Oasis: An Out-of-core Approximate Graph System via All-Distances Sketches

Tsun-Yu Yang, *The Chinese University of Hong Kong* (*CUHK*)*;* Yi Li, *The University of Texas at Dallas;* Yizou Chen, *The Chinese University of Hong Kong* (*CUHK*)*;* Bingzhe Li, *The University of Texas at Dallas;* Ming-Chang Yang, *The Chinese University of Hong Kong* (*CUHK*)

https://www.usenix.org/conference/fast25/presentation/yang

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Oasis: An Out-of-core Approximate Graph System via All-Distances Sketches

Tsun-Yu Yang[†], Yi Li[‡], Yizou Chen[†], Bingzhe Li[‡], and Ming-Chang Yang[†]
[†]*The Chinese University of Hong Kong*
[‡]*The University of Texas at Dallas*

Abstract

The All-Distances Sketch (ADS) is a powerful and theoretically-sound sketching scheme that captures neighborhood information in graphs for approximate processing. It enables high-accuracy estimation of many useful applications with a guarantee of accuracy and can significantly accelerate the execution times by orders of magnitude. However, ADS requires a substantial amount of space that is multiple times larger than the graph data. More seriously, existing studies mainly focus on managing ADSs in memory, posing an increasing challenge for users who aim to leverage ADS for large-scale graph processing, particularly in light of the exponential growth of real-world graphs nowadays.

To this end, this paper introduces Oasis, an Out-of-core Approximate graph SYStem that brings the ADS technique into practical use by leveraging storage effectively. Specifically, Oasis offers a holistic framework that facilitates both ADS construction and estimation. For ADS construction, it allows users to adjust the memory usage based on the machine's available memory and enable an efficient construction process. For ADS estimation, Oasis provides a user-friendly interface to easily execute the estimators while mitigating the impact of slow storage I/O. Evaluation results show that Oasis provides a practical graph processing solution with exceptional execution time and low memory usage, at the cost of a slight decrease in accuracy.

1 Introduction

Graphs are a powerful data structure that can express a wide range of real-world relationships by storing entities as vertices and connections between entities as edges. The most distinguishing feature of graphs, in comparison to other data structures, is the concept of "neighborhood". Namely, how a vertex is connected to other vertices in a graph is pivotal information that has been studied in many graph theories and applied in diverse applications. For instance, neighborhood information among vertices is critical in applications like social network analysis [33], where understanding relationships between users can help identify influencers or clusters of like-minded individuals, and in recommendation systems [27], where finding similar items or users hinges on shared connections. However, as graph sizes have grown exponentially in recent years, the execution time for gathering neighborhood information with graph traversals has increased correspondingly. In scenarios such as identifying influencers in social networks, it often needs to perform graph traversals for numerous vertices, or sometimes, even for all vertices. This generally makes it unfeasible to complete these queries within a reasonable timeframe.

In this regard, neighborhood sketching provides an effective alternative to traditional approaches for handling neighborhood-related queries on large-scale graphs [1, 4, 10, 26]. This technique leverages approximate computation to achieve fast execution times. Essentially, in many real-world applications, exact answers are not always necessary [19], which offers the opportunity to trade answer accuracy for a significant reduction in execution time. Meanwhile, a neighborhood sketch is a data structure that stores some pre-specified and partial neighborhood information. It allows users to leverage approximate computations on this partial neighborhood information to efficiently estimate various useful graph insights and properties. Once constructed, the sketch can be reused multiple times as needed. These capabilities are crucial for applications that require substantial computational workload, such as recommendations in social networks, or those needing quick interactive responses, such as drill-down analytics and network-aware search systems [25, 39].

All-distances sketch (ADS) has recently emerged as a promising scheme for neighborhood sketching [9, 10]. It is a probabilistic data structure that is defined for each vertex, which includes capturing and sampling the neighborhood information for each vertex in relation to the entire graph. More precisely, an ADS of a vertex u is a summary of the distances of u connected to other "landmark" vertices. After constructing a raw graph into ADSs, many useful graph properties can be estimated accurately via the constructed ADSs during runtime by orders of magnitude faster. According to an existing study [1], ADS is the only sketching scheme that combines the following three characteristics. (1) **Multi-Functionality**: ADSs can be deployed to estimate various critical applications. (2) **Controllable and Guaranteed Accuracy**: ADS-based estimation offers accurate, guaranteed error bounds, and users can adjust the parameter to control the error bounds. (3) **Scalability**: The time complexity for constructing ADSs is near-linear to the graph scale.

Despite the fact that ADS is well-developed in theory, there is still a wide gap in its practical use in real-world cases, mostly because of its excessively high memory consumption. Specifically, recent efforts in this area are mainly focused on theoretical aspects. Hence, they propose algorithms and demonstrate their results with all-in-memory environments, assuming a large amount of available memory and running on relatively small graphs. Nevertheless, the state-of-the-art scheme to efficiently construct ADSs of all vertices requires an enormous amount of memory that can be up to 30x-60x larger than the graph itself. For instance, if we have a large-scale graph of 512 GB, the memory overhead of constructing its ADSs could require 30 TB, which is unaffordable for most machines/users nowadays. This severely hinders the use of ADS in real-world practice.

Handling ADSs with storage is an effective approach to mitigate the high memory costs. However, leveraging storage is a double-edged sword. In particular, storage I/O speed is generally slow, and managing ADSs with storage presents many technical challenges. Moreover, although graph processing with storage has been extensively studied [16, 20, 28, 40–42], the complex nature of using ADSs makes existing graph processing techniques incompatible. For example, since traditional graph processing usually handles one algorithm at a time, the number of active vertices (i.e., the vertices participate in a computational process in an iteration) is bounded by the number of all vertices $O(V)$. Conversely, the state-of-the-art method for constructing ADSs is to perform bounded graph traversals on all vertices, which can lead to the number of active vertices bounded by O(V^2). Thus, applying the existing graph processing techniques to ADS construction will lead to either huge memory consumption for managing their vertex attributes or an excessive amount of I/O for loading/saving those attributes (see Section 3 for more details). Further, unlike traditional graph processing, ADS estimation can handle multiple queries on different vertices simultaneously, necessitating the development of new interfaces and designs for ADS estimation. All these differences make managing ADSs with storage a new and challenging task.

To this end, this paper introduces Oasis, the first out-of-core approximate graph system that brings the ADS technique into practical use by leveraging storage effectively. Specifically, Oasis offers a holistic framework that facilitates both ADS construction and estimation through system-level optimizations, without altering the underlying ADS theory. For ADS construction, it presents a scheme that allows users to adjust the memory usage based on available memory of the machine. Additionally, to address performance bottlenecks during ADS construction, this work proposes designs such as *lock-free edge layout* to prevent the thread synchronization overhead, and *active data separation* and *selective ADS accessing* to reduce the unnecessary I/O for loading ADSs. For estimation, Oasis offers a user-friendly framework that allows users to implement and execute their ADS estimators easily. Further, to mitigate the slow storage I/O, this paper presents *locality-aware query assignment* and *grid-based estimation*, which can re-schedule the queries to achieve better locality of access during estimation.

Our evaluation results demonstrate that Oasis[1] offers a highly practical graph processing solution than other state-of-the-art work by achieving an outstanding trade-off between memory and performance. Specifically, in comparison to in-memory ADS-based solution, although Oasis is slower by around 1.8x for construction and 1.7x for estimation, it can save a significant 13.8x of memory, which demonstrates the high practicality of Oasis by allowing it to process much larger graphs. Moreover, when compared with the graph system that compute exact answer, Oasis can significantly improve the execution time by several orders of magnitude with only a minor loss of accuracy. Oasis offers the flexibility to control memory overhead, enabling users to adjust memory usage according to their machine's capacity. We also observe that ADS and storage synergize with each other; ADS's exceptional speedup can outshine the slow I/O bandwidth, while the huge size of ADS can reside in the cheap-and-massive storage. These evaluation results suggest that Oasis is a superior option for graph processing, with efficient performance, low memory usage, and high levels of flexibility.

