Towards Faster External-Memory Graph Computing on Small Neighborhoods

Di Xiao, Yi Cui, and Dmitri Loguinov
Texas A&M University, College Station, TX 77843 USA
{di, yicui}@cse.tamu.edu, dmitri@cse.tamu.edu

***Abstract*—Knowledge mining in graphs has been widely adopted in such fields as information retrieval, social networks, databases, recommendation systems, and data analytics, where computation often involves evaluating a function applied to node/edge weights within a certain distance of each node. While traditional graph problems (e.g., PageRank) deal only with immediate neighbors, a variety of newer applications (e.g., triangle listing) examine all *wedges*, i.e., 2-node paths, originating from each source. We generalize these scenarios using a simple model, create a taxonomy of wedge-related computation based on the type of decomposition they allow, and propose efficient external-memory solutions to a particular class of such problems. We model the I/O cost of our framework and demonstrate that it achieves much better asymptotic complexity than prior methods. For performance evaluation, we focus on two sample applications – counting the number of unique supporters at distance 2 from each source and identifying all 4-cycles in a given graph. Results show that our algorithms can execute 2-3 orders of magnitude faster, and are similarly more efficient in terms of I/O, than the alternatives from related work.**

I. Introduction

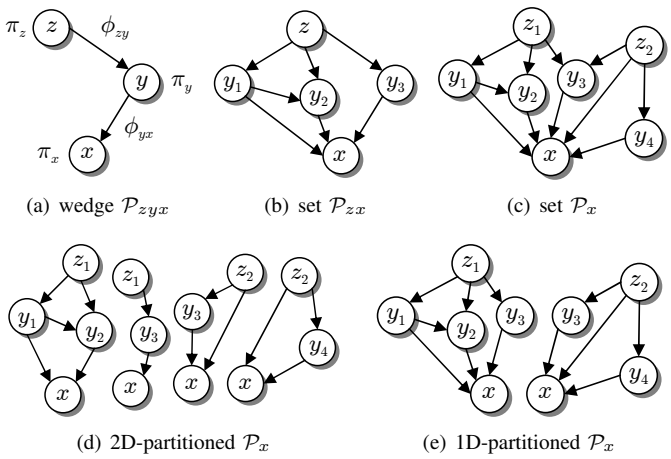
With a rapidly increasing capability of mankind to produce and consume data, computation on massive datasets has become an indispensable element of the current technological landscape of our society. In particular, many applications perform analysis, data mining, and information extraction on huge graphs that may be orders of magnitude larger than RAM. This can mean a 2-TB graph processed on a machine with 16 GB of RAM, or a 2-PB graph split across a 1000-node cluster. Algorithms for such problems must take into account not only the CPU complexity, but also the cost of I/O. While small-scale tasks can run by randomly accessing the disk to retrieve the necessary edges, this quickly becomes prohibitively expensive, even with SSDs, as the number of operations increases into the trillions. As a result, massive I/O-bound jobs often have to use sequential disk access and restrict calculations during each pass to small neighborhoods of the currently visible nodes.

Such local computation has shown feasibility in two main scenarios. The first one, which we call *neighbor-based*, collects information from immediate neighbors of each node along the directly attached links. Prime examples include PageRank [20], [39],[62], which iteratively passes weights among neighbors to determine the stationary distribution of a random walker, and construction of a search index [17], which inverts a weighted graph containing relationships between crawled pages and their keywords. After decades of research, first-neighborhood problems are well-understood, both algorithmically and theoretically, and efficiently covered by such existing frameworks as GraphChi [47], GraphLab [54], Graphene [52], GridGraph [79], Mosaic [55], and MapReduce [30].

The second scenario walks to distance two from each node, examining the various *wedges* (i.e., two-node paths) in the second neighborhood of the source. This category, which we label *wedge-based*, includes triangle enumeration [23], [28], [29], [41], [63], similarity ranking [15], [33], [34], [42], [45],[73], four-node motif discovery [6], [57], [69], spam avoidance in web crawling [13], [27], SpGEMM sparse matrix-matrix multiplication [18], [61], [74], [77], and various related problems [16], [64]. In general, these algorithms are quite expensive due to the enormous number of wedges that participate in the computation and their tendency to scatter across multiple partitions on disk.

The goal of this paper is to combine seemingly disjoint problems in wedge-computing under the umbrella of a unifying external-memory framework, which makes it easier to analyze the various applications, obtain deeper generalizations and insight, and translate new algorithms/theory into the context of multiple research communities. To this end, we first model wedge-based problems using a simple abstraction and show that the underlying algorithms fall into one of three classes, depending on the type of decomposition they admit. In particular, Category-I problems allow *two-dimensional* partitioning of each wedge, which is a problem that already has efficient solutions in triangle-enumeration literature [29], [63]. Assuming p is the number of partitions, m is the number of edges, and T_n is the number of wedges in a graph with n nodes, Category-I problems require I/O no larger than $\min(T_n, \sqrt{p}m)$ [29].

Category-II problems permit only *one-dimensional* partitioning of the wedge, which includes the majority of the problems listed above (i.e., supporter graph ranking, 4-node cycles, similarity calculation, recommendation systems, matrix multiplication). While general graph libraries [47], [52], [54], [55], [79] and MapReduce [30] can be adopted to deal with such problems, they are vastly expensive and often require sorting all wedges on disk. Even on relatively small graphs, this translates into TBs and even PBs of I/O. Instead, we propose a novel framework called *Decomposable Wedge Computing* (DWC) that solves these types of problems using $\min(T_n, pm)$ I/O complexity. We derive a precise closed-



(a) wedge $\mathcal{P}_{zyx}$ (b) set $\mathcal{P}_{zx}$ (c) set $\mathcal{P}_x$

(d) 2D-partitioned $\mathcal{P}_x$ (e) 1D-partitioned $\mathcal{P}_x$

Fig. 1. Definitions.

form I/O model for our method and show that it has *linear* scaling when the expected product of in/out-degree at each node remains $O(1)$ as the number of nodes $n \to \infty$. Finally, problems in Category III (e.g., enumeration of 4-node cliques) appear to be strictly more difficult and are left open for future work.

We test our algorithms on several real graphs, which range from 400M to 238B edges. Using RAM that is 10-100$\times$ smaller than the graph, we obtain runtimes and total I/O that are quite competitive against the alternatives. For example, our implementation can process the largest graph in the dataset (i.e., 1 TB with $T_n = 80$T wedges) using 8 GB of memory in just 31 hours. Estimates show that performing the same job using the fastest alternative from prior work (i.e., GraphChi) requires almost 9 months.

II. Definitions and Objectives

There is a large body of work in big-data graph processing; however, most of it is orthogonal to our investigation here. The majority of prior literature assumes either that the entire graph fits in memory [6], [22], [56] or that the problem can be reduced to scans over the adjacency lists of the graph [47], [55], [52], [54]. In contrast, we argue that the challenging problems lie in *external-memory wedge-based* computation, which unfortunately cannot be solved efficiently by the classical approaches. We use this section to define the specifics of our model and propose a new taxonomy for it.

A. Wedge-Based Computation

Assume a graph $G = (V, E)$ with a set of nodes V and edges E, which may be directed or undirected. Depending on the application, the graph may additionally be weighted, where ϕ_{xy} represents the value attached to edge $(x, y) \in E$ and π_x signifies the weight given to node $x \in V$. Define a *wedge* $\mathcal{P}_{zyx} = \{(z, y, x); \phi_{zy}, \phi_{yx}; \pi_z, \pi_y, \pi_x\}$ to be a simple path of length 2 that begins in z, which we call the *originator*, passes through y, and terminates in x, i.e., $(z, y) \in E$ and $(y, x) \in E$. If no such path exists, $\mathcal{P}_{zyx} = \emptyset$. Fig. 1(a) shows an illustration for directed graphs.

While there are many disjoint wedges scattered across the graph, we are interested in analyzing their collective properties *in the vicinity of each node*. Specifically, suppose $\mathcal{P}_{zx}$ is the set of wedges between z and x, i.e., $\mathcal{P}_{zx} = \cup_{y \in V} \mathcal{P}_{zyx}$. In the example of Fig. 1(b), which is no longer weighted to avoid clutter, this set consists of three wedges passing through (y_1, y_2, y_3). Finally, let $\mathcal{P}_x = \cup_{z \in V} P_{zx}$ be the set of wedges that end in x, which we call a *wedge neighborhood of* x. In Fig. 1(c), this set contains a total of six wedges, originating at z_1, z_2, and y_1.

Definition 1. *Let N_x be the set of neighbors of x and $f(\mathcal{P}_x, N_x)$ be some function that operates in RAM. Then,* wedge-based computation *on G is a process that evaluates $f(\mathcal{P}_x, N_x)$ for every $x \in V$.*

It should be noted that calculations inside f vary between applications. They may involve counting the number originators and intermediate nodes, computing an overlap between N_x and wedge originators in $\mathcal{P}_x$, and detecting intermediate nodes that serve as originators for other wedges. We elaborate on these examples next.

B. Motifs

One important area of graph mining deals with discovering small subgraphs, often called *motifs* [60], that have particular interest in various fields (e.g., biology [43], [68], cybersecurity [38], social networks [36]). We cover two such cases here. The first one is triangle enumeration, which is often formulated as detecting all 3-node cycles in an undirected graph [23], [48]. To reduce CPU complexity, the graph is usually preprocessed using an acyclic orientation, where each edge is given direction such that the resulting digraph has no cycles [7], [71]. Assume N_x keeps in-neighbors in a separate set N_x^- and out-neighbors in N_x^+. Then, triangles enumeration at node x boils down to finding all pairs (z, y) such that z is an in-neighbor of x and there is a wedge (zyx), i.e.,

$$f(\mathcal{P}_x, N_x) = \{(z, y) \mid z \in N_x^-, \mathcal{P}_{zyx} \neq \emptyset\}. \quad (1)$$

In Fig. 1(c), there are three such pairs for x – (y_1, y_2), (z_2, y_3), and (z_2, y_4). The second case, non-induced quadrangle (or 4-cycle) enumeration, identifies all nodes z in the second neighborhood of x that contain at least two distinct paths to x, i.e.,

$$f(\mathcal{P}_x, N_x) = \{P_{zx} \mid |\mathcal{P}_{zx}| \geq 2\}. \quad (2)$$

If the number of paths is $|\mathcal{P}_{zx}| = k \geq 2$, there are $\binom{k}{2}$ quadrangles between z and x. Fig. 1(c) shows three wedges that originate from z_1, two from z_2, and one from y_1, which leads to a total of 4 non-induced quadrangles at node x.

C. Supporters

Define the *neighborhood function* [16], [26], [64] to count the number of nodes at distance no larger than d from x. Intuitively, computation of this metric over all nodes requires

a d-fold self-join over the edge relation, which is usually prohibitively expensive in external memory. However, one particularly useful version of this metric, under the name of *supporters* in the field of web ranking, spam detection, and frontier prioritization [13], [12], [19], [27], uses $d = 2$ to limit the search to the directed wedge-neighborhood of x, which as we show below achieves more reasonable I/O bounds.

In our notation, the supporters algorithm simply obtains the number of unique originators z for each x, except those that are direct in-neighbors of x. This metric can be easily generalized to collect weights π_z from each level-2 supporter. These weights might represent the white-listed status of the node or score from prior ranking in some iterative procedure. With this in mind, the *generalized supporter framework* is given by

$$f(\mathcal{P}_x, N_x) = \{\pi_z \mid z \notin N_x^-, \mathcal{P}_{zx} \neq \emptyset\}. \tag{3}$$

Note that immediate neighbors of x are omitted since their count and weights can be easily inflated by spammers using collusion, e.g., link-exchange alliances [37]. Armed with a list of weights $\{\pi_z\}$, a search engine or its web crawler can use a scalar reducer (e.g., summation) over these weights to arrive at the ranking/reputation score of x. Because computing supporters still requires incredible amounts of I/O in traditional neighbor-based frameworks, this technique has yet to be applied to graphs bigger than RAM [13], [27].

