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

Many applications perform analysis, data mining, and information extraction on huge graphs that may be orders of magnitude larger than RAM. 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 [21], which iteratively passes weights among neighbors to determine the stationary distribution of a random walker, and construction of a search index [6], 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 [16], GraphLab [18], and MapReduce [13].

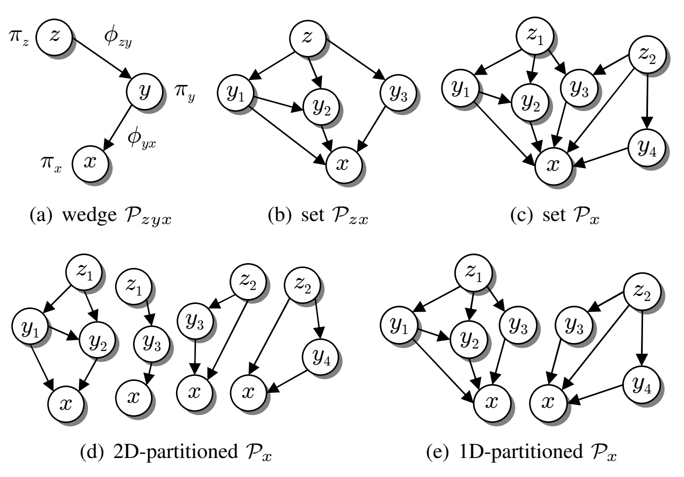
The second scenario walks to distance two from each node, examining the various *wedges* (i.e., two-node paths) in the second neighborhood of the source. This category, which we label *wedge-based*, includes triangle enumeration [8], [11], [12], similarity ranking [4], [15], [27], four-node motif discovery [2], spam avoidance in web crawling [10], SpGEMM sparse matrix-matrix multiplication [20], and various related problems [5], [22]. 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 [12].

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 [16], [18], and MapReduce [13] 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 much more efficiently. In particular, we derive a precise closed-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.

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



(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.

graph fits in memory [2], [7], [19] or that the problem can be reduced to scans over the adjacency lists of the graph [16], [18]. 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.

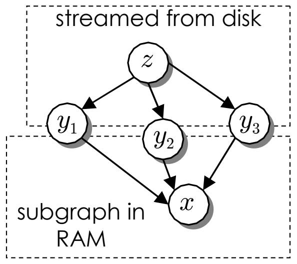
B. Taxonomy

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* 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.

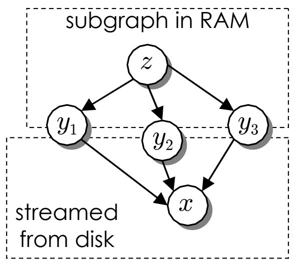
Let n be the number of nodes in the graph, m be the corresponding number of edges, T_n be the number of wedges, and M be RAM size. We say that function f belongs to *Category I* if it can process the wedges in $\mathcal{P}_x$ independently of each other. The main advantage of this type of problems is 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 [12]. 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. In the worst case, this requires sorting all T_n tuples in external memory at 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 [4], [15], spam avoidance [10], and sparse matrix multiplication [20] 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 [2], 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



(a) Method A ($x \in V_r$)



(b) Method B ($z \in V_r$)

Fig. 2. Partitioning options.

detection of the corresponding motif. Efficient I/O in such cases is beyond the scope of this paper.

III. Decomposable Wedge Computing (DWC)

This section introduces our approach for solving Category-II wedge problems using external memory. The proofs omitted from the paper, a more in-depth theoretical analysis, and additional discussion/experiments can be found in [25].

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 [1], 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 [23].

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. 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.

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 [11], 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.

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

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

and in Method B using

$$\sum_{i=a_r}^{a_{r+1}-1} X_i = M. \tag{2}$$

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 1 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 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

Algorithm 1: Wedge computing under type-A partitioning

```
for $r = 1$ to $p$ do ◁ iterate over all partitions
    $v$ = array of $|V_r|$ zeros                ◁ initialize output values
    load $G_r^+ = \{(y, N_y^+(r))\}$ into RAM
    $H$ = hash table built from $G_r^+$     ◁ maps nodes $y$ to out-lists
    while !eof($G^+$) do
        load next neighbor list $(z, N_z^+)$ from $G^+$
        for all $x$ reachable in two hops from $z$ using $H$ do
            construct $(\mathcal{P}_{zx}, N_x^-)$ from $(z, N_z^+, H)$
            $v[x - a_r] = h(v[x - a_r], f(\mathcal{P}_{zx}, N_x^-))$   ◁ reducer
    for all $v[i] \neq 0$ do
        append $(i + a_r, v[i])$ to file $F_{out}$
    set file pointer in $G^+$ to the start
```

Algorithm 2: Wedge computing under type-B partitioning

```
for $r = 1$ to $p$ do ◁ iterate over all partitions
    load $G_r^- = \{(y, N_y^-(r))\}$ into RAM
    $H$ = hash table built from $G_r^-$     ◁ maps nodes $y$ to in-lists
    while !eof($G^-$) do
        load next neighbor list $(x, N_x^-)$ from $G^-$
        $v = 0$                          ◁ initialize output value for $x$
        for all $z$ reachable in two hops from $x$ using $H$ do
            construct $\mathcal{P}_{zx}$ from $(x, N_x^-, H)$
            $v = h(v, f(\mathcal{P}_{zx}, N_x^-))$                ◁ reducer
        if $v \neq 0$ then
            append $(x, v)$ to file $F_{out}(r)$
    set file pointer in $G^-$ to the start
```

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 (1) 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

$$\sum_{i=a_r}^{a_{r+1}-1} (Y_i + k_i) = M, \tag{3}$$

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 2, 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 (2) is still correct, which means that I/O performance of Algorithm 2 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 (3) 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*

$$c_n(1A) = \frac{m^2}{M}\Big(1 + \frac{E[k_i]}{E[X_i]}\Big), \tag{4}$$

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

$$c_n(1B) = \frac{m^2}{M}, \tag{5}$$

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

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*

$$\tau(p) \leq mE[k_i] \min\Big(1, \frac{n}{M}\Big). \tag{6}$$

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 2, which satisfies the decomposability conditions introduced earlier.

IV. 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.

A. Directed Graphs

Our first evaluation involves the supporter-based ranking problem [10], which counts the number of unique nodes at distance two from each source x along the in-edges, excluding direct in-neighbors. 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

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

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 [9], 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.

The second half of Table I uses the 2007 IRLbot crawl [17]. 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. Actual I/O and Runtime

We first 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 [3], STXXL [14], GraphChi [16], and Rstream [24]. 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 [13], 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 [7], [19], [28]. 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 [24], [26] – 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.

TABLE II
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

Note that prior work generates T_n wedges to disk in the worst case, which are then sorted using $T_n \log(T_n)$ I/O complexity. 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 II shows the I/O cost of all six methods in TBs and Table III displays the corresponding runtime in days. Note that experiments that could not finish within three weeks 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

TABLE III
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

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 II-III. 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 1 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*.

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

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

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

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

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] D. Xiao, Y. Ciu, and D. Loguinov, "Towards Faster External-Memory Graph Computing on Small Neighborhoods," Texas AM University, Tech. Rep., Nov. 2025. [Online]. Available: http://irl.cs.tamu.edu/publications.

[26] 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.

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

[28] 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.