2 Background and Motivation

2.1 Preliminaries

Let $G = (V, E)$ be a graph where V and E denote the vertex set and edge set of G, and $|V|$ and $|E|$ means the number of vertices and edges, respectively. Vertices typically represent the entities in real world, and edges describes the neighoring relationship of two connected vertices. For an edge $e = (u, v)$ in a directed graph, v is referred to as the *out-neighbor* of u, while u is referred to as the *in-neighbor* of v. u can also be described as the *source* vertex of e, and v is the *destination* vertex of e. The terms *out-degree* and *in-degree* further indicate the number of out-neighbors and in-neighbors for a given vertex, respectively In practice, in the vertex data, each vertex is assigned with a distinct value, called vertex ID, for identification purpose. Edge data, on the other hand, represent all the edges in the form of edge list that enumerates the neighbors' vertex IDs for a specific vertex, and all the edge

[1]We open source Oasis at https://github.com/tsunyuyang/Oasis

lists are further sorted by vertex IDs.

2.2 All-Distances Sketches

The workflow of using All-distance sketches (ADSs) can be illustrated in Figure 1, which involves two main steps, namely *ADS construction* and *ADS estimation*. ADS construction is to produce ADSs based on its raw graph, and ADS estimation uses the produced ADSs to estimate the approximate answers. Please note that the ADS construction only needs to be performed once per graph, and we can estimate the various graph properties via the produced ADSs multiple times.

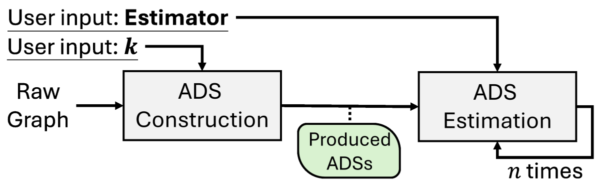


Figure 1: The workflow of utilizing ADSs.

ADSs are defined with a integer parameter k and a random rank assignment function r to all vertices. The parameter k decides the trade-off between sketch size and estimation accuracy, and r is rank function, where $r(v) \rightarrow [0,1]$ for any vertex $v \in V$. For $u, v \in V$, we define $N(u,v)$ as the set of vertices that are closer to u than v. Thus, it is can formally defined as follow: $N(u,v) = \{x \in V \mid d_{u,x} < d_{u,v}\}$, where $d_{i,j}$ indicates the shortest-path distance from vertex i to j. For a vertex subset $S \subseteq V$, we define the function $k_r^{th}\{S\}$ as the k-th smallest rank value for vertices in S. If $|S| < k$, we define $k_r^{th}\{S\} = 1$. To concisely formulate the *ADS Equation*, we define threshold rank value $\pi(u,v) = k_r^{th}\{N(u,v)\}$ for any pair of vertices u and v. Using $\pi(u,v)$, the ADS of a vertex u is defined as follows.

$$ADS(u) = \{(v, d_{u,v} \mid v \in V, r(v) < \pi(u,v))\} \tag{1}$$

In this paper, each $(v, d_{u,v})$ is referred to as an *ADS entry*, and an *ADS* is referred to as all entries of a vertex's ADS. The size of an ADS can be calculated as follows [10]. For a vertex u, let n_u be the number of reachable vertices from u, and $H(i)$ be the i^{th} harmonic number. The expected size of $ADS(u)$ is $k(1 + H(n_u) - H(k))$. Because $n_u \leq |V|$ and $H(|V|) = O(log|V|)$, the expected size of $ADS(u)$ is $O(klog|V|)$. Thus, the total size of all ADSs is $O(|V|klog|V|)$.

After the ADSs are produced based on Equation 1, many types of useful graph properties and applications can be efficiently estimated with the guarantee of accuracy. Since ADS is a probabilistic data structure that has attracted much attention, existing works have proposed many methods to calculate approximate answers via ADSs. These methods are referred to as *ADS estimators*. Following, we discuss several popular and useful graph properties and applications that can be approximated by ADS estimators [1].

Neighborhood Function [9, 10, 13]. This function $n_d(v)$ returns the number of vertices that can be reached from v within the distance d. $n_d(v)$ can be estimated by the ADS of v with the coefficient of variation bounded by $\frac{1}{\sqrt{2(k-1)}}$.

Shortest-Path Distance [11]. The shortest path distance from vertex u to v can be estimated by $ADS(u)$ and $ADS(v)$. This estimation is $O(logn)$-approximation for constant k, and $(2a-1)$-approximation when $k = n^{1/a}$ (for some $a \geq 1$).

Closeness Centrality [10]. Closeness centrality $C_{\alpha,\beta}(u)$ is one of the most fundamental way to measure the importance of a vertex u, with distance decay function α and vertex weight function β. $C_{\alpha,\beta}(u)$ can be estimated by the ADS of u with coefficient of variation bounded by $\frac{1}{\sqrt{2(k-1)}}$.

Closeness Similarity [11, 35]. Closeness similarity is used to measure how similar two vertices are in terms of their positions to the entire graph. For two vertices $u, v \in V$, the closeness similarity between u and v can be estimated using the ADSs of u and v with the error bounded by $\frac{1}{\sqrt{k}}$.

Average Distance and Effective Diameter [4]. These two are the fundamental properties of a graph. They receive a lot of interest because of their relation to the *small world phenomenon* [38]. They can be accurately estimated by ADSs with confidence intervals.

Reverse Raking and Nearest Neighbors [6]. Reverse ranking measures the relevance of a vertex u to a vertex v by the number of vertices that are closer to v than u. The approximate reverse nearest neighbors (rNNs) of any size can be efficiently obtained by using the neighborhood function estimation.

Continuous-Time Influence [7, 12, 15]. The continuous-time influence model is an influence propagation model with time decay property from a set of vertices X. The approximate influence can be estimated by using the ADS of u, where $u \in X$. Its coefficient of variation is bounded by $\frac{1}{\sqrt{2(k-1)}}$.

2.3 Motivation and Challenges

2.3.1 ADS Construction

Although there are several existing schemes that can construct ADSs correctly, they are either slow or consume a huge memory amount. For instance, the most straightforward scheme is to implement ADS Equation 1 directly. We refer to this scheme as *Basic*. It simply runs the single-source shortest paths (SSSP) from every vertex as root independently. For each SSSP, the root's ADS will be updated by the traversed vertices based on ADS Equation 1. However, this scheme is extremely inefficient because it requires an extensive $O(VE)$ edge traversals in total, which is impractical for large-scale graphs with millions of vertices and billions of edges.

A state-of-the-art (*SOTA*) scheme [6] is proposed to construct ADSs efficiently. The pseudocode is presented in Algorithm 1. The core concept of this SOTA scheme is to perform the *pruned* shortest-path algorithm on *transpose graph*. As shown in Algorithm 1, it starts from the vertex of the lowest rank value and then perform shortest-path algorithm. If the condition of Line 4 is true, the traversed vertex is pruned from the shortest-path algorithm. Otherwise, we include the root vertex u into the ADS of v and keep traversing. This scheme only requires $O(EklogV)$ edge traversals during ADS construction, which is significantly less than $O(VE)$.