D. Similarity Ranking

Our next application is a recommender system that maintains an undirected bipartite graph between users and movies, where each edge signifies the "likes" relationship. The goal may be for each user x to find the person with the most similar taste. Because each wedge $\mathcal{P}_{zyx}$ indicates that both z and x like movie y, this can be solved by identifying the originator z with the largest set $\mathcal{P}_{zx}$, i.e.,

$$f(\mathcal{P}_x, N_x) = \arg\max_{z \in V} |\mathcal{P}_{zx}|. \tag{4}$$

Alternatively, users can provide a numerical ranking of each movie, where edge weight $\phi_{zy} \in [0, 1]$ may indicate the score assigned by user z to movie y. Then, it might be beneficial for x to identify the user whose average deviation in score from its own is minimal. If z and x both ranked a given movie y, the corresponding wedge $\mathcal{P}_{zyx}$ contains their scores ϕ_{zy} and ϕ_{xy}, which leads to

$$f(\mathcal{P}_x, N_x) = \arg\min_{z:\mathcal{P}_{zx}\neq\emptyset} \sum_{y:\mathcal{P}_{zyx}\neq\emptyset} \frac{|\phi_{zy} - \phi_{xy}|}{|\mathcal{P}_{zx}|}. \tag{5}$$

Additional examples include mining user-activity graphs at Facebook for potential like-minded social groups and studying academic publication graphs connecting keywords with papers to help researchers explore related topics. This problem has seen an abundant research effort [2], [8], [9], [15], [21], [32], [33], [34], [35], [40], [42], [44], [45], [51], [53], [59], [73], [75]; however, these papers usually assume the graph fits in RAM and do not consider external-memory situations.

In terms of function f, many techniques in this field require computing pair-wise intersections of neighbor lists $N(x) \cap N(z)$, which in our notation is P_{zx}, for all (x, z). For example, counting the number of common neighbors [34] reduces to evaluating $|P_{zx}|$, where the final objective might be identical to that in (4). Discovery of nodes most similar according to the Jaccard coefficient [9], [34], [35] can be performed using

$$f(\mathcal{P}_x, N_x) = \arg\max_{z \in V} \frac{|\mathcal{P}_{zx}|}{|N(x)| + |N(z)| - |\mathcal{P}_{zx}|}, \tag{6}$$

where the degree of z can be included in the wedge by setting node weight $\pi_z = |N(z)|$. Usage of edge-weighted graphs has been explored in [32], [34], [59]. These algorithms include a core similarity metric based on

$$f(\mathcal{P}_x, N_x) = \arg\max_{z:\mathcal{P}_{zx}\neq\emptyset} \sum_{y:\mathcal{P}_{zyx}\neq\emptyset} \phi_{zy}\phi_{yx}, \tag{7}$$

which computes the product of weights within each wedge and sums them up across all elements of $\mathcal{P}_{zx}$.

Regardless of the specific f, many of these problems can be generalized to identify the top-k candidate nodes z for each x. Assuming k is a small constant compared to RAM size, keeping $k-1$ extra counters with each node x does not change the asymptotics of I/O complexity in the studied methods.

E. Sparse Matrix Multiplication

Yet another direction is multiplication of large sparse matrices A and B, where the vast majority of prior work is limited to in-memory scenarios [11], [18], [49], [61], [74]. We are aware of only one paper [77] that uses a hybrid method, which keeps A (assumed to be dense) in RAM and streams B (assumed to be sparse) from disk. In contrast, wedge-based computing can evaluate products AB when both matrices are sparse and larger than RAM. Matrix-matrix products compute sums in the form of (7) for each pair of nodes (z, x), where weights ϕ_{zy} come from the graph associated with matrix A and ϕ_{yx} from that associated with B.

F. Taxonomy

We have shown several ways for utilizing wedge-neighborhoods $\mathcal{P}_x$ during processing of graphs. Because the computational model of Definition 1 covers a wide range of scenarios, it would be impossible to list all functions f exhaustively and design a separate algorithm for each. However, there is a way to classify wedge-based computation into three distinct families, each requiring a separate partitioning scheme and exhibiting different I/O complexity, based on the level of *decomposition* [14]they admit. In this context, decomposition refers to breaking the problem into smaller independent sub-problems whose solutions can be aggregated with low complexity to yield the desired result.

Let n be the number of nodes in the graph, m be the corresponding number of edges, T_n be the number of wedges, and M be RAM size. We say that function f belongs to *Category I* if it can process the wedges in $\mathcal{P}_x$ independently of each other. The main advantage of this type of problems

is that they allow *two-dimensional* partitioning of the graph (i.e., by both originator z and intermediate node y) without snowballing the I/O cost. With $p = m/M$ partitions, Category-I problems, where triangle enumeration is the main representative, can be solved in no more than $\min(T_n, \sqrt{p}m)$ I/O [29]. For the example in Fig. 1(c), consider its decomposition into four subgraphs in subfigure (d). It is not difficult to see that function f can detect all three triangles at x by processing each subgraph separately from the others.

Next, suppose f does not fit into Category I. Then, we say the function belongs to *Category II* if it can operate on sets $\mathcal{P}_{zx}$ independently of each other. To understand this better, consider the problem of discovering undirected quadrangles (i.e., 4-cycles) involving node x in Fig. 1(c), which contains four such motifs (i.e., $(z_1y_1y_2x), (z_1y_2y_3x), (z_1y_1y_3x), (z_2y_3y_4x)$). In the 2D-partitioned version of Fig. 1(d), only the leftmost subgraph contains a 4-cycle (i.e., $(z_1y_1y_2x)$). Thus, function f applied independently to each subgraph would be missing three out of the four quadrangles.

To overcome this issue, the application would need to write all triples $(x, z, |\mathcal{P}_{zx}|)$ to disk, where $|\mathcal{P}_{zx}|$ is the number of wedges between z and x, sort them, and reduce the result on the counter field. Similarly, supporter enumeration in Fig. 1(d) causes function f to detect each of z_1, z_2 twice. In order to eliminate these duplicates, which is a key feature of supporter-based ranking that allows it to stay spam-resilient, the user would need an external-memory sort on pairs (x, z). In the worst case, the sort involves all T_n tuples, which carries some exorbitant cost. On the other hand, one-dimensional partitioning of $\mathcal{P}_x$ on the originator z, as illustrated in Fig. 1(e), allows discovery of the first three quadrangles in the left subgraph and the other one in the right. For the same reasons, similarity ranking [15], [33], [34], [42], [45], [73], spam avoidance [13],[27], and sparse matrix multiplication [18], [61], [74], [77] fall into Category II.

Finally, wedge-based problems that do not allow 1D wedge decomposition belong to *Category III*. This typically means that the entire set of wedges $\mathcal{P}_x$ needs to be witnessed in RAM in order for the computation to succeed. One example would be finding 4-node cliques [6], which entails discovering all triples of wedges $(\mathcal{P}_{zyx}, \mathcal{P}_{zwx}, \mathcal{P}_{ywx})$ in the set $\mathcal{P}_x$, where $z \in N_x$. The main challenge here is that 1D partitioning may put $\mathcal{P}_{zyx}$ and $\mathcal{P}_{ywx}$ into separate files, which precludes detection of the corresponding motif. Efficient I/O in such cases is beyond the scope of this paper.

G. Existing Approaches

External-memory algorithms in Category I are well-studied in triangle enumeration [29], [63] and generally possess the highest I/O efficiency because of 2D decomposition. We do not cover them here.

For Category-II problems, the two main classes of solutions are outlined in [27], where the goal is to count supporters (i.e., unique nodes at distance 2 from each x) in directed graphs. Their first method, which we call *ACC*, attempts to assemble in RAM the entire list $\{z : |P_{zx}| > 0\}$ of originators z for each x. This is done by retaining a small portion of nodes x with their in-neighbors y in memory and performing a scan of the entire in-graph G^- on disk to detect $y \leftarrow z$ relationships. Letting X_i be the out-degree of node i and Y_i its in-degree, the I/O cost of ACC is [27]

$$c_n(ACC) = \frac{m^2}{M}\Big(1 + \frac{E[Q_i]}{E[Y_i]}\Big), \tag{8}$$

where Q_i is the number of level-2 supporters at node i. Besides a huge multiplier $E[Q_i]$ to the quadratic term m^2, this method equires that $\max_i Q_i \leq M$, a hefty constraint by itself.

The second approach in [27] adopts the MapReduce route. For each node y, it reads the in-list $N^-(y)$ and out-list $N^+(y)$ concurrently from in/out graphs, emitting all pairs (x, z) from the Cartesian product $N^+(y) \times N^-(y)$. The reduce phase sorts pairs using x as the key and counts the number of unique supporters z. This method initially writes $T_n = \sum_i X_i Y_i$ pairs to disk and requires multiple merge passes during external-memory sorting. Its total I/O cost is

$$c_n(MR) = T_n + T_n \log_s\Big(\frac{T_n}{M}\Big), \tag{9}$$

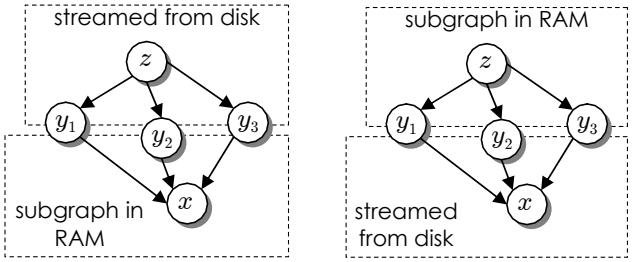
assuming an s-way merge of sorted blocks. In practice, duplicate pairs (x, z) can be eliminated during each merge phase and the I/O may be slightly better than (9), although the asymptotics remain the same as $m \to \infty$. While writing all wedges to disk and then sorting them may seem inefficient, this approach has received interest even for simpler tasks (e.g., triangle enumeration [25], [66]).

Perhaps a more straightforward application of MapReduce to Category-II problems is to apply a general-purpose sort library (e.g., Hadoop [10], STXXL [31]) directly to T_n key-value pairs. While there are other ways to accomplish this task – table joins in databases [3], [58], [67], [70], [72] and dedicated graph libraries [47], [55], [79] – we consider them to also fall under the MapReduce umbrella because they essentially require sorting/merging/grouping T_n items on disk.

In real graphs, where the in/out degree is highly correlated and heavy-tailed, the I/O complexity in (8)-(9) can be quite dramatic (i.e., 3-5 orders of magnitude larger than G depending on RAM size). For example, the 14-GB webgraph in [27] requires sorting 49 TB in a system with 8 GB of RAM. As an alternative, it might be tempting to give up sequential I/O and engage SSDs in retrieving random wedges from disk; however, that same graph now requires 1.8B seeks and 24 TB of I/O, which are not negligible either. Thus, our aim is to understand whether Category-II problems can be solved with lower overhead, model their CPU and I/O cost, and compare our methods against alternatives from prior work.

III. Decomposable Wedge Computing (DWC)

This section introduces our approach for solving Category-II wedge problems using external memory.



(a) Method A ($x \in V_r$) (b) Method B ($z \in V_r$)

Fig. 2. Partitioning options.

