10.1142/S0129626407002843

[24] Lovász László, L. Lov, and Of Erdos. 1996. Random Walks on Graphs: A Survey. *Combinatorica*, 1–46.

[25] J. P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, D. Lazer, K. Kaski, J. Kertész, and A. L. Barabási. 2007. Structure and tie strengths in mobile communication networks. *Proc. National Academy of Sciences of the United States of America* 104, 18 (2007), 7332–7336. https://doi.org/10.1073/pnas.0610245104

[26] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proc. ACM Manag. Data* 2, 3, Article 125 (2024). https://doi.org/10.1145/3654928

[27] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *Proc. 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 701–710. https://doi.org/10.1145/2623330.2623732

[28] Ilya Safro, Dorit Ron, and Achi Brandt. 2006. Graph minimum linear arrangement by multilevel weighted wdge contractions. *Journal of Algorithms* 60, 1 (2006), 24–41. https://doi.org/10.1016/J.JALGOR.2004.10.004

[29] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endowment* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[30] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The Future is Big Graphs: A Community View on Graph Processing Systems. *Commun. ACM* 64, 9 (2021), 62—71. https://doi.org/10.1145/3434642

[31] Satu Elisa Schaeffer. 2007. Survey: Graph clustering. *Comput. Sci. Rev.* 1, 1 (2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[32] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2017. Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations. *Proc. 31st IEEE Int. Parallel & Distributed Processing Symp.* (2017), 588–597. https://doi.org/10.1109/IPDPSW.2017.164

[33] Robin Steinhaus, Dan Olteanu, and Tim Furche. 2010. G-Store : A storage manager for graph data. https://api.semanticscholar.org/CorpusID:13046587

[34] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442. https://doi.org/10.1038/30918

[35] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. ACM Press, 1813–1828. https://doi.org/10.1145/2882903.2915220

[36] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. 2013. Fast iterative graph computation with block updates. *Proc. VLDB Endowment* 6, 14 (2013), 2014–2025. https://doi.org/10.14778/2556549.2556581

[37] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2012), 181–213. https://doi.org/10.48550/arxiv.1205.6233

[38] Abdurrahman Yaşar, Buğra Gedik, and Hakan Ferhatosmanoğlu. 2017. Distributed block formation and layout for disk-based management of large-scale graphs. *Distributed and Parallel Databases* 35, 1 (2017), 23–53. https://doi.org/10.1007/s10619-017-7191-3

[39] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *Proc. 2017 IEEE Int. Conf. on Big Data*. 293–302. https://doi.org/10.1109/BigData.2017.8257937