Algorithm 1 State-of-the-art ADS Construction Scheme

```
1: for each vertex $u \in V$ by increasing $r(u)$ do
2:     Run shortest-path algorithm from $u$ on $G^T$
3:     for each visited vertex $v$ do
4:         if $d_{v,u} > k^{th}\{y \mid (x,y) \in ADS(v)\}$ or $|\{x \in ADS(v) \mid d_{v,x} \le d_{v,u}\}| > k$ then
5:             Prune $v$ from the shortest-path algorithm
6:         else
7:             $ADS(v) \leftarrow ADS(v) \cup (r(u), d_{vu})$
8:         end if
9:     end for
10: end for
```

However, even though SOTA is efficient, it requires a significant amount of memory for holding ADSs. This is because Algorithm 1 accesses the ADSs of visited vertices that are randomly traversed (i.e., Line 3-8), causing the access pattern to ADSs also random and unpredictable. By contrast, Basic scheme only accesses the ADS of root for each shortest-path search. Thus, the memory cost of Basic can be better controlled than SOTA.

Table 1 shows the comparison between these two ADS construction schemes, namely *Basic* and *SOTA*, with different k values on soc-LiveJournal graph [2, 22]. The last row in Table 1 reveals the ratio of ADS sketch size over the raw graph size. It can be first observed that the Basic scheme generally takes a long time to complete compared to the SOTA scheme. Specifically, the difference can be up to 2-3 orders of magnitude. The reason behind this is that Basic scheme needs $O(VE)$ edge traversals during ADS construction, while SOTA only requires $O(EklogV)$ traversals. This makes the Basic scheme inefficient in ADS construction. Moreover, as the input graph becomes larger, the Basic scheme becomes more impractical for requiring an unbearably long time to finish. Take Twitter graph [14] with $k = 32$ as an example, the Basic scheme cannot complete this task within a year (it will take around 472 days to complete, as estimated from the completion time of a subset). On the other hand, the SOTA scheme can produce its ADSs in 3.4 days, which is an efficient time, particularly considering the scale of the graph.

2.3.2 ADS Estimation

After the ADSs are produced, many useful graph properties and applications can be estimated accurately, as discussed in Section 2.2. Following, we will illustrate that out-of-core ADS estimation is an appealing and practical solution with three relevant metrics: *execution time*, *memory overhead*, and *accuracy*.

Table 1: Existing schemes for ADS construction on soc-LiveJournal graph with different k values. The raw graph size of soc-LiveJournal is 527 MB.

	$k = 4$	$k = 16$	$k = 32$	$k = 64$
Time (Basic)	2.58 hrs	2.68 hrs	2.59 hrs	2.61 hrs
Memory (Basic)	2.7 GB	8.0 GB	14.1 GB	25.4 GB
Time (SOTA)	105 sec	326 sec	424 sec	894 sec
Memory (SOTA)	5.1 GB	17.5 GB	33.1 GB	63.4 GB

This section evaluates out-of-core graph system (i.e., Graphene [24]), in-memory graph system (i.e., Ligra+ [31]), in-memory ADS estimation (abbrev. MAE), and out-of-core ADS estimation (abbrev. OAE). Graphene is a recent out-of-core graph system which keeps the edge data in storage. Ligra+ is an in-memory graph system that can traverse graphs efficiently. MAE keeps the ADSs in memory during estimation, while OAE stores them in storage and on-demand load the needed ones into memory for estimation. We use a well-known graph application, *closeness centrality*, as example to evaluate these graph processing solutions. The task is to calculate the closeness centrality of 15,000 vertices by random selection. For ADS estimator, we use the Historic Inverse Probability (HIP) [10], which is an elegant and accurate ADS-based estimator.

Table 2 shows the results of Twitter graph [14] with $k = 32$. It can be first observed that Graphene performs the worst in execution time, while Ligra+ improves this execution time by 2.6x but requires 9.2x more memory than Graphene. As for ADS estimation (including both MAE and OAE), their amazing speedups significantly outperform Graphene and Ligra+, with a minor loss of accuracy only. On the other hand, although the MAE is slightly faster than the OAE, the memory overhead of OAE is significantly smaller than the MAE. Moreover, compared to Ligra+, OAE is significantly faster and even requires less amount of memory. Thus, the outstanding performance of out-of-core ADS estimation inspires us to develop the first out-of-core ADS-based graph system.

It is worth noting that the result of out-of-core ADS estimation shown in Table 2 is via a naive implementation. In Section 3.3, we explore optimizations to further improve the performance of out-of-core ADS estimation.

Table 2: Evaluation of existing graph processing solutions.

	Graphene	Ligra+	MAE	OAE
Exe. Time	9.2 hours	3.6 hours	0.014 sec	0.024 sec
Mem. overhead	1.2 GB	11 GB	147 GB	1.4 GB
Avg. Accuracy	100%	100%	95.4%	95.4%

3 Oasis System

3.1 Overview

Motivated by Section 2.3, this section introduces Oasis, the first out-of-core approximate graph system based on ADSs. The main goal of Oasis is to make the ADS technique practical to users. To achieve this, we aim to develop Oasis into a holistic, ADS-based graph system that can efficiently manage ADSs on storage.

The system architecture and workflow of Oasis are illustrated in Figure 2. There are two main modules, which are ADS Construction Module (details in Section 3.2) and ADS Estimation Module (details in Section 3.3). To begin with, users provide two inputs, which are k value and the number of *partitions* (this will be discussed shortly in Section 3.2), to ADS Construction Module. Oasis will produce the ADSs based on the raw graph and store them in the storage. After ADSs are produced, users provide their ADS estimators with the Oasis's interface to the ADS Estimation Module. This module will automate and optimize the process of loading required ADSs and executing the estimators to get the approximate answers efficiently and accurately.

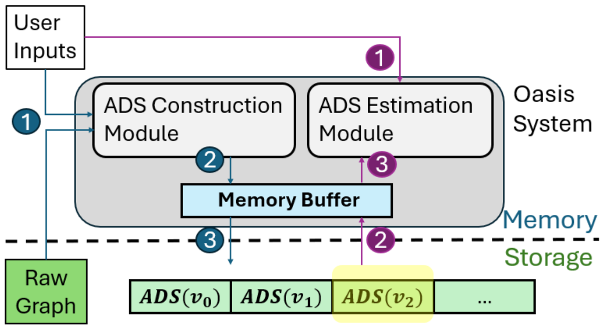


Figure 2: The system architecture of Oasis.

3.2 Oasis ADS Construction

3.2.1 Partition-based ADS Construction with Lock-free Layout

This section aims to propose an ADS construction scheme that is efficient in terms of edge traversal and can also alleviate the issue of large memory cost by using storage. Specifically, we convert the SOTA ADS construction scheme into a *scan-and-merge* approach [4, 26], which starts the shortest-path searches of all vertices together to increase the utilization of loaded data. The ADS construction runs in iterations until convergence (i.e., no more searches are running).

Our ADS construction scheme is built upon *partitioning* technique to effectively address the problems of insufficient memory while achieving good locality of access. The partitioning technique is widely used in many traditional out-of-core graph systems [16, 20, 36, 42]. The idea behind partitioning is to divide the data into multiple disjoint partitions so that we can execute one or two partition(s) at a time to minimize the peak memory overhead. Since the total size of ADSs is enormous, partitioning is a well-matching technique for Oasis to create ADS partitions. Therefore, users can run graphs of different scales with Oasis by controlling the number of ADS partitions to meet the machine's memory capacity. Please note