Algorithm 1: Partitioning in DWC_1-A

```
1 while !eof(G+) do
2     load next neighbor list (y, N_y+) from G+
3     for r = 1 to p do
4         N_y+(r) = N_y+ ∩ V_r
5         if N_y+(r) ≠ ∅ then
6             write (y, N_y+(r)) to G_r+
```

Algorithm 2: Partitioning in DWC_1-B

```
1 while !eof(G−) do
2     load next neighbor list (y, N_y−) from G−
3     for r = 1 to p do
4         N_y−(r) = N_y− ∩ V_r
5         if N_y−(r) ≠ ∅ then
6             write (y, N_y−(r)) to G_r−
```

A. Setup

For an algorithm $\mathcal{A}$, define $c_n(\mathcal{A})$ to be its I/O complexity, which is the number of edges exchanged with the disk (i.e., read or written during the execution). Unlike the classical I/O model of [5], which counts the number of transfers using block size B, all studied algorithms assume *streaming* (i.e., sequential) access to disk. Thus B is no longer necessary; instead, the I/O-related runtime is fully determined by $c_n(\mathcal{A})$ and the sequential read/write speed of the disk [65].

To avoid carrying edge weights ϕ_{zy} and node weights π_z in all formulas and algorithms, we first make a note that inclusion of (ϕ_{zy}, π_z) increases the *size* of each edge three-fold, but does not change their *number*. Similarly, techniques that solve wedge-related problems in directed graphs can be easily adopted to undirected cases, i.e., by replacing each edge (x, y) with a directed pair $x \to y$ and $y \to x$. As a result, we omit weights in the discussion below and only focus on directed graphs, where wedges in $\mathcal{P}_{zx}$ are assumed to have a fixed orientation $z \to y \to x$ (see Fig. 1).

Let the input consist of an out-graph $G^+ = (V, E^+)$, where $n = |V|$ is the number of nodes and $m = |E^+|$ is the number of edges. Additionally, assume existence of an inverted version $G^- = (V, E^-)$ that contains the corresponding in-edges. As before, suppose N_i^+ represents the adjacency list of i in G^+, i.e., its out-neighbors, and N_i^- serves the same purpose in G^-. We assume that source nodes and neighbor lists are stored in ascending order of IDs, which for performance reasons are relabeled into a compact range 0 to $n-1$. Finally, we use $X_i = |N_i^+|$ to represent the out-degree, $Y_i = |N_i^-|$ the in-degree, and $d_i = X_i + Y_i$ the total degree of node i.

B. Partitioning

We next introduce our first framework, which we call DWC_1, for handling Category-II problems in external memory. Recall that these problems require that all wedges in $\mathcal{P}_{zx}$, i.e., between nodes z and x, be visible in RAM simultaneously. To make the computation feasible with restricted RAM, the original graph must be chunked into smaller units that allow retracing of all wedges in $\mathcal{P}_{zx}$. Since a subgraph can guarantee visibility of only the edges adjacent to either z or x, it must be combined with streaming passes over additional graph data that brings the missing edges into RAM. We deal with these issues in the rest of the section.

Assume a partitioning scheme on V that splits the nodes into $p \geq 1$ pair-wise non-overlapping sets $V_1, \ldots, V_p$, where $\cup_i V_i = V$, and let M be the size of RAM in edges. As in [28], there are two options for splitting the graph, i.e., along either the destination or source node of each directed edge. In the former case, which we call *Method A*, we partition G^+ into p subgraphs $G_1^+, \ldots, G_p^+$ such that an edge $y \to x$ belongs to G_r^+ iff $x \in V_r$. In the latter case, which we call *Method B*, we split G^- into $G_1^-, \ldots, G_p^-$ such that $y \leftarrow z$ belongs to G_r^- iff $z \in V_r$. This is shown in Fig. 2 and Algorithms 1-2.

In order to achieve the best I/O performance, i.e., the lowest $p = m/M$, the subgraphs must be load-balanced to the size of RAM. This requires setting up boundaries $a_1, \ldots, a_{p+1}$, where $a_1 = 1$ and $a_{p+1} = n+1$, such that node i belongs to partition V_r iff $a_r \leq i < a_{r+1}$. In Method A, this is accomplished using

$$\sum_{i=a_r}^{a_{r+1}-1} Y_i = M \tag{10}$$

and in Method B using

$$\sum_{i=a_r}^{a_{r+1}-1} X_i = M. \tag{11}$$

Deciding partition boundaries requires one scan over the degree sequence, which has negligible I/O cost, i.e., n. Also note the implicit constraints $\max_i Y_i \leq M$ for Method A and $\max_i X_i \leq M$ for Method B. On real graphs, where the largest in-degree is often much higher than the largest out-degree, Method B is less restrictive.

C. Computation

Algorithm 3 shows the logic behind DWC_1-A's operation. For each partition $r \in [1, p]$, Line 1 initializes an array v of output values, one for each unique $x \in V_r$. In Lines 3-4, the method loads the corresponding partition G_r^+ into RAM and sets up a hash table that maps each node y to its partial out-list $N_y^+(r)$. As a result of this operation, the hash table H contains all relationships $y \to x$ for $x \in V_r$. The algorithm then proceeds to scan the entire out-graph G^+ to discover the missing edges $z \to y$ (Lines 5-6). For a given z, it is possible

Algorithm 3: Wedge computing under type-A partitioning

```
1  for r = 1 to p do ◁ iterate over all partitions
2      v = array of |V_r| zeros                          ◁ initialize output values
3      load G_r^+ = {(y, N_y^+(r))} into RAM
4      H = hash table built from G_r^+                   ◁ maps nodes y to out-lists
5      while !eof(G^+) do
6          load next neighbor list (z, N_z^+) from G^+
7          for all x reachable in two hops from z using H do
8              construct (P_zx, N_x^-) from (z, N_z^+, H)
9              v[x − a_r] = h(v[x − a_r], f(P_zx, N_x^-))   ◁ reducer
10     for all v[i] ≠ 0 do
11         append (i + a_r, v[i]) to file F_out
12     set file pointer in G^+ to the start
```

Algorithm 4: Wedge computing under type-B partitioning

```
1  for r = 1 to p do ◁ iterate over all partitions
2      load G_r^- = {(y, N_y^-(r))} into RAM
3      H = hash table built from G_r^-                   ◁ maps nodes y to in-lists
4      while !eof(G^-) do
5          load next neighbor list (x, N_x^-) from G^-
6          v = 0                                         ◁ initialize output value for x
7          for all z reachable in two hops from x using H do
8              construct P_zx from (x, N_x^-, H)
9              v = h(v, f(P_zx, N_x^-))                  ◁ reducer
10         if v ≠ 0 then
11             append (x, v) to file F_out(r)
12     set file pointer in G^- to the start
```

to identify all targets x reachable in two hops via H, construct a pair of sets $(\mathcal{P}_{zx}, N_x^-)$, invoke the user-provided function f to perform the desired calculation, and use a reducer function h to accumulate the result across different z (Lines 8-9).

Note that $x - a_r$ specifies the sequence number of x within partition r, which we use as an index into array v. Applying the user-provided reducer function h in Line 9 updates $v[x - a_r]$ as many times as there are unique nodes z that can reach x at distance two. After the scan of G^+ reaches the end, all non-zero values in v, together with the corresponding node IDs, are saved to disk, which is done by appending them to some file F_{out}. Because all wedges $\mathcal{P}_{zx}$ for $x \in V_r$ are witnessed during processing of partition r, nodes x appearing in F_{out} are both unique and sorted in ascending order.

The main caveat of DWC$_1$-A is that (10) needs to allow for counter values v when selecting partition boundaries. If the result of function h has size k_i for node i, boundaries can be computed using

$$\sum_{i=a_r}^{a_{r+1}-1} (Y_i + k_i) = M, \tag{12}$$

which does not change the rest of the algorithm. This carries low cost when h is a) scalar (e.g., summation or max) or b) limited to storing the top-k result from some calculation, where k is small compared to the average degree. However, if merge function h needs to accumulate large lists of values, e.g., all unique supporters z, the overhead of v may be substantial.

We next examine how DWC$_1$-B overcomes this problem. Its logic is illustrated in Algorithm 4, which is pretty similar to the one just discussed, except there is an important difference in how v is created. Because wedge traversals begins with x rather than z, all sets $\mathcal{P}_{zx}$ for different z are available to DWC$_1$-B back-to-back (Lines 6-9). Therefore, v is no longer an array, but rather a single entity. After applying all needed reducer operations to v, its value can be immediately saved to disk in Line 11. As a result, boundary calculation in (11) is still correct, which means that I/O performance of Algorithm 4 does not depend on the size of values producer by h. However, since the same x may be seen in multiple partitions, the result (x, v) must be written into a separate file $F_{out}(r)$ for each r and later merged. Because the output files contain nodes x in ascending order, the merge operation requires sequential scans of total size $\tau(p) = \sum_{r=1}^{p} |F_{out}(r)|$, i.e., no additional sorting. We put bounds on this cost below.

D. I/O Analysis

For the results that follow, assume that value size k_i attached to node i is independent of the node's in-degree Y_i. Taking into account our discussion earlier in the section, where DWC$_1$-A uses an updated model (12) and DWC$_1$-B requires a merge at the end, yields the following result.

Theorem 1. *DWC$_1$-A needs at least $p = (m + E[k_i]n)/M$ partitions and has total I/O*

$$c_n(1A) = \frac{m^2}{M}\Big(1 + \frac{E[k_i]}{E[X_i]}\Big), \tag{13}$$

where $E[X_i] = m/n$ is the average degree of the graph. DWC$_1$-B needs at least $p = m/M$ partitions and I/O

$$c_n(1B) = \frac{m^2}{M}, \tag{14}$$

plus an additional $\tau(p)$ to merge the output files.

Proof. Re-writing (12) and taking a summation over all r

$$\sum_{r=1}^{p} \sum_{i \in V_r} (Y_i + k_i) = pM. \tag{15}$$

Splitting the sums produces

$$m + \sum_{r=1}^{p} \sum_{i \in V_r} k_i = m + nE[k_i] = pM, \tag{16}$$

from which it follows that $p = (m + E[k_i]n)/M$. Because DWC$_1$-A reads the graph p times, its total cost is pm, which is given by (13).

On the other hand, SNF-B can use $p = m/M$ partitions, which follows from (11), and achieve runtime complexity $pm = m^2/M$. Adding the merge overhead yields (14). □

To decide which method A or B is better, we need to analyze the merge overhead, whose upper bound is derived next.

Theorem 2. *The I/O merge cost in DWC$_1$-B is bounded by*

$$\tau(p) \le mE[k_i] \min\Big(1, \frac{n}{M}\Big). \tag{17}$$

Algorithm 5: Partitioning in DWC_2-A

```
1 while !eof(G+) and !eof(G−) do
2     load list (y, N_y^+) from G+ and (y, N_y^−) from G−
3     for r = 1 to p do
4         N_y^+(r) = N_y^+ ∩ V_r
5         if N_y^+(r) ≠ ∅ and N_y^− ≠ ∅ then
6             write (y, N_y^+(r)) to G_r^+
7             write (y, N_y^−) to S_r^−
8     for r = 1 to p do
9         S_r^+ = InvertGraph (S_r^−)
```

Algorithm 6: Partitioning in DWC_2-B

```
1 while !eof(G+) and !eof(G−) do
2     load list (y, N_y^+) from G+ and (y, N_y^−) from G−
3     for r = 1 to p do
4         N_y^−(r) = N_y^− ∩ V_r
5         if N_y^−(r) ≠ ∅ and N_y^+ ≠ ∅ then
6             write (y, N_y^−(r)) to G_r^−
7             write (y, N_y^+) to S_r^+
8     for r = 1 to p do
9         S_r^− = InvertGraph (S_r^+)
```

Proof. Assume that Algorithm 4 writes $k_i(r)$ values for node i in partition r. This might be smaller than k_i because an insufficient number of supporters z was found in the partition. Then, we get that

$$\tau(p) = \sum_{i=1}^{n}\sum_{r=1}^{p} k_i(r) \leq \sum_{i=1}^{n} k_i \min(Y_i, p) \tag{18}$$

since node i cannot appear in more than $\min(Y_i, p)$ partitions. We therefore get two upper bounds out of (19). First, using the fact that k_i and Y_i are independent,

$$\tau(p) \leq \sum_{i=1}^{n} k_i Y_i = mE[k_i] \tag{19}$$

and

$$\tau(p) \leq \sum_{i=1}^{n} k_i p = \frac{mnE[k_i]}{M}. \tag{20}$$

Combining the two cases, we obtain (17). □

It is now easy to see that DWC_1-B is always no worse than DWC_1-A, making it a winner in this comparison. In fact, when $E[k_i] \leq \max(E[d_i], m/M)$, the cost of the final merge using reducer h is asymptotically no different from that of Algorithm 4, which satisfies the decomposability conditions introduced earlier.

E. Improvements

As the graph gets larger, DWC_1's quadratic I/O and hash-table lookup complexity may eventually become a bottleneck. In order to achieve more efficient external-memory operation, our next algorithm for graph partitioning, which we call DWC_2, couples each subgraph with an auxiliary file that contains only the edges relevant to that partition. This is illustrated in Algorithm 5 for DWC_2-A. The process begins by sequentially reading both G^+ and G^- to discover nodes y that have non-zero in/out degree. This works because both graphs are sorted by y. For those nodes that have at least one out-neighbor $x \in N_y^+(r)$ from partition r, we store the relevant portion of N_y^+ into subgraph G_r^+, which is the same as before, but additionally write the entire vector of in-neighbors into an auxiliary stream file S_r^- (Line 7).

Note that S_r^- almost certainly cannot fit in RAM, which requires the application to sequentially load its edges from disk. Because nodes z are now scattered in random locations throughout S_r^-, construction of sets $\mathcal{P}_{zx}$ becomes difficult. As a result, Algorithm 5 has to invert each auxiliary graph in Line 9 before it becomes usable. Now, armed with pairs (G_r^+, S_r^+), we can invoke Algorithm 3, where Lines 5-6 are modified to read from S_r^+ instead of G^+, to accomplish the same functionality, i.e., wedge-based computing on the entire graph. The rest of the operations remain identical. Algorithm 6 shows partitioning for DWC_2-B, which is similar, except it flips the in/out relationships on all edges and calls Algorithm 4 to perform the computation.

We keep in mind that DWC_2-B still has a separate merge overhead $\tau(p)$ from (17) and now focus on the cost to run the main portion of Algorithms 3-4. Define p_y^+ to be the number of partitions into which the out-neighbors of y are split and let p_y^- measure the same for the in-neighbors. Further recall that $T_n = \sum_{i=1}^{n} X_i Y_i$ is the number of wedges in the graph.

Theorem 3. *The I/O of DWC_2-A is given by*

$$c_n(2A) = \sum_{i=1}^{n} p_i^+ Y_i \leq \sum_{i=1}^{n} \min(X_i, p) Y_i \tag{21}$$

and that of DWC_2-B by

$$c_n(2B) = \sum_{i=1}^{n} p_i^- X_i \leq \sum_{i=1}^{n} \min(Y_i, p) X_i, \tag{22}$$

where both formulas are no larger than $\min(T_n, m^2/M)$.

If $E[X_i Y_i]$ stays bounded as $m \to \infty$, both methods produce I/O that is a *linear* function of m. The upper bound at the end of Theorem 3 also shows that DWC_2 is at least as good as its DWC_1 counterpart, which holds for all graphs and all memory sizes M. While it might be tempting to dismiss DWC_1 from future consideration, it is possible to encounter situations where the preprocessing delay to invert auxiliary files is undesirable, in which case DWC_1 might still be a viable candidate.

Although it is difficult to further simplify the result of Theorem 3 without access to the specific graph and its node-ID assignment, we can derive the expected I/O cost assuming the in/out degree sequence is known.

Theorem 4. *Assume a random assignment of node labels and let $\epsilon = 1/p$. Then, the expected amount of I/O in DWC$_2$ is*

$$E[c_n(2A)] = p\sum_{i=1}^{n}[(1-(1-\epsilon)^{X_i})Y_i] \tag{23}$$

$$E[c_n(2B)] = p\sum_{i=1}^{n}[(1-(1-\epsilon)^{Y_i})X_i]. \tag{24}$$

Proof. We focus only on DWC$_2$-B since DWC$_2$-A is symmetric (i.e., replaces X_i with Y_i and vice versa). The objective is to derive the following expectation, where we explicitly condition on Y_i to show that p_i^- depends on it

$$E[c_n(2B)] = \sum_{i=1}^{n} E[p_i^-|Y_i]X_i, \tag{25}$$

where p_i^- is a random variable that depends on assignment of nodes to partitions. Let C_{ij} be the partition of the j-th in-neighbor of node i. When averaging over all shuffles of IDs, it follows that each neighbor has the same probability $\epsilon = 1/p$ to be from any of the partitions, i.e., for all $r = 1, \ldots, p$

$$P(C_{ij} = r) = \epsilon. \tag{26}$$

Next, define B_{ir} to be the number of i's in-neighbors from partition r, i.e.,

$$B_{ir} = \sum_{j=1}^{Y_i} \mathbf{1}_{C_{ij}=r}. \tag{27}$$

Counting the number of partitions for which i has at least one in-neighbor yields

$$p_i^- = \sum_{r=1}^{p} \mathbf{1}_{B_{ir}\geq 1}. \tag{28}$$

Assuming a large-enough number of colors and a sufficiently large graph, B_{ir} can be viewed as binomial. Therefore,

$$P(B_{ir} \geq 1|Y_i) = 1-(1-\epsilon)^{Y_i} \tag{29}$$

and

$$E[p_i^-|Y_i] = p(1-(1-\epsilon)^{Y_i}). \tag{30}$$

Combining with (25), we get (24). □

In some cases, Theorem 4 may be sufficient to make a choice between DWC$_2$-A/B, but oftentimes their I/O differs only by a small amount. We therefore have to consider other factors (e.g., CPU cost, speed of update operations) to decide which one to use. This is our next topic.

F. In-Memory Complexity

The CPU cost of introduced algorithms depends on two metrics – the number of wedges T_n visited during execution and the number of hash-table lookups l_n in H. The latter value is determined by the volume of attempted hits on $y \in N_z^+$ in Algorithm 3 and $y \in N_x^-$ in Algorithm 4. Note that depending on how sparse the graph is, it is possible for either l_n or T_n to dominate the other. The next result shows that Algorithms 3-4 visit each wedge exactly once regardless of how many partitions there are, but the number of lookups grows with p.

Theorem 5. *All methods introduced in the paper require T_n wedge traversals and c_n lookups in the hash table. Furthermore, the expected size of the hash table in Algorithm 3 is*

$$\sum_{i=1}^{n}[1-(1-\epsilon)^{X_i}] \tag{31}$$

and that in Algorithm 4

$$\sum_{i=1}^{n}[1-(1-\epsilon)^{Y_i}]. \tag{32}$$

Since this result shows that $l_n = c_n$, the lower I/O volume in DWC$_2$ also helps it reduce the CPU cost compared to DWC$_1$, i.e., usage of auxiliary files has multi-faceted benefits. A more subtle issue arises in the updates to counters performed by the user-supplied functions (f, h), which in many cases is sensitive to the direction from which x is approached. In type-A partitioning, we follow edges starting from z and increment counters of x at dispersed locations throughout memory. On the other hand, type-B partitioning starts from x and collects all updates to its value counter in one variable that stays in L1 cache. This can be seen by comparing Line 9 of Algorithms 3-4. As a result, even if both methods perform a similar amount of instructions, *B usually has them running a lot faster*.

IV. Asymptotic Cost of I/O

While Theorem 3 shows trivial bounds for DWC$_2$, it is beneficial to understand the growth rate of (21)-(22) in more tangible terms. For example, are these bounds tight? How fast can T_n grow in the best/worst case? How much better is DWC$_2$ compared to DWC$_1$?

A. Growth Rates

Our analysis will focus on the asymptotic behavior of cost as $n \to \infty$. To this end, define the *power-law scaling rate* of a function c_n as

$$\gamma(c_n) = \lim_{n\to\infty} \frac{\log c_n}{\log n}. \tag{33}$$

For example, $c_n = 3n^2/\log(n)$ has scaling rate 2. Assume the average degree $d = m/n$ scales at rate $\gamma(d) = a \in [0, 1]$. Note that a covers a wide range of graphs, where 0 represents sparse cases (e.g., constant average degree) and 1 means dense (e.g., complete graphs). Suppose the available memory scales as $\gamma(M) = r$, where $r \in [a, 1+a]$. We need $r \geq a$ to fit the largest degree in RAM, which is a requirement for (10)-(11). To put these definitions to use, notice that $\gamma(m) = 1 + a$ and the scaling rate for DWC$_1$-A/B is

$$\gamma(m^2/M) = 2 + 2a - r. \tag{34}$$

The next challenge is to determine $\gamma(T_n)$, which is necessary for understanding the number of wedges witnessed by all studied algorithms and the amount of I/O in DWC$_2$. Our approach is to construct directed graphs with the highest

possible T_n for a given pair of (a, r) and then show that such graphs have I/O in (21)-(22) that scales at the same rate as T_n, i.e., that the bound is tight.

B. Adversarial Degree Sequences

To determine the properties of $T_n = \sum_{i=1}^{n} X_i Y_i$, we first phrase the problem as finding two non-negative vectors $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_n)$, where $\sum_i u_i = \sum_i v_i = m$ is fixed, such that their dot-product is maximized. For the problem to be interesting, we additionally impose a constraint $u_i \leq U$ and $v_i \leq U$ for all i, where U is an upper bound on the degree. For now, we abstract away the fact that these sequences have to be *digraphic* (i.e., be realizable by a directed graph), but return to this issue later.

Our first step is to understand how to obtain $\mathbf{u}$ that maximizes $\sum_i u_i^2$. For this, we need the following definition.

Definition 2. *Assume two vectors* $\mathbf{u}$ *and* $\mathbf{v}$ *are sorted in descending order to obtain vectors* $\mathbf{u}'$ *and* $\mathbf{v}'$*. Then,* $\mathbf{u}$ *is said to majorize* $\mathbf{v}$*, which is written as* $\mathbf{u} \succ \mathbf{v}$*, if for all* $k = 1, \ldots, n$ *the prefix sum of the first* k *elements in* $\mathbf{u}'$ *is no smaller than that in* $\mathbf{v}'$ *, i.e.,* $\sum_{i=1}^{k} u_i' \geq \sum_{i=1}^{k} v_i'$*.*

An important property of majorization is that $\mathbf{u} \succ \mathbf{v}$ implies $F(\mathbf{u}) \geq F(\mathbf{v})$ for all Schur-convex functions F. A special case of this result is stated in the next lemma.

Lemma 1. *If* $\mathbf{u} \succ \mathbf{v}$*, then* $\sum u_i^2 \geq \sum_i v_i^2$*.*

Therefore, the largest sum of squares is attained by the vector $\mathbf{u}^*$ that majorizes all other vectors, which can be constructed by setting $k = \lfloor m/U \rfloor$ largest values to U, the next one to $m \pmod U$, and filling the remaining values with zeros. Note that $\mathbf{u}^*$ is unique. Armed with this result, we next show that the worst-case degree sequence has to use vector $\mathbf{u}^*$ for both in/out degree sequence.