Algorithm 2 Oasis ADS Construction

```
1:  Create the transpose graph $G^T$
2:  Divide $G^T$ into partitions
3:  for each vertex $v \in V$ do
4:      $ADS(v) \leftarrow ADS(v) \cup (v,0)$                    ▷ Mark as active
5:  end for
6:  $num\_active = |V|$
7:  $iteration = 0$
8:  while $num\_active > 0$ do                  ▷ ADS Constr. w/ Partitions
9:      $num\_active = 0$
10:     for each destination partition $P_x$ do
11:         Load ADSs of $P_x$ into memory
12:         for each source partition $P_y$ do
13:             Load active ADSs of $P_y$ ($A_y$) into memory
14:             Load edges $E_{xy} = \{(a,b) \mid a \in P_y, b \in P_x\}$
15:             for each active $(c,d) \in ADS(u)$, $ADS(u) \in A_y$ do
16:                 for each edge $(u,v) \in E_{xy}$ do
17:                     if $r_c < k_r^{th}\{(x,y) \in ADS(v) \mid y < d\}$ then
18:                         $ADS(v) \leftarrow ADS(v) \cup (c, d+w)$
19:                                 ▷ $w$ is the weight between edge $(u,v)$
20:                         Mark this entry active for next iteration
21:                         $num\_active += 1$
22:                     end if
23:                 end for
24:             end for
25:         end for
26:         Write ADSs of $P_x$ into storage
27:         $iteration$++
28:     end for
29: end while
30: for each vertex $u \in V$ do                      ▷ Final adjustment
31:     Sort $ADS(u)$ by increasing distance
32:     for each ADS entry $(x,y) \in ADS(u)$ do
33:         Create auxiliary data for ADS estimators
34:     end for
35: end for
```

that the total size of ADSs is predictable, which is bounded by $O(VklogV)$. Suppose the machine's memory capacity is M and the number of ADS partitions is P, we should satisfy the inequality $O(VklogV)/2P < M$ by adjusting P.

The pseudocode of Oasis ADS Construction is shown in Algorithm 2. It begins by initializing the necessary data for Oasis ADS construction. Line 1 creates the transposed graph G^T, and Line 2 divides G^T into partitions based on the discussion in Section 3.1. Transposing a graph can be easily achieved by a single pass over all the edges while swapping the source and destination vertex IDs. After the initialization (Lines 1-7), the core function of Oasis ADS construction begins (Line 8-Line 29). The basic concept is to perform ADS construction iteration by iteration. In each iteration, Oasis goes through all the combinations of ADS partitions (i.e., P_x and P_y). Line 10 selects a to-be-updated ADS partition P_x, and Line 12 goes through all other ADS partitions P_y one at a time. Lines 17-22 implement the ADS equation. If the condition of Line 17 is true, we include this entry into the $ADS(v)$ and mark this entry active for the next iteration. This act means that the shortest-path search keeps traversing from this vertex in the next iteration. Otherwise, Oasis does nothing, which is akin to the process of pruning. The Oasis ADS construction ends if there is no more active ADS entry. Finally, Lines 30-35 allow users to do the final adjustment for ADS entries or create auxiliary data for their ADS estimators. Line 31 sorts the all the entries of an ADS in increasing distance, which is helpful for some ADS estimators.

One may wonder if the existing, well-developed out-of-core graph systems can be utilized for ADS construction. However, they are unsuitable for two main reasons. First, out-of-core graph systems usually focus on reducing access to edge data to enhance the overall processing performance, as edge data is the largest data for graph processing. Conversely, because the largest data for ADS construction is the ADS itself, out-of-core graph systems are less effective in ADS scenarios. Secondly, out-of-core graph systems typically employ an in-memory array of size $O(V)$ to indicate the active vertices and distances for each shortest-path search. Nevertheless, our ADS construction performs shortest-path searches from all vertices simultaneously, requiring the graph systems to use $O(V^2)$ of memory space. This is impractical because $O(V^2)$ is even larger than the ADS structure $O(VklogV)$ itself.

In contrast to traditional graph algorithms which generally update destination vertices with low-cost operations, updating destination ADS is much more complex, involving checking the distances and ranks of each ADS entry (i.e., Line 17-22). When multiple threads work on the same ADS partitions, locks are often required to resolve conflicts if many threads update the same destination ADS. This severely degrades the overall processing performance. Hence, on top of partitioning technique, we further introduce a *lock-free edge layout*, as shown in Figure 3. The core idea behind this is to split an *edge grid* into multiple equal-sized *blocks*, each containing a disjoint set of destination vertices. Thus, when each thread computes a different block, we can prevent two threads from updating the same ADS, thereby avoiding the use of locks to resolve conflicts. Creating a lock-free edge layout can be done in two simple passes over the edge data. In the first pass, we collect neighbor information to determine the destination vertex ID range for each block. This process ensures that each block is of similar size. In the second pass, we reorder the edges by writing each edge to its corresponding block, based on the destination vertex ID of each edge.

3.2.2 Active Data Separation

This section presents *active data separation*, which aims to minimize the loading of source/active ADSs (i.e., Line 13 in Algorithm 2). Active ADSs refer to the set of ADSs that were added in the previous iteration and are needed for processing in the current iteration. Since ADS is the largest data structure during construction, how to minimize the I/O amount of loading ADSs is crucial for optimizing Algorithm 2. Furthermore, the overall cost of loading source/active ADSs is substantial and grows in proportion with the number of partitions, as Algorithm 2 iterates through all source ADS partitions for each destination partition.

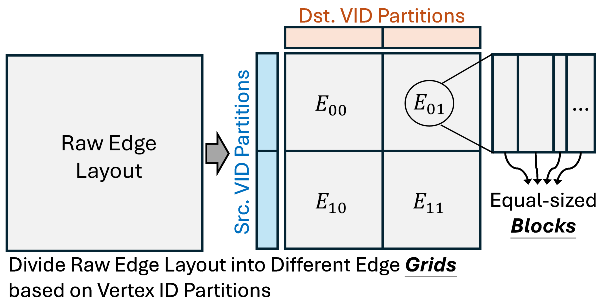


Figure 3: Example of lock-free edge layout.

In this scenario, we introduce *active data separation* to duplicate the active ADSs and stores them in a separate file. When constructing ADSs, we not only write the newly-generated active ADSs in the original ADS structure but also store them in a separate file containing the new active ADSs only. Therefore, in the next iteration, this design allows us to directly load active ADSs from the separate file instead of loading and searching active ADSs from the huge original ADS structure, thereby saving I/O. The processed active ADSs can be deleted in bulk from the file, and the construction process will produce new active ADSs in the current iteration. We can briefly calculate the overall I/O amounts between traditional method and active data separation. Suppose I indicates the number of iterations and P denotes the number of partitions, the total I/O amount of loading source/active ADSs with traditional method is $O(VklogV \cdot P \cdot I)$. On the other hand, the total I/O amount with active data separation is only $O(2 \cdot VklogV)$, where the value 2 means the reads and writes to the separate file, and the total number of active ADSs is bounded by $O(VklogV)$.

3.2.3 Selective ADS Accessing

This section presents *selective ADS accessing* to reduce the I/O load for *destination ADSs* (i.e., Line 11 in Algorithm 2). The access pattern of ADS construction is fundamentally graph traversal; it begins with a small number of active vertices and rapidly increases during the early to middle iterations. As ADS construction nears convergence, the working set of ADSs becomes significantly smaller, but it is unknown which ADSs will be required until they are actually being processed. Naïvely loading an entire partition of ADSs into memory wastes I/O bandwidth. Thus, selective ADS accessing is proposed to spend additional I/O reads for edge data to gather neighborhood first, allowing us to identify necessary ADSs and thus selectively load them. Notably, since the size of edge data is typically much smaller than that of ADSs, this design is effective to save I/O amounts.

3.3 Oasis ADS Estimation

3.3.1 Programming Framework and Interface

The ADS estimators come in different types. Estimating different graph properties and applications may require different ADS estimators. Therefore, it is important for Oasis to provide a user-friendly interface for users to express or develop their own ADS estimators of interest.

To our knowledge, the existing ADS estimators utilize either one ADS or two ADSs to derive an answer. We categorize them into two types: *single-ADS* and *dual-ADS*. For instance, closeness centrality [10] uses a single-ADS estimator to compute a vertex's centrality relative to entire graph; closeness similarity [11] requires a dual-ADS estimator to quantify a similarity value between two vertices' ADSs. Although the multi-ADS problems also exist, they are primarily extended from single-ADS/dual-ADS estimators. An example is the top-k centrality problem, which employs a single-ADS estimator to calculate centrality scores for a vertex set and then identifies the top-k vertices with the highest centrality.