Lemma 2. *The maximum sum* $\sum_{i=1}^{n} u_i v_i$ *is achieved by setting* $\mathbf{u} = \mathbf{v} = \mathbf{u}^*$*.*

Proof. We prove this by contradiction. Suppose there exists a pair of vectors $\mathbf{u}, \mathbf{v}$ such that their $\sum_{i=1}^{n} u_i v_i > \sum_{i=1}^{n} (u_i^*)^2$. Now define another vector $\mathbf{w}$, where $w_i = \sqrt{u_i v_i}$. Since we know that $\mathbf{u}^* \succ \mathbf{w}$, it must be that $\sum_{i=1}^{n} (u_i^*)^2 \geq \sum_{i=1}^{n} w_i^2$, which contradicts the assumption we just made. □

We thus obtain that T_n cannot be larger than $kU^2 = mU$ for any pair of sequences $\{X_i\}, \{Y_i\}$. Since the largest degree in a graph suitable for external-memory wedge computing must satisfy $U \leq \min(n-1, M)$, we obtain that

$$\gamma(T_n) \leq 1 + a + \min(r, 1). \tag{35}$$

However, this does not mean there exists a graph that realizes this bound. Our next result sheds light on this issue.

Theorem 6. *For any* $w \in [0, 1 + a + \min(r, 1)]$*, there exists a directed graph that achieves* $\gamma(T_n) = w$*.*

Proof. We modify the optimal sequence $\mathbf{u}^*$ constructed earlier to include $n - k$ nodes with in/out degree equal to some value $L > 0$ (instead of zero). We first create an adversarial (worst-case) graph. The edge process links the top-k nodes in a complete graph, which satisfies k^2 edges. However, the remaining $kU - k^2$ edges must be thrown towards the small-degree nodes. Thus, their total degree $(n-k)L$ must be sufficient to support $k(U-k)$ incoming edges. This results in $(n-k)L = k(U-k)$, or

$$L = \frac{k(U-k)}{n-k}. \tag{36}$$

We now determine k. Note that this requires solving a quadratic equation

$$kU + (n-k)L = m, \tag{37}$$

which can be simplified by raising L to $kU/(n-k)$. This makes the graph less dense, but nevertheless allows us to achieve the upper bound in (35). We thus get

$$kU + kU = m, \tag{38}$$

from which $k = m/(2U)$. As a result, $T_n \geq kU^2 = mU/2$ and $\gamma(T_n) \geq 1 + a + \min(r, 1)$. Since this matches the upper bound, it follows that the omitted terms related to L cannot affect the asymptotics of T_n.

Next, by increasing L towards the average degree m/n allows us to achieve a full spectrum between the upper bound (35) and d-regular graphs with $\gamma(T_n) = 1 + 2a$. It should also be noted that (35) cannot be smaller than $1 + 2a$ since $U \geq a$ must hold, i.e., the largest degree is no smaller than the average. We omitted explicit restriction on U since $r \geq a$ already guarantees that $\min(1, r) \geq a$.

To construct examples with $\gamma(T_n) < 1 + 2a$, we turn to bipartite graphs. We split the nodes in half and direct edges from each node in the first set S_1 to random nodes in the other set S_2. This trivially results in $X_i Y_i = 0$ for all i, which makes $T_n = 0$ as well. To achieve a flexible number of wedges, we can assign custom degree and flip the direction of edges on k special nodes in S_1. The goal is to give them a desired product $X_i Y_i \sim n^b$, where $b \in [0, 2a]$. This produces $\gamma(T_n) = \gamma(k) + b$. By varying k and b, we can achieve any scaling rate in $[0, 1 + 2a]$. □

C. I/O Complexity Revisited

We now come back to the issue of deducing the scaling rate of the actual I/O from (21)-(22). While we know from Theorem 6 that $T_n = \sum_i X_i Y_i$ can achieve the upper bound (35), our next result shows that I/O cost c_n hits the same worst-case performance in adversarial graphs. Not only that, but there are cases that reach the other upper bound m^2/M as well.

Theorem 7. *The scaling rate of DWC*$_2$ *I/O is tightly upper-bounded by*

$$\min(1 + a + \min(r, 1), 2 + 2a - r). \tag{39}$$

Proof. We show that for every combination of (a, r) there exists a graph whose I/O is asymptotically equal to the upper

bound $\min(\gamma(T_n), \gamma(m^2/M))$. We start with the adversarial graph from Theorem 6, where we have two choices. The first one is the rate at which p grows exceeds that of U, which is equivalent to $1 + a - r > \min(1, r)$. In this case, the number of partitions per node's degree tends to infinity, which means there will be at most U copies of the each node's list in the auxiliary file. This is equivalent to $\min(X_i, p)$ and $\min(Y_i, p)$ in (21)-(22) both resolving in favor of the degree. As a result, the total I/O rate tends to that of $\gamma(T_n)$. On the other hand, when $1+a-r < \min(1, r)$, there are infinitely many neighbors compared to the number of partitions. This can be viewed as both $\min$ functions resolving in favor of p, which makes the total I/O cost equal to m^2/M. We thus get that the scaling rate of DWC_2 on adversarial graphs is given by

$$\begin{cases} 1 + a + \min(r, 1) & 1 + a - r > \min(1, r) \\ 2 + 2a - r & otherwise \end{cases}. \tag{40}$$

Combining the two cases, we get (39). □

To put this in perspective to related work, we have the next result.

Theorem 8. *The scaling rate of ACC I/O is contained in the range* $[2+2a-r, 2+2a-r+\min(r, 1)]$ *and that of MapReduce equals* $\gamma(T_n)$.

Proof. For ACC, the number of supporters in the graph ranges from 0 to T_n, which constrains $E[Q_i]$ to the range $[0, T_n/n]$. This leads to

$$\gamma\Big(\frac{E[Q_i]}{E[Y_i]}\Big) \in [0, \gamma(T_n) - 1 - a], \tag{41}$$

which coupled with (34) and (35) produces the stated range.

The result for MapReduce directly follows from (9). □

While (9) scales better than (8), MapReduce often requires a huge amount of disk space to store all T_n wedges. In cases when insufficient resources exist, the read-only approach of ACC may be the only viable solution from prior work. The same dichotomy exists between DWC_1 and DWC_2, except the amount of data in auxiliary stream files is much less than T_n in practical situations.

D. Discussion

We now provide a graphical illustration of the results derived in this section. Fig. 3(a) begins with ACC, where the x-axis shows the scaling rate r of RAM as $n \to \infty$, which ranges from a to $1 + a$, and the y-axis displays the growth rate of I/O. The lower bound of ACC is a dashed line BI, which is the m^2/M best-case scenario where the average number of supporters per node $E[Q_i]$ is negligible. Predictably, as r increases (i.e., more RAM is available), the dashed line decreases. However, the upper bound AFH depends on the graph structure rather than RAM and in certain portions (segment AF) remains constant.

In Fig. 3(b), MapReduce begins at $1 + 2a$ (point C) and monotonically increases towards $2 + a$ (point G). This can be explained by the fact that bigger memory size M allows

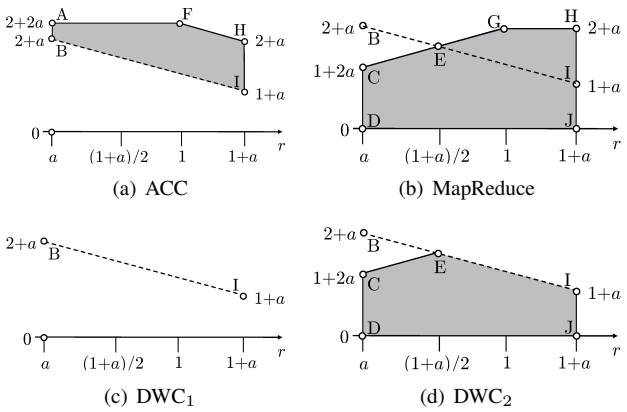


Fig. 3. Scaling rates.

adversarial graphs with larger maximum degree U, which in turn increases T_n. Interestingly, there is a regime contained inside the convex hull of points EGHI, where it is possible for ACC to beat MapReduce. This corresponds to relatively large memory scenarios with $r \geq (1 + a)/2$. For example, in sparse graphs with constant average degree (i.e., $a = 0$), this means M grows at least as fast as $\sqrt{n}$. In these cases, ACC can exhibit I/O cost that is $\Theta(n)$ better (i.e., where r is close to $1 + a$ and almost the entire graph fits in RAM). On the opposite end of the spectrum, where $r = a$ is the smallest possible, MapReduce wins by at least a factor of n^{1-a}.

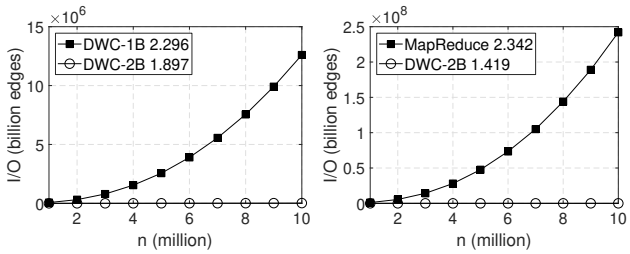
Fig. 3(c) models the cost of DWC_1. Unlike the pentagons we saw in the previous two cases, where the specific graph determines which internal point is realized in practice, the I/O here is limited to a single line. In the worst case, i.e., $r = a = 1$ in complete graphs, DWC_1 has cubic I/O complexity (point B). While it is trivially no worse than ACC for all graphs and memory configurations, it is sometimes inferior to MapReduce, i.e., for $r < (1 + a)/2$. Thus, there is no clear winner among the first three approaches.

This is solved in Fig. 3(d), which shows the result for DWC_2. This is also a pentagon, but this time its upper bound lies entirely below the curves of the other methods. The largest advantage of DWC_2 over DWC_1 emerges at $r = a = 0$ (point C), where the difference is by a factor of n. This corresponds to sparse graphs being processed on machines with small RAM size compared to m. Given that large social and Internet graphs have trillions of edges and average degree below 100, they match the best-case scenario for DWC_2.

Fig. 4 demonstrates accuracy of our asymptotic models. In part (a), we scale the average degree as $\Theta(n^{0.4})$ and allow $\Theta(\sqrt{n})$ memory. We construct adversarial graphs from Theorem 6 and run Algorithm 4 on them, which produces the actual I/O c_n. We then fit a power function to these curves to determine the scaling rate. As shown in the figure, both functions follow closely the predicted rates $n^{2.3}$ and $n^{1.9}$. Fig. 4(b) illustrates a scenario with the average degree $\Theta(n^{0.3})$ and memory size $\Theta(n^{1.1})$, where MapReduce uses 10-way merging in (9). The extra $\log$ term brings its rate to $n^{2.34}$

TABLE I
DIRECTED GRAPH PROPERTIES

Name	Graph	Nodes n	Edges m	Degree	Size (GB)	Wedges $T_n = \sum_i X_i Y_i$	$\max_i Y_i$	$\max_i X_i$
$\mathcal{D}_1$	ClueWeb-domain	30,558,375	415,167,456	13.6	1.7	4,240,567,641,185	1,034,416	2,360,978
$\mathcal{D}_2$	ClueWeb-host	110,675,107	1,064,508,293	9.6	4.5	1,404,873,157,927	2,326,861	855,063
$\mathcal{D}_3$	ClueWeb-page	2,570,747,470	49,902,497,310	19.4	199.0	2,889,895,321,002	44,410,991	12,571
$\mathcal{D}_4$	IRLbot-domain	86,534,418	1,799,516,827	20.8	7.1	3,073,393,262,407	2,947,630	1,496,324
$\mathcal{D}_5$	IRLbot-host	641,982,060	6,752,615,553	10.5	27.9	2,704,210,948,405	5,475,224	1,333,966
$\mathcal{D}_6$	IRLbot-page	4,051,690,819	238,194,440,791	58.8	916.2	79,864,490,755,128	129,744,852	65,527



(a) $a = 0.4, r = 0.5$ (b) $a = 0.3, r = 1.1$

Fig. 4. Actual I/O c_n on adversarial graphs with curve-fitted scaling rates.

instead of the predicted $n^{2.3}$. DWC$_2$-B is expected to increase I/O as $n^{1.5}$, but n is not large enough to complete the switch to m^2/M asymptotics, so the result comes out slightly better.

V. EXPERIMENTS

In this section, we build an implementation of our techniques and examine its performance on real graphs. Benchmarks are performed on a machine with an 8-core Intel i7-7820X @ 4.7 GHz, 32 GB of DDR4-3000 quad-channel memory, and a 200-TB file system with 24 magnetic hard drives (8 TB Hitachi Ultrastar) driven by two Areca 1882ix controllers in RAID-50. We conduct experiments using two applications from Section II – counting supporters and enumerating 4-cycles.

A. Directed Graphs

Our first evaluation involves the supporter-based ranking problem [27], which counts the number of unique nodes at distance two from each source x along the in-edges, excluding direct in-neighbors. In our earlier notation, this maps to problem (3), where $\pi_z = 1$ and the reducer function is a summation.In the example of Fig. 1(c), we have $f(\mathcal{P}_x, N_x) = 1$ since only node z_1 is a true supporter of x (z_2 and y_1 are direct neighbors). The output of the program is a list of tuples $\{x, f(\mathcal{P}_x, N_x)\}$, ordered by x. Note that other Category-II problems will exhibit similar amounts of I/O in terms of edges, but their runtime may vary depending on the complexity of target function f.

For the supporter problem, we use six directed graphs whose properties are outlined in Table I. The first three are built by parsing HTML from the ClueWeb 2009 crawl [24], producing a 199-GB page-level dataset $\mathcal{D}_3$ with 2.5B nodes, 49B edges, average degree 19.4, and 2.9T wedges. Condensing this graph at the host level (110M nodes) and domain level (30M nodes) produces the first two rows in the table. Even though these are much smaller graphs, the number of wedges in $\mathcal{D}_1$ and $\mathcal{D}_2$ is comparable to that in $\mathcal{D}_3$. This can be explained by their much denser wedge structure.

TABLE II
SUPPORTERS: ACCURACY OF STOCHASTIC MODELS ON $\mathcal{D}_4$

p	DWC$_2$-A (B edges)			DWC$_2$-B (B edges)		
	Actual	(23)	Error	Actual	(24)	Error
1	1.5	1.5	0.00%	1.8	1.8	0.00%
8	10.2	10.2	−0.01%	12.3	12.3	0.00%
64	61.5	61.5	−0.02%	68.8	68.8	0.01%
512	290.3	291.0	0.22%	277.2	277.8	0.25%

The second half of Table I uses the 2007 IRLbot crawl [50]. Because the full graph contains over 41B nodes and thus requires 8-byte labels, we truncate it by dropping low-degree nodes, which allows our code to process all six graphs using 4-byte node IDs. The resulting graph $\mathcal{D}_6$ has 238B edges, 4B nodes, 80T directed wedges, and occupies almost a terabyte. Note that its largest in-degree $\max_i Y_i$ shown in the table (i.e., 129M) constrains type-A partitioning to no less than 520 MB of RAM. Type-B, on the other hand, can process this graph with just 250 KB of memory due to the much smaller maximum out-degree. From $\mathcal{D}_6$, we obtain the condensed domain/host graphs $\mathcal{D}_4 - \mathcal{D}_5$.

B. Scaling Rates of I/O

We begin with evaluating the four baseline algorithms for counting supporters – ACC [27], MapReduce [27], DWC$_1$, and DWC$_2$ – whose models of I/O are well-understood. Specifically, ACC uses (8), MapReduce sorts T_n wedges (z, x) of 8 bytes each in (9), DWC$_1$ employs the result of Theorems 1-2, and DWC$_2$ relies on the closed-form derivation in Theorem 4. Models for the first three methods are exact and do not require accuracy verification. Even though the fourth model provides *expected* cost, i.e., averaged over all assignments of node IDs, its estimates on a single graph instance are usually spot on. This is shown in Table II using DWC$_2$-A/B on $\mathcal{D}_4$, where (23)-(24) indeed produce negligible error. Unlike Theorem 3, which requires creation of all p partitions, results (23)-(24) operate with just the in/out degree sequence. This line of modeling allows a low-overhead comparison between DWC$_2$-A/B, estimation of the total I/O cost, and even prediction of the hash-table size in Theorem 5.

Armed with these models, Table III compares the scaling

TABLE III
SUPPORTERS: MODEL I/O (TB) ON $\mathcal{D}_5$

RAM	ACC	MapReduce	DWC_1	DWC_2-A	DWC_2-B
8 GB	13	157	0.10	0.06	0.08
2 GB	53	197	0.35	0.18	0.23
512 MB	214	197	1.33	0.56	0.64
128 MB	857	235	5.33	1.49	1.64
32 MB	3,430	235	21.25	3.36	3.60

TABLE IV
SUPPORTERS: MODEL I/O (TB) ON $\mathcal{D}_6$

RAM	ACC	MapReduce	DWC_1	DWC_2-A	DWC_2-B
64 GB	150	4,649	13.1	8.1	5.5
16 GB	598	5,811	52.4	20.2	12.5
8 GB	1,168	5,811	101.3	28.2	18.3
4 GB	2,394	6,973	207.4	38.1	27.2
1 GB	9,351	6,973	805.9	60.4	52.2
512 MB	18,415	8,135	1,588.6	72.1	66.5

rate of the four methods using $\mathcal{D}_5$. ACC begins at a reasonable 13 TB in the first row, but then skyrockets all the way to 3.4 PB as memory pressure takes its toll. MapReduce, which uses Hadoop's default 10-way merge and requires sorting 19.7 TB of data, behaves the opposite. Taking into account its multi-pass merge cost, it starts at an astronomical 157 TB at $M = 8$ GB, but then scales much slower than ACC, finishing at only 235 TB. Another concern with MapReduce is the hefty CPU effort needed to sort and merge this amount of data, as well as the non-trivial HDD space to store intermediate files.

DWC_1, where we omit specifying A/B as both version post essentially the same numbers, is quite efficient with $M = 8$ GB in the first row of the table, completing the task with just 100 GB of read-only traffic, i.e., 2-4 orders of magnitude better than ACC/MapReduce. While DWC_2-A/B beat this number, their advantage over DWC_1 becomes more clear as RAM size shrinks. The gap between the two versions grows as we progress down the table, where DWC_2 eventually ends up with $\sim$3.5 TB of I/O in the bottom row, which is 6$\times$ lower than DWC_1. The final observation is that between the two DWC_2 options, method A wins by 5-33% on this graph.

Moving on to our largest graph $\mathcal{D}_6$, Table IV shows that ACC now begins at 150 TB and reaches a whopping 18 PB in the last row. MapReduce needs to sort 581 TB of wedges, which exceeds our file system size, and requires over 4.6 TB of I/O even for the largest memory size. In the next column, DWC_1 again exhibits poor scaling characteristics, starting off with a decent 13 TB, but then finishing with an enormous 1.6 PB, the latter of which is 22$\times$ less efficient than DWC_2 in the next two columns. As $M \to 0$, both DWC_2 methods perform quite well, increasing I/O from 5-8 TB to around 70, which corresponds to a sublinear trend $1/M^{0.6}$ rather than inverse linear $1/M$ inherent to ACC and DWC_1. Also note that the relationship between DWC_2-A/B is now flipped, where B is up to 60% better in certain rows.

TABLE V
SUPPORTERS: ACTUAL I/O (TB) WITH 8 GB RAM

Method	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
Hadoop	564	155	338	328	315	9,399
STXXL	237	75	159	170	149	4,974
GraphChi	47	17	29	31	27	808
Rstream	62	21	44	45	41	1,165
DWC_2-A	0.002	0.004	2.5	0.007	0.06	28
DWC_2-B	0.002	0.004	1.9	0.007	0.08	18

C. Actual I/O and Runtime

Since ACC appears infeasible in most non-trivial cases, we no longer consider it in our evaluation. We also tested general-purpose databases by formulating a self-join on the graph; however, this produced excruciatingly slow results. For example, a 0.01% subsample of our smallest graph $\mathcal{D}_1$ took 1,181 seconds in MySQL. We therefore do not consider databases as a viable solution to this problem. We further dismiss DWC_1 in favor of DWC_2 and replace the generic MapReduce concept with four alternative implementations that have similar asymptotic I/O complexity – Hadoop [10], STXXL [31], GraphChi [47], and Rstream [70]. The main difference between these methods lies in the efficiency of the merger/reducer within each framework. We next briefly review their features and capabilities.

Hadoop is a widely used Java implementation of Google MapReduce [30], while STXXL is a highly optimized C++ platform for various external-memory computation, including sorting. GraphChi is a C++ representative of the family of libraries in which weights are iteratively passed along the edges between immediate neighbors [22], [55], [56], [76]. This model of computation works well with a scalar reducer that shrinks multiple arriving weights into one (e.g., PageRank); however, it incurs significant I/O cost when the weights must be accumulated into growing vectors (e.g., all supporters of a given node) whose size potentially exceeds RAM. Because the latest generation of GraphChi allows storing edge weights to disk, it can successfully operate on Category-II wedge problems. Finally, Rstream comes from a related branch of literature – graph-mining systems [58], [67], [70], [72] – that perform an inner join of (z, y) with (y, x) to produce wedges (z, y, x). To construct $\mathcal{P}_{zx}$, all rows in the self-join result are sorted and aggregated on x.

Note that prior work generates T_n wedges to disk in the worst case, where the cost of sorting or doing a self-join is given by (9). The main difference lies in the s-way merge/distribution factor and possible removal of duplicate pairs during the intermediate steps. For example, Hadoop uses $s = 10$ by default, STXXL adjusts s based on RAM and input size, and GraphChi uses $s = T_n/M$ to create enough partitions to finish in one distribution pass. It further compacts the edges before writing them to disk. These nuances explain why the actual I/O and runtime vary between prior methods.

Table V shows the I/O cost of all six methods in TBs and Table VI displays the corresponding runtime in days. Note that experiments that could not finish within three weeks

TABLE VI
SUPPORTERS: RUNTIME (DAYS) WITH 8 GB RAM

Method	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$
Hadoop	115	34	77	83	72	2,437
STXXL	41.4	11.6	26.7	28.7	24.8	1,093
GraphChi	16.6	4.8	9.4	9.9	8.8	259
Rstream	16.5	4.9	10.2	10.8	9.5	281
DWC_2-A	0.20	0.17	0.24	0.28	0.20	2.2
DWC_2-B	0.08	0.07	0.15	0.11	0.09	1.3

TABLE VII
PREPROCESSING TIME WITH 8 GB RAM

	Partition		Inversion	
	DWC_2-A	DWC_2-B	DWC_2-A	DWC_2-B
$\mathcal{D}_3$	28 min	20 min	2.9 hrs	1.9 hrs
$\mathcal{D}_5$	87 sec	75 sec	4.3 min	5.4 min
$\mathcal{D}_6$	4.2 hrs	3.6 hrs	35.8 hrs	21.7 hrs

are marked with gray background and their numbers are extrapolated based on the remaining fraction of T_n that was still left unprocessed. In the first row, Hadoop is unable to produce results on any of the graphs, requiring an estimated 564 TB on the smallest dataset $\mathcal{D}_1$ and 9.3 PB on the largest. Its predicted runtime is also exceedingly slow – almost 4 months on $\mathcal{D}_1$ and 6.7 years on $\mathcal{D}_6$. STXXL runs 2-3× faster and exhibits success on $\mathcal{D}_2$, which it finishes in roughly 11 days, but its performance on the remaining datasets (i.e., 1-36 months) leaves much to be desired. GraphChi and Rstream are tied for top place, beating Hadoop and STXXL by 6-12× in I/O and 3-8× in runtime. Despite the win, they require over two weeks on $\mathcal{D}_1$ and neither is capable of handing $\mathcal{D}_6$ in less than an estimated 8-9 months and close to a PB of I/O.