Figure 4 shows the Oasis's estimation framework that can support both single-ADS and dual-ADS. The ADS estimators are provided by users, and the query queue is filled by users. Each query in the queue contains the information of *target vertex ID(s)* (i.e., which ADSs should be loaded and estimated) and *pointer* (i.e., point to the ADS estimator of interest). Thus, during ADS estimation, Oasis can handle those queries by loading the required ADSs into memory and using the correct estimators to process.

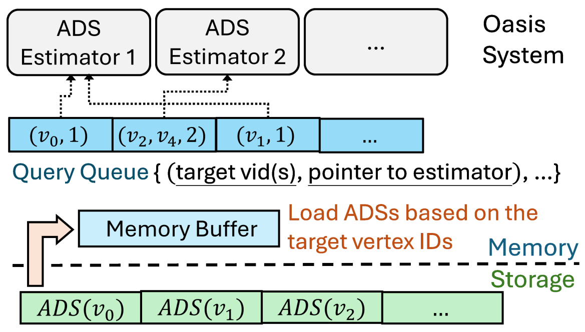


Figure 4: The framework of Oasis ADS estimation.

Algorithm 3 shows the pseudocode of Oasis ADS estimation framework. Users need to define ADS estimator function and fill in Query Queue before execution. Taking closeness centrality as an example, we first define how to use each ADS entry for computing the centrality (Line 1-5), and specify which vertices' centrality is of interest (Line 35). Oasis system will handle the remaining processes. Next, Line 38 splits all the queries in Query Queue among multi-threads. Each thread is responsible for a disjoint set of queries in Line 39. In EXECUTE function, Line 14 declares a bitmap to keep track the to-be-loaded ADSs, and Line 15 declares a buffer to hold the loaded ADSs. Both are per-thread data structures. We set the size of memory buffer to be 8 MB by default as a different buffer size affects the overall performance trivially based on our survey. Next, the while loop (Lines 17-27) will identify the queries where the union of the required ADSs of these queries can fit inside the memory buffer. After the required ADSs are loaded, Oasis perform the ADS estimation via the user-provided estimator. Algorithm 3 shows the examples of ADS estimators, which are ADS_Estimator_1 for *closeness centrality* and ADS_Estimator_2 for *closeness similarity*.

Algorithm 3 Oasis ADS Estimation

```
 1: procedure ADS_ESTIMATOR1($ads, len, idx$)
 2:     for $i = 0..len$ do                                  ▷ Closeness Centrality
 3:         $ans[idx]$ += HIP_weight · IMP($ads[i].id$) · $\frac{1}{ads[i].dist}$
 4:     end for
 5: end procedure
 6:
 7: procedure ADS_ESTIMATOR2($ads_1, len_1, ads_2, len_2, idx$)
 8:     Let $A$ be the set from $ads_1[0]$ to $ads_1[len_1 - 1]$
 9:     Let $B$ be the set from $ads_2[0]$ to $ads_2[len_2 - 1]$
10:     $ans[idx] = \frac{|A \cap B|}{|A \cup B|}$                          ▷ Closeness Similarity
11: end procedure
12:
13: procedure EXECUTE($beg\_idx, end\_idx$)
14:     Declare a bitmap $bmap$ to record the to-be-loaded ADSs
15:     Declare a memory buffer $Buf$ to hold required ADSs
16:     $io\_size = 0$
17:     while $beg\_idx < end\_idx$ do
18:         if $io\_size + Q[beg\_idx] <$ buffer size then
19:             Record $Q[beg\_idx]$ into $bmap$
20:         else
21:             Load the required ADSs into $Buf$ based on $bmap$
22:             Call the corresponding ADS Estimators
23:             Clear $bmap$
24:             $io\_size = 0$
25:         end if
26:         $++beg\_idx$
27:     end while
28:     if $io\_size > 0$ then
29:         Do Lines 21-22 again
30:         $io\_size = 0$
31:     end if
32: end procedure
33:
34: procedure MAIN
35:     Fill the queries in the Query Queue $Q$ by user
36:     $n \leftarrow$ number of elements in Query Queue
37:     Declare an $n$-sized answer array: $ans[.]$
38:     $[beg\_idx, end\_idx] \leftarrow$ Split all queries across multi-threads
39:     Execute($beg\_idx, end\_idx$)
40: end procedure
```

3.3.2 Locality-aware Query Assignment

Based on the framework presented in Section 3.3.1, this section studies how to improve the execution performance of ADS estimation. Specifically, different estimators may work on the same set of ADSs. When we have multi-threads, it is important to assign the same set of ADSs to the same thread so as to enhance the utilization of loaded ADSs. In this context, this section presents *locality-aware query assignment*. Specifically, the query assignment among multi-threads can largely affect the overall performance in Algorithm 3. This is because, if a thread is assigned with many queries requiring the same $ADS(u)$, the thread only needs load $ADS(u)$ once to serve many queries, considerably improving the I/O utilization. Based on this motivation, the proposed locality-aware query assignment works as follow. First, it sorts all the queries in the Query Queue with an increasing order based on the target vertex IDs. Nevertheless, because finding the optimal locality of assignment for dual-ADS queries can be quite complex, we heuristically sort the dual-ADS queries by the first target vertex ID only to simplify the cost spending on the query assignment. After sorting, the good locality of access to the required ADSs can be easily achieved by processing the queries in the sequential order. Thus, the next step is to split all the queries sequentially and equally to each thread.

3.3.3 Grid-based Estimation

This section introduces *grid-based estimation* to optimize the execution of dual-ADS queries. Specifically, dual-ADS queries could incur a more complex access pattern than that of single-ADS queries. For example, suppose there is a set of vertices, our task is to measure the closeness similarity between every pair of vertices within this set. This scenario cannot be easily optimized by the locality-aware query assignment because an ADS will be loaded into memory multiple times by different threads regardlessly. Thus, the *grid-based estimation* is proposed to tackle this issue. In fact, after Oasis ADS construction, our memory should be enough to hold two ADS partitions, which can be taken advantage of by the grid-based estimation. First, the grid-based estimation splits all the dual-ADS queries into different *query grids* based on ADS partitions, as shown in Figure 5. Following, Oasis runs one grid at a time by loading the required ADSs into memory. It could have good locality of ADS accesses when Oasis process these grids in a sequential order. For example, when Oasis moves from (P_0, P_0) to (P_0, P_1), we can hold P_0 in memory to reduce unnecessary I/O opportunistically.

4 Evaluation

4.1 Evaluation Setup

As shown in Figure 1, the process of utilizing ADSs can be divided into two parts: ADS construction and ADS estimation. Hence, this work also presents the evaluation of these two parts separately in different sections (i.e., Section 4.2 and Section 4.3).

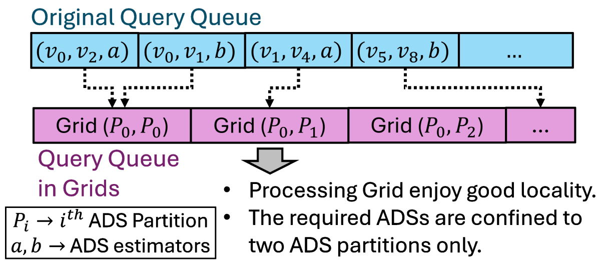

Original Query Queue
(v_0, v_2, a)
(v_0, v_1, b)
(v_1, v_4, a)
(v_5, v_8, b)
...
Grid (P_0, P_0)
Grid (P_0, P_1)
Grid (P_0, P_2)
...
Query Queue in Grids
$P_i \rightarrow i^{th}$ ADS Partition
$a, b \rightarrow$ ADS estimators
Processing Grid enjoy good locality.
The required ADSs are confined to two ADS partitions only.