We now turn attention to the performance of proposed methods in the bottom two rows of Tables V-VI. Our C++ implementation performs a full count of unique supporters z, excludes immediate neighbors of x, and saves the resulting tuples $\{x, f(\mathcal{P}_x, N_x)\}$ to disk. Results from multiple partitions are aggregated into a final counter for each node. When the graph fits in RAM (i.e, cases $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_4$), DWC_2 yields astronomically lower I/O since it does not generate any auxiliary files and its runtime is 70-200× better than the fastest techniques in prior work. On medium-size graphs that are 3-25× larger than RAM (i.e., $\mathcal{D}_3$, $\mathcal{D}_5$), the advantage is 23-270× in terms of I/O and 62-88× in terms of runtime. Finally, the largest graph, which exceeds RAM by 114×, yields an estimated improvement by 42× and 199×, respectively.

As predicted earlier, dispersing counters to many random locations x in Algorithm 3 has a noticeable negative impact on the CPU-related runtime. As a result, DWC_2-B performs 1.6-2.8× faster than DWC_2-A. It also incurs less I/O in cases where it matters – 35% on $\mathcal{D}_3$ and 56% on $\mathcal{D}_6$. Additionally considering its lower minimum RAM constraint, DWC_2-B emerges as a safe default choice for Category-II wedge problems in directed graphs.

Overall, the outcome for solving supporter-style problems using DWC_2-B is quite encouraging – graphs with a few trillion wedges require 1-2 hours using an 8-core desktop CPU, while those approaching 100 trillion take about a day, *even on input 100× larger than RAM*. While this is not as efficient as triangle counting, which takes 148 seconds on IRLbot-domain [28], there are mitigating factors – wedge-based computing over directed graphs does not have the luxury of applying custom acyclic orientations to undirected graphs to reduce complexity, running SIMD neighbor intersection, or discarding candidate supporters z that are not direct neighbors of x.

To complete the picture, Table VII reports preprocessing delays that apply to the three graphs larger than RAM. This result again demonstrates an advantage for DWC_2-B, which is 1.5× faster on inverting ClueWeb-page and 1.6× on IRLbot-page. While these delays are comparable to the total runtime, preprocessing is considered an offline operation that is performed once. In contrast, evaluation of various functions f over the graph may be executed multiple times.

D. Undirected Graphs

The second application we consider in this paper is counting non-induced 4-cycles in G. For this purpose, we use the six undirected graphs in Table VIII. The first five, obtained from the Network Repository [1], are relatively small, with sizes ranging from 17 MB to 110 MB; however, because related work cannot handle anything much larger, these graphs are common benchmarks in the field of quadrangle enumeration. The last graph $\mathcal{U}_6$ comes from [46] and contains 41M nodes, 1.2B edges, and 502T quadrangles. Even though it is much smaller than the largest graphs in Table I, its undirected nature yields an enormous wedge count $T_n = \sum_i d_i^2 = 246$T, requiring 3× more CPU work than $\mathcal{D}_6$.

Our comparison includes the four methods from the supporter problem and two new frameworks – EMRC [78] and Multiway Join MapReduce (MJMR) [3], [4]. The former is a state-of-the-art solution for counting 4-cycles in external memory, while the latter focuses on speeding up MapReduce joins during the search for certain families of subgraphs, including triangles and quadrangles. Since neither approach provides an implementation, we develop our own using C++. It should be noted that EMRC's in-memory search component WCRC uses a hash table to keep a counter for each unique wedge pair (z, x), which leads to a massive data-amplification problem. With our notation from (8), the hash table size scales proportional to the number of supporters, i.e., $\sum_i Q_i$. For example, WCRC implemented using the C++ *unordered_map* exceeds 24 GB on graph $\mathcal{U}_1$ and runs out of 32 GB memory on $\mathcal{U}_2$. As confirmed by the analysis in [78], EMRC subgraphs with x edges require in-RAM space complexity $O(x^{1.5})$, which lowers the size of each subgraph from M to $M^{2/3}$ and leads to $O(m^2/M^{2/3})$ asymptotic I/O complexity. This is already worse than our introductory method DWC_1.

Applying DWC to quadrangle detection requires operation on undirected graphs. This can be done by replacing G^+ and G^- in Algorithms 1-6 with the undirected version G. Note that DWC can obtain 4-cycle counts per node, as well as produce a list of participants in each of them; however, a common benchmark in previous work is to output the total

TABLE VIII
UNDIRECTED GRAPH PROPERTIES

Name	Graph	Nodes n	Edges m	Degree	Size (MB)	Wedges $T_n = \sum_i d_i^2$	Quadrangle count	$\max_i d_i$
$\mathcal{U}_1$	Web-Google	875,713	4,322,051	9.9	39.7	1,463,478,550	539,575,204	6,332
$\mathcal{U}_2$	Web-Stanford	281,903	1,992,636	14.1	17.4	7,892,123,458	13,316,840,570	38,625
$\mathcal{U}_3$	WikiTalk	2,394,385	4,659,565	3.9	53.8	25,196,363,974	2,152,013,141	100,029
$\mathcal{U}_4$	Com-Youtube	1,134,890	2,987,624	5.3	31.5	2,954,940,368	468,774,021	28,754
$\mathcal{U}_5$	Bio-mouse	43,126	14,464,096	642.3	110.8	51,645,960,052	4,357,162,782,838	8,032
$\mathcal{U}_6$	Twitter	41,652,230	1,202,513,046	57.7	9,492	246,873,584,440,346	502,590,430,301,880	2,997,487

TABLE IX
QUADRANGLES: ACTUAL I/O (GB) WITH 8 GB RAM

Method	$\mathcal{U}_1$	$\mathcal{U}_2$	$\mathcal{U}_3$	$\mathcal{U}_4$	$\mathcal{U}_5$	$\mathcal{U}_6$
Hadoop	23.3	139	705	61.4	1,249	17M
STXXL	14.4	73.3	405	12.7	753	7M
GraphChi	0.5	53.6	326	9.9	98	2M
Rstream	8.4	37.9	151	18.5	277	1M
EMRC	0.2	0.13	1.8	0.3	0.33	23K
MJMR	2.0	0.9	2.2	1.4	6.75	–
DWC_2	0.04	0.02	0.05	0.03	0.11	28

TABLE X
QUADRANGLES: RUNTIME WITH 8 GB RAM

Method	$\mathcal{U}_1$	$\mathcal{U}_2$	$\mathcal{U}_3$	$\mathcal{U}_4$	$\mathcal{U}_5$	$\mathcal{U}_6$
Hadoop	180s	2,029s	3.45h	841s	4.81h	7.5y
STXXL	45s	274s	0.31h	108s	0.57h	234d
GraphChi	39s	1,515s	2.67h	369s	1.31h	924d
Rstream	227s	887s	0.98h	486s	1.16h	772d
EMRC	285s	772s	2.06h	725s	1.06h	762d
MJMR	43s	190s	0.44h	141s	17.6h	–
DWC_2-B	0.58s	1.01s	18.8s	1.08s	3.58s	6.8d

number of quadrangles in the entire graph. In such cases, it is wasteful to visit each motif four times, which can be avoided by considering only wedge collections P_{zx} with $z < x$ and wedges P_{zyx} such that $z < y$, which we also do in EMRC and MJMR.

To obtain the number of wedges in each set $\mathcal{P}_{zx}$, DWC walks from x to discover all supporters z and increment their hit counters using Algorithms 3 and 4 (Lines 7-9). After x is fully processed, we revisit each supporter z, retrieve its counter c, and increment a global sum by $\binom{c}{2}$, resetting all modified counters back to zero. When dealing with multiple partitions, 1D decomposition of the 4-cycle problem guarantees that individual partition-specific counters can be added to each other to produce the final value. For this algorithm, counter space is limited to a small array of size $|V_r|$ for each subgraph r, which is usually orders of magnitude less than in EMRC. Furthermore, with $|V_r|$ greatly smaller than subgraph size, we can multi-thread the algorithm by assigning a separate counter array to each thread, eliminating the need for synchronization.

We next examine the actual cost to perform enumeration of all 4-cycles in the six graphs of Table VIII. In this comparison, we set the memory limit M to 8 GB and do not differentiate between DWC_2-A and DWC_2-B because their I/O is identical when operating on undirected graphs. We present the I/O result in Table IX, again marking extrapolated values with a shaded background. Although the first five graphs $\mathcal{U}_1 - \mathcal{U}_5$ are smaller than RAM, all methods in related work produce non-trivial amounts of I/O. This can be explained by their disk workload being a function of either T_n or the number of supporters in each subgraph. Thus, Hadoop begins with 23 GB of I/O on $\mathcal{U}_1$, which snowballs to 1.3 TB by the time we get to $\mathcal{U}_5$. STXXL suffers a 50× increase between these datasets and GraphChi 200×. While EMRC comes out ahead of the other methods in previous work, it still requires between 2 and 16 partitions due to the high hash-table space cost.

When dealing with $\mathcal{U}_6$, however, the situation gets worse. The first four methods in the table produce an estimation I/O that ranges from 1 to 17 PB. While EMRC is still decidedly better, it nevertheless needs 815 partitions and 23 TB of I/O. It should be noted that MJMR is second-best on the small graphs (i.e., within a factor of 1.2-20 of EMRC), but its extrapolation model is fairly complex, which explains why it is left blank in the table. DWC_2, on the other hand, can finish $\mathcal{U}_1 - \mathcal{U}_5$ without partitioning and $\mathcal{U}_6$ with just 28 GB of I/O. This is almost 1000× less than the closest competitor.

Table X shows the corresponding runtime. On the first five graphs, STXXL is frequently the fastest among prior work, topping out at 0.57 hours on $\mathcal{U}_5$. Even though EMRC has the lowest I/O among these methods, its poor RAM usage leaves space for only one hash table that occupies almost the entire 8 GB. As a result, EMRC runs faster when using a single thread, while usage of multiple threads leads to performance degradation due to the extra synchronization cost. MJMR is close to STXXL on $\mathcal{U}_1 - \mathcal{U}_4$, but its CPU efficiency sharply deteriorates on graphs with larger quadrangle counts, resulting in an enormous 17.6 hours on $\mathcal{U}_5$. Extrapolating the runtime to $\mathcal{U}_6$ leads to an estimated 7.5 years for Hadoop, 7.8 months for STXXL, and over 2 years for the next three methods.

For better cache locality, we use DWC_2-B in Table X, where it posts results that are 34-570× faster than the best alternatives. The largest speedup is achieved on our densest graph $\mathcal{U}_5$, where DWC_2-B delivers the output in just 3.58 seconds. One reason for this performance is Algorithm 4 that finishes full wedge discovery in the neighborhood of each node x before moving elsewhere, which leads to tiny hash tables that fit into L2/L3 caches and effortless parallelization. Additionally, by minimizing the amount of I/O, the number of hits against the hash table in Theorem 5 is also kept to a minimum, resulting in faster runtime. In the final graph $\mathcal{U}_6$, DWC_2-B has no trouble finishing the job, enumerating all 502T quadrangles in 164 hours.

VI. Conclusion

We created a novel taxonomy of wedge-based computation in external memory for a wide range of graph-analytics applications. Under this umbrella, we identified three distinct categories of algorithms based on the type of decomposition they allowed and proposed an I/O-efficient solution for the medium-complexity class, significantly improving previous techniques in this area. While this type of computation is rarely attempted in external memory, experiments show that our approach is feasible even on graphs that exceed RAM size by two orders of magnitude and contain hundreds of trillions of wedges, without requiring exorbitant cluster resources.