Figure 5: Example of grid-based estimation.

Table 3: Evaluated graph datasets.

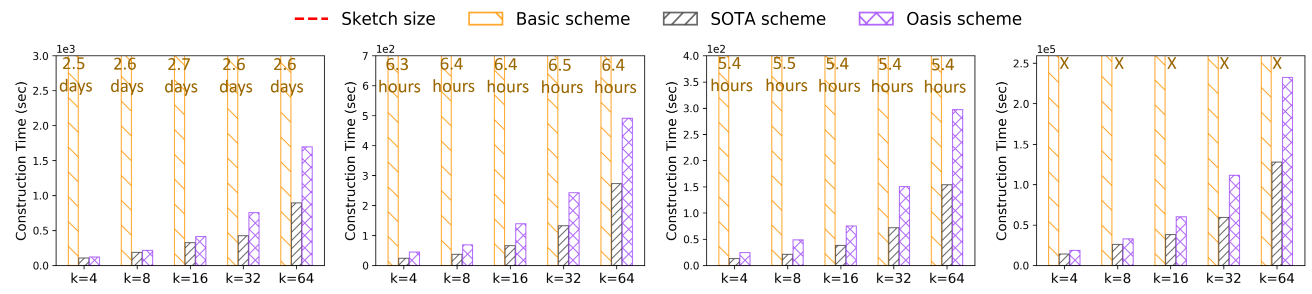
Name	V	E	Type
Pokec [34]	1.6 M	31 M	directed
soc-LiveJournal [2]	4.8 M	69 M	directed
hollywood2009	1.1 M	113 M	undirected
Twitter [14]	42 M	1.4 B	directed

Table 3 lists the evaluated graphs. These graphs are publicly available real-world graphs from the Stanford Large Network Dataset Collection [21] and Laboratory for Web Algorithms [3, 5]. The evaluated graphs involve both directed and undirected graphs of various sizes. To handle undirected graphs, there is a common method to convert all the undirected graphs into directed graphs by replacing every undirected edge with two directed edges in opposite directions (i.e., outgoing and incoming edges) [30]. This method is also deployed in many graph systems such as GridGraph [42], Graphene [24], Lumos [36], etc. Oasis also follows this common method to handle undirected graphs.

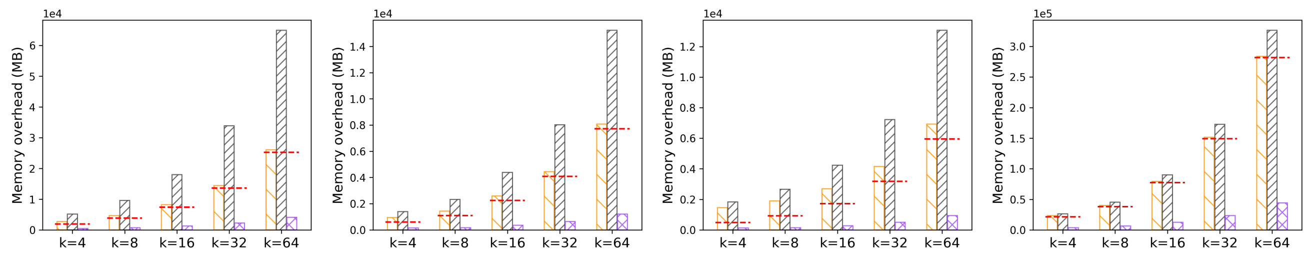
To compare all types of graph processing solutions on the same platform, all experiments are conducted on the same server: HPE ProLiant DL560 Gen10 server with Intel Xeon Platinum 8160 CPU and 32 x 32GB Dual Rank DDR4-2666 memory on Debian GNU/Linux 9, and 2 x 1 TB Samsung NVMe SSD drives [32] with 6.0 GB/s sequential read bandwidth in total. To restrict the computational resource, we use `taskset` to confine the used cores, and the number of threads is set to 16 for all programs because it is a reasonable cost for most users nowadays. Oasis divides the evaluated graphs into 16 partitions by default to restrict the memory overhead. Section 4.4 further discusses the performances and memory overheads under different number of partitions.

4.2 ADS Construction

This section focuses on the evaluation of ADS construction. We implement the basic ADS construction scheme (denoted as *Basic*) and the state-of-the-art ADS construction scheme (denoted as *SOTA*). The SOTA scheme is referenced based on [6]. Both of them had been elaborated in Section 2.3. Section 4.2.1 compares these two in-memory ADS construction schemes against Oasis version. Section 4.2.2 further shows the design choices of Oasis ADS construction.



(a) Construction time on soc-LiveJournal. (b) Construction time on Pekoc. (c) Construction time on hollywood09. (d) Construction time on Twitter.



(e) Construction memory on soc-LiveJournal. (f) Construction memory on Pekoc. (g) Construction memory on hollywood09. (h) Construction memory on Twitter.

Figure 6: Overall comparison among Oasis's ADS construction and other ADS construction schemes.

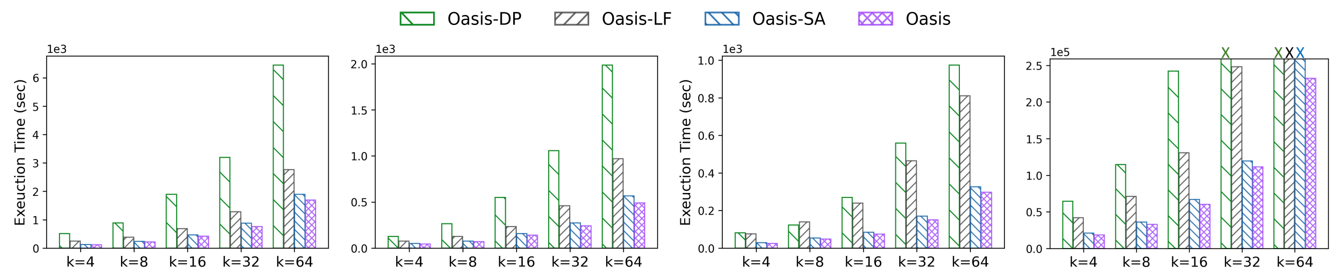
4.2.1 Comparison of ADS Construction

Figure 6 reveals the overall ADS construction costs of different schemes with different k values in terms of construction time, construction memory, and sketch space. Construction memory indicates the memory amount needed during ADS construction, and sketch size is the total size of final produced ADSs. We compare Oasis against Basic and SOTA.

It can be first observed that the Basic scheme generally requires a significant amount of time to complete. We do not show the complete execution time results of Basic scheme in Figure 6 because these results will make the figures visually skewed. Instead, we provide the real execution times with text in Figure 6. For k from 4 to 64, the execution times of Basic scheme for soc-LiveJournal, Pokec, hollywood09 graphs are around 2.6 days, 6.4 hours, and 5.4 hours, respectively. It is worth noting that the different k values trivially affect the performance of Basic scheme. This is because Basic spends $O(VE)$ edge traversals, independent from the value of k. On the other hand, we terminated the Basic scheme for Twitter since it cannot complete this task within 3 days. Regarding the memory overhead, Basic scheme holds the entire ADSs in memory during ADS construction, so its memory overhead is dominated by the sketch size.

Compared to Basic, SOTA significantly improves the execution time. The improvement of SOTA over BASIC is around hundreds to thousands of times. This great amount of improvement is credited to the reduction in the number of edge traversals. Specifically, Basic scheme takes $O(VE)$ edge traversals, while SOTA scheme only requires $O(EklogV)$. Besides, because the number of edge traversals of SOTA depends on k value (i.e., $O(EklogV)$), the execution times of SOTA increases along with the k value, which is around 1.4x-2.1x as the k value doubles. To enable SOTA to work with multi-threads, we declare per-thread data structure during SOTA ADS construction. Thus, the total memory overhead of SOTA is larger than its sketch size by 2.13x on average. On the other hand, running SOTA with a single thread could alleviate the memory overhead at the cost of degrading execution time. We use $k = 16$ as a demonstration. The memory overhead of single-threaded SOTA is roughly the same as the sketch size, but the execution time is 691 sec, 155 sec, 84 sec, and 23.9 hours for soc-LJ, Pekoc, hollywood, and Twitter, respectively, indicating higher memory usage but potentially slower performance compared to Oasis.

Oasis offers a practical scheme for ADS construction. Compared to Basic, Oasis notably improves the execution times by hundreds to thousands of times. Compared to SOTA, the memory overhead of Oasis is considerably less by 13.8x on average. Although Oasis is slower than SOTA scheme by 1.79x on average, we believe this is a worthy deal due to the considerable amount of memory saving, which enables the ADS technique with reasonable costs. Moreover, users can control the memory overhead in Oasis by adjusting the number of partitions, which offers greater flexibility than SOTA. Section 4.4 will study the impact with different numbers of partitions in Oasis. Lastly, while existing graph systems can technically run ADS construction, their performance is often suboptimal because they are not designed for this task. To illustrate, we run GridGraph [42] with $k = 16$. For soc-LJ,



(a) Executoin time on soc-LiveJournal. (b) Executoin time on Pekoc. (c) Executoin time on hollywood09. (d) Executoin time on Twitter.

Figure 7: Performance studies of different major designs in Oasis ADS construction.

Pekoc, and hollywood, the construction times are 52 min, 15.2 min, and 14.3 min, respectively. For Twitter, we terminate the program as it cannot complete within 3 days. These significant inefficiencies in existing graph systems for ADS construction highlight the necessity of Oasis.

4.2.2 Design Choices for Oasis ADS Construction

This section demonstrates the performance impact of the major designs in Oasis ADS construction. Specifically, we evaluate lock-free edge layout, active data separation, and select ADS accessing, respectively. We compare the entire Oasis ADS construction scheme (denoted as Oasis) against Oasis without lock-free edge layout (denoted as Oasis-LF), Oasis without active data separation (denoted as Oasis-DP), and Oasis without select ADS accessing (denoted as Oasis-SA). The results are presented in Figure 7.

Overall speaking, the design of active data separation impacts performance the most. For soc-LiveJournal, Pokec, hollywood09, and Twitter, Oasis averagely improves Oasis-DP by 4.2x, 3.8x, 3.3x, and 3.7x in terms of execution times. This amount of improvement is contributed to the I/O reduction in loading the active ADSs. Let P be the number of partitions, the I/O of loading the source ADS partitions is $P \times VklogV$ for *each iteration*; with active data separation, the I/O of loading the active ADSs is $2P \times VklogV$ for *all iterations*. On the other hand, the design of lock-free edge layout also contributes a decent amount of execution time improvement. Specifically, for soc-LiveJournal, Pokec, hollywood09, and Twitter, Oasis improves the performance of Oasis-LF by 1.76x, 1.81x, 2.97x and 2.2x on average. This is because, without lock-free edge layout, the utilization of multi-threads becomes low since the locks are used to avoid the write conflict of updating the same ADS. Lock-free edge layout enables all threads to work together without locks needed. Finally, the design of selective ADS accessing can reduce the unnecessary ADS accessing when the ADS construction process is close to convergence, thereby reducing I/O. For soc-LiveJournal, Pokec, hollywood09, and Twitter graphs, Oasis improves Oasis-SA by 11.3%, 12.8%, and 9.5%, 10.9% on average in terms of execution times. It can be observed that the improvement brought by selective ADS accessing is less than those of the other designs. This is because the other two designs are effective for all iterations, while the design of selective ADS accessing is beneficial for the last several iterations. Nevertheless, selective ADS accessing still incurs an observable 11.1% improvement on average.

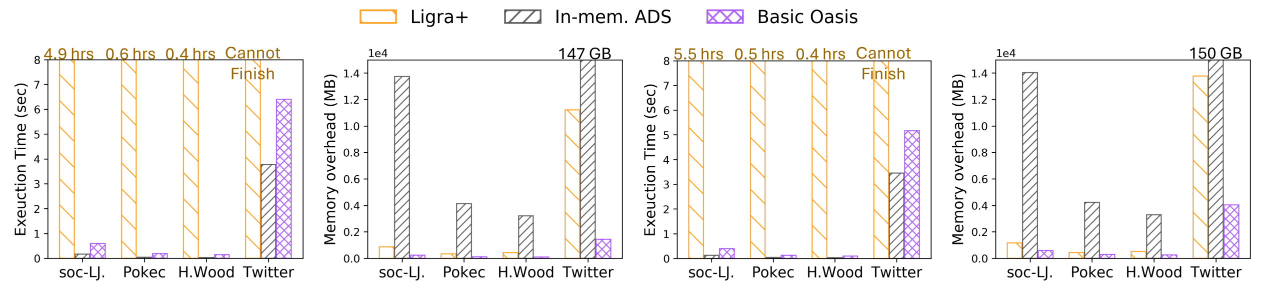
4.3 ADS Estimation

This section presents the evaluation of ADS estimation. As discussed in Section 2.2, there are seven graph applications that can be estimated by ADSs. However, despite the diversity of these applications, their ADS estimators work similarly. To process a single-ADS query, we load the required ADS into memory, and then perform operations to compute each entry within the loaded ADS. To process a dual-ADS query, we load the two required ADSs into memory and calculate an answer based on the entries from both ADSs. The main divergence between different ADS estimators lies in which operations are applied to calculate each ADS entry. Hence, due to the 12-page limit, this section presents the results of closeness centrality [10] and closeness similarity [11, 35]. These two are chosen because they are classic and well-known graph applications from each estimator category.

Following, to demonstrate the advantage of exploiting ADSs, Section 4.3.1 will evaluate in-memory graph system (i.e., Ligra+), in-memory ADS estimation, and the basic version of Oasis's estimation framework. Section 4.3.2 further justifies the impact of the optimizations proposed for improving Oasis's estimation framework.

4.3.1 Comparison between Exact Processing and ADS Estimation

Because Ligra+ is well known for being an efficient in-memory graph system for graph traversal, this section evaluates Ligra+ as the representative to compute exact answers. We also evaluate in-memory ADS estimation and out-of-core ADS estimation with Oasis. We conduct experiments on all the evaluated graphs with their ADSs of $k = 32$. For closeness centrality (i.e., single-ADS category), we randomly select 10% of vertices to be a set X, and calculate the centralities of all vertices within X based on their ADSs. For closeness



(a) Executoin time with closeness centrality. (b) Memory overhead with closeness centrality. (c) Executoin time with closeness similarity. (d) Memory overhead with closeness similarity.

Figure 8: Evaluation with closeness centrality and closeness similarity on all evaluated graphs.

similarity (i.e., dual-ADS category), we use the same set of vertices X but generate the queries in the following form: $(X[2i], X[2i+1])$, where $i \in [0,1,2,3,..]$ and $X[i]$ indicates the i^{th} vertex in the set X. Thus, with this setting, the amounts of needed data for both applications are the same.