References

[1] "Network Repository," Oct. 2024. [Online]. Available: https://networkrepository.com/.

[2] L. A. Adamic and E. Adar, "Friends and Neighbors on the Web," *Social Networks*, vol. 25, no. 3, pp. 211–230, 2003.

[3] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating Subgraph Instances Using Map-Reduce," in *Proc. IEEE ICDE*, Apr. 2013, pp. 62–73.

[4] F. N. Afrati and J. D. Ullman, "Optimizing Multiway Joins in a Map-Reduce Environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1282–1298, Feb. 2011.

[5] A. Aggarwal and J. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *CACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.

[6] N. Ahmed, J. Neville, R. Rossi, and N. Duffield, "Efficient Graphlet Counting for Large Networks," in *Proc. IEEE ICDM*, Nov. 2015.

[7] N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, March 1997.

[8] T. Anastasakos, D. Hillard, S. Kshetramade, and H. Raghavan, "A Collaborative Filtering Approach to Ad Recommendation using the Query-ad Click Graph," in *Proc. ACM CIKM*, 2009, pp. 1927–1930.

[9] I. Antonellis, H. G. Molina, and C. C. Chang, "Simrank++: Query Rewriting through Link Analysis of the Click Graph," *PVLDB*, vol. 1, no. 1, pp. 408–421, 2008.

[10] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/.

[11] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication," *ACM Transactions on Parallel Computing*, vol. 3, no. 3, pp. 18:1–18:34, Dec. 2016.

[12] L. Becchetti, C. Castillo, D. Donato, S. Leonardi, and R. Baeza-Yates, "Link-based Characterization and Detection of Web Spam," in *Proc. AIRWeb*, Aug. 2006.

[13] L. Becchetti, C. Castillo, D. Donato, S. Leonardi, and R. Baeza-Yates, "Using Rank Propagation and Probabilistic Counting for Link-Based Spam Detection," in *Proc. WebKDD*, Aug. 2006.

[14] J. L. Bentley, "Decomposable searching problems," *Information Processing Letters*, vol. 8, no. 5, pp. 244–251, Jun. 1979.

[15] P. Boldi, F. Bonchi, C. Castillo, D. Donato, and S. Vigna, "Query Suggestions using Query-Flow Graphs," in *Proc. WSCD*, 2009, pp. 56–63.

[16] P. Boldi, M. Rosa, and S. Vigna, "HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget," in *Proc. WWW*, Mar. 2011, pp. 625–634.

[17] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Proc. WWW*, Apr. 1998, pp. 107–117.

[18] A. Buluç and J. R. Gilbert, "Highly Parallel Sparse Matrix-Matrix Multiplication," *CoRR*, vol. abs/1006.2183, Jun. 2010. [Online]. Available: http://arxiv.org/abs/1006.2183.

[19] C. Castillo, D. Donato, A. Gionis, V. Murdock, and F. Silvestri, "Know Your Neighbors: Web Spam Detection Using the Web Topology," in *Proc. SIGIR*, Jul. 2007.

[20] Y.-Y. Chen, Q. Gan, and T. Suel, "I/O-Efficient Techniques for Computing PageRank," in *Proc. ACM CIKM*, Nov. 2002, pp. 549–557.

[21] T. Cheng, H. W. Lauw, and S. Paparizos, "Entity Synonyms for Structured Web Search," *IEEE TKDE*, vol. 24, no. 10, pp. 1862–1875, 2011.

[22] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale," in *Proc. VLDB*, Aug. 2015, pp. 1804–1815.

[23] S. Chu and J. Cheng, "Triangle Listing in Massive Networks and Its Applications," in *Proc. ACM SIGKDD*, Aug. 2011, pp. 672–680.

[24] ClueWeb09 Dataset. [Online]. Available: http://www.lemurproject.org/clueweb09/.

[25] J. Cohen, "Graph twiddling in a MapReduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[26] P. Crescenzi, R. Grossi, L. Lanzi, and A. Marino, "A Comparison of Three Algorithms for Approximating the Distance Distribution in Real-world Graphs," in *Proc. TAPAS*, Apr. 2011, pp. 92–103.

[27] Y. Cui, C. Sparkman, H.-T. Lee, and D. Loguinov, "Unsupervised Domain Ranking in Large-Scale Web Crawls," *ACM Trans. Web*, vol. 12, no. 4, pp. 26:1–26:29, Nov. 2018.

[28] Y. Cui, D. Xiao, and D. Loguinov, "On Efficient External-Memory Triangle Listing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 8, pp. 1555–1568, Aug. 2019.

[29] Y. Cui, D. Xiao, D. B. Cline, and D. Loguinov, "Improving I/O Complexity of Triangle Enumeration," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 4, pp. 1815–1828, Apr. 2022.

[30] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.

[31] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard Template Library for XXL Data Sets," *Software: Practice and Experience*, vol. 38, no. 6, pp. 589–637, May 2008.

[32] H. Deng, M. R. Lyu, and I. King, "A Generalized Co-hits Algorithm and its Application to Bipartite Graphs," in *Proc. ACM SIGKDD*, 2009, pp. 239–248.

[33] C. Ding, X. He, P. Husbands, H. Zha, and H. Simon, "PageRank, HITS and a Unified Framework for Link Analysis," in *Proc. SDM*, 2003, pp. 249–253.

[34] A. Epasto, J. Feldman, S. Lattanzi, S. Leonardi, and V. Mirrokni, "Reduce and Aggregate: Similarity Ranking in Multi-categorical Bipartite Graphs," in *Proc. WWW*, 2014, pp. 349–360.

[35] D. Fogaras, B. Racz, K. Csalogany, and T. Sarlos, "Towards Scaling Fully Personalized PageRank: Algorithms, Lower Bounds, and Experiments," *Internet Mathematics*, vol. 2, no. 3, pp. 333–358, Nov. 2005.

[36] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin, "Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs," *PVLDB*, vol. 7, no. 13, pp. 1379–1380, Aug. 2014.

[37] Z. Gyöngyi and H. Garcia-Molina, "Web Spam Taxonomy," in *Proc. AIRWeb*, May 2005, pp. 39–47.

[38] S. Haas, F. Wilkens, and M. Fischer, "Efficient Attack Correlation and Identification of Attack Scenarios based on Network-Motifs," in *Proc. IEEE IPCCC*, Oct. 2019, pp. 1–11.

[39] T. Haveliwala, "Efficient Computation of PageRank," Stanford University, Tech. Rep., Oct. 1999.

[40] T. H. Haveliwala, "Topic-sensitive Pagerank," in *Proc. WWW*, 2002, pp. 517–526.

[41] X. Hu, Y. Tao, and C. Chung, "Massive Graph Triangulation," in *Proc. ACM SIGMOD*, Jun. 2013, pp. 325–336.

[42] G. Jeh and J. Widom, "SimRank: A Measure of Structural-Context Similarity," in *Proc. ACM SIGKDD*, Jul. 2002.

[43] Z. R. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: A New Algorithm for Finding Network Motifs," *Bioinformatics*, vol. 10, no. 318, Oct. 2009.

[44] L. Katz, "A New Status Index Derived from Sociometric Analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[45] J. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *J. of the ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999.

[46] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. WWW*, Apr. 2010, pp. 591–600.

[47] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proc. USENIX OSDI*, 2012, pp. 31–46.

[48] M. Latapy, "Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs," *Elsevier Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, Nov. 2008.

[49] A. Lazzaro, J. VandeVondele, J. Hutter, and O. Schütt, "Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI," in *Proc. ACM PASC*, Jun. 2017.

[50] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," *ACM Trans. Web*, vol. 3, no. 3, pp. 1–34, Jun. 2009.

[51] Z. Lin, M. R. Lyu, and I. King, "MatchSim: a Novel Similarity Measure based on Maximum Neighborhood Matching," *KAIS*, vol. 32, no. 1, pp. 141–166, 2012.

[52] H. Liu and H. H. Huang, "Graphene: Fine-Grained IO Management for Graph Computing," in *Proc. USENIX FAST*, Feb. 2017, pp. 285–299.

[53] F. Long, X. Wang, and Y. Cai, "API Hyperlinking via Structural Overlap," in *Proc. ACM ESEC-FSE*, 2009, pp. 203–212.

[54] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *Proc. VLDB*, Aug. 2012, pp. 716–727.

[55] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "MOSAIC: Processing a Trillion-Edge Graph on a Single Machine," in *Proc. ACM EuroSys*, Apr. 2017, pp. 527–543.

[56] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. ACM SIGMOD*, Jun. 2010, pp. 135–145.

[57] D. Marcus and Y. Shavitt, "RAGE – A rapid graphlet enumerator for large networks," *Computer Networks*, vol. 56, pp. 810–819, Feb. 2012.

[58] D. Mawhirter and B. Wu, "AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining," in *Proc. ACM SOSP*, 2019, pp. 509–523.

[59] Q. Mei, D. Zhou, and K. Church, "Query suggestion using hitting time," in *Proc. ACM CIKM*, 2008, pp. 469–478.

[60] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, Oct. 2002.

[61] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on Intel KNL and multicore architectures," in *Proc. IEEE ICPP*, Aug. 2018, pp. 34:1–34:10.

[62] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project, Tech. Rep., Jan. 1998. [Online]. Available: http://dbpubs.stanford.edu:8090/pub/1999-66.

[63] R. Pagh and F. Silvestri, "The Input/Output Complexity of Triangle Enumeration," in *Proc. ACM PODS*, Jun. 2014, pp. 224–233.

[64] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, "ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs," in *Proc. ACM SIGKDD*, Jul. 2002, pp. 81–90.

[65] G. Stella and D. Loguinov, "On High-Latency Bowtie Data Streaming," in *Proc. IEEE BigData*, Dec. 2022, pp. 75–84.

[66] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *Proc. WWW*, Mar. 2011, pp. 607–614.

[67] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A System for Distributed Graph Mining," in *Proc. ACM SOSP*, 2015, pp. 425–440.

[68] N. H. Tran, K. P. Choi, and L. Zhang, "Counting motifs in the human interactome," *Nature Communications*, vol. 4, p. 2241, Aug. 2013.

[69] J. Ugander, L. Backstrom, and J. Kleinberg, "Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections," in *Proc. WWW*, Apr. 2013, pp. 1307–1318.

[70] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine," in *Proc. USENIX OSDI*, 2018, pp. 763–782.

[71] D. Xiao, Y. Cui, D. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," in *Proc. ACM PODS*, May 2017, pp. 261–272.

[72] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. Lui, "G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph," in *Proc. IEEE ICDE*, 2020, pp. 1369–1380.

[73] W. Yu and J. A. McCann, "High Quality Graph-based Similarity Search," in *Proc. ACM SIGIR*, 2015, pp. 83–92.

[74] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," in *Proc. ESA*, Sep. 2004, pp. 604–615.

[75] P. Zhao, J. Han, and Y. Sun, "P-Rank: a Comprehensive Structural Similarity Measure over Information Networks," in *Proc. ACM CIKM*, 2009, pp. 553–562.

[76] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs," in *Proc. USENIX FAST*, Feb. 2015, pp. 45–58.

[77] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Semi-External Memory Sparse Matrix Multiplication for Billion-Node Graphs," *IEEE TPDS*, vol. 28, no. 5, pp. 1470–1483, May 2017.

[78] R. Zhu, Z. Zou, and J. Li, "Fast rectangle counting on massive networks," in *Proc. IEEE ICDM*, 2018, pp. 847–856.

[79] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," in *Proc. USENIX ATC*, Jul. 2015, pp. 375–386.