Figure 8 shows the overall results. Figure 8(a) and Figure 8(b) present the execution time and memory overhead of running closeness centrality. Figure 8(c) and Figure 8(d) reveal the outcomes of running closeness similarity. It is clear to observe that Ligra+ generally takes a considerable time to finish all the queries, which is significantly slower than both in-memory and out-of-core ADS estimation by many orders of magnitudes. When the graph scale becomes larger (e.g., Twitter), Ligra+ cannot finish within a reasonable time. Through the completion time of a subset of queries, we can roughly estimate that Ligra+ will need 50 days to finish all the queries, which is unacceptable for real-world application. Thus, ADS estimation is proposed to tackle this issue by trading off a minor loss of accuracy. In-memory ADS estimation performs the best, at the cost of requiring a vast amount of memory to hold the entire ADSs in memory. In contrast, out-of-core ADS estimation (i.e., Oasis) is shown to be the most appealing solution. It enjoys the superb ADS speedup and requires only a small memory amount by loading the required ADSs into memory in an on-demand style. Specifically, although Oasis is averagely slower than in-memory ADS estimation by 2.9x, memory overhead can be saved by a significant 42x on average. It is worth noting that, using soc-LJ as an example, ADS construction and estimation with Oasis only take around 754.6 seconds in total for closeness centrality, compared to 4.9 hours for exact processing. This gap widens with more queries.

Finally, ADS estimation is proven to be theoretically accurate by many existing studies. Controlling the k value during ADS construction can increase or decrease the accuracy. Figure 9 reveals the accuracy of answer with k values from 4, 8, 16, 32, to 64 on the soc-LiveJournal graph in terms of closeness centrality and closeness similarity. The accuracy is measured by the normalized root-mean-square error between the estimated values and exact values. We can observe the accuracy of both closeness centrality and similarity basically increase as the k value increases, as suggested theoretically.

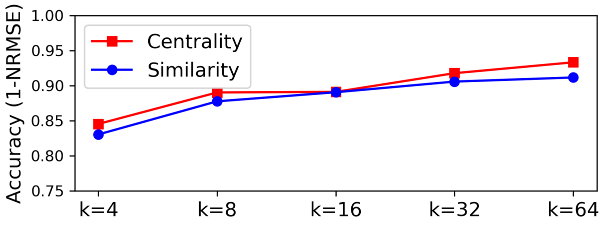


Figure 9: The accuracy of closeness centrality and closeness similarity on soc-LiveJournal graph with different k values.

4.3.2 Design Choices for Oasis ADS Estimations

This section aims to evaluate the effectiveness of the proposed designs, namely locality-aware query assignment and grid-based estimation, in optimizing the Oasis estimation framework. To better assess their performance impacts, these proposed designs will be tested with different input queries.

Locality-aware Query Assignment. To evaluate this design, we conduct experiments on closeness centrality application. We randomly select 10% of vertices to be a set X, and perform two different types of methods to measure the centrality on this set X. Specifically, we change the vertex weight function and distance decay function when measuring closeness centrality [10]. In other words, there are totally $2|X|$ of input queries; the front $|X|$ queries use one ADS estimator, and the back $|X|$ queries use the other ADS estimator.

Figure 10 shows the results. Since in-memory graph system is already shown to be slow in Section 4.3.1, we evaluate in-memory ADS estimation (denoted as In-mem. ADS), the basic Oasis's estimation framework (denoted as Baisc Oasis), and improved Oasis's estimation framework with locality-aware query assignment (denoted as Oasis w/ QA). We can observe that the design of locality-aware query assignment is helpful to further improve the performance of Basic Oasis. This is because we can increase the I/O utilization of the loaded ADSs with this design, and the average improvement is 40.7% on average with these input queries.

Grid-based Estimation. We conduct the experiments of closeness similarity and consider the input queries as fol-

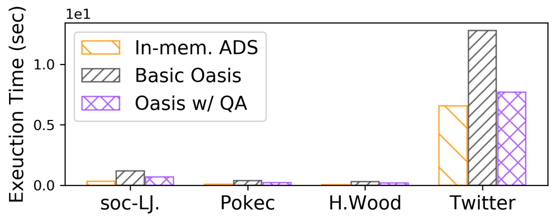


Figure 10: Evaluation with locality-aware query assignment.

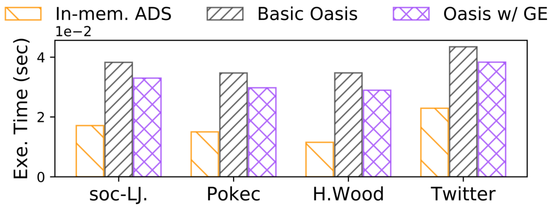


Figure 11: Evaluation with grid-based estimation.

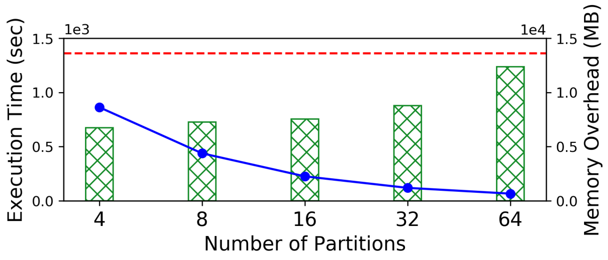


Figure 12: Evaluation on soc-LiveJournal graph with different number of partitions, $k = 32$, for ADS construction.

lows. First, we create three vertex sets, namely X_1, X_2, and X_3 by randomly selecting 10,000 vertices respectively from each P_1, P_2, and P_3, where P_i indicates the vertex ID range of the i^{th} partition. Next, we set the queries to calculate the closeness similarity of vertex pairs between X_1 and X_2. Specifically, those queries would be the following form: $(X_1[i], X_2[i])$, where $i \in [1,2,..,10,000]$ and $X_i[j]$ indicates the j^{th} vertex in the set X_i. Then, we set another batch of queries to calculate the closeness similarity between X_1 and X_3 as follow: $(X_1[i], X_3[i])$, where $i \in [1,2,..,10,000]$.

Figure 11 reveals the results. The results with grid-based estimation are denoted as Oasis w/ GE. Specifically, compared to Basic Oasis, Oasis w/ GE can improve the execution times by 13.8%, 14.2%, 16.7%, and 11.8% on soc-LiveJournal, Pokec, hollywood, and Twitter graphs, respectively. This is because the grid-based estimation can exploit the locality between processing the consecutive query grids. Thus, it performs better than the Basic Oasis.

4.4 Memory Scalability

This section studies the performance impact of applying different numbers of partitions to the graph for Oasis ADS construction. Specifically, we conduct ADS construction experiments on soc-LiveJournal graph with $k = 32$ as example, and divide the graph into 4, 8, 16, 32, and 64 partitions. Figure 12 reveals the results of execution time, memory overhead, and the sketch size (i.e., the horizontal dashed red line). It can be observed that the execution time and memory overhead present an inverse relationship. When $P = 64$, Oasis can perform ADS construction with only 668 MB, but it needs 1236 sec to complete. At the other end, when $P = 4$, Oasis needs 8638 MB memory overhead to finish the ADS construction process in 676 sec. This outcome is because a fewer number of partitions leads to a smaller amount of I/O involved in loading active ADSs. Further, the work balancing between threads can also be improved if a partition has a larger amount of work to do. Therefore, to achieve high performance, it is recommended that users minimize the number of partitions P under the constraint that the memory overhead does not exceed the available memory capacity of machine.

5 Related Work

Approximation has gained significant attention in graph processing due to the ability to trade accuracy for efficiency. Each scheme possesses unique advantages and assumptions for specific scenarios. One notable example is graph sparsification, which reduces the size of a graph by retaining only a subset of its vertices and edges so as to improve run time. Its goal is to preserve as many graph properties as possible based on different metrics [8, 17, 29]. For approximate graph mining tasks, both ASAP [18] and Arya [43] exploit neighborhood sampling method to estimate target pattern occurrences. [23] and [37] introduce approximate algorithms for shortest path and Pragerank, respectively. In contrast, Oasis does not propose a new approximate scheme. Instead, it builds upon a promising existing scheme, ADS. By incorporating storage devices and system-level optimizations, Oasis aims to make ADS practical and efficient for real-world applications.

6 Conclusion

This work introduces Oasis, the first out-of-core approximate graph system based on ADSs to manage ADSs with low memory and high efficiency. First, this work studies how to construct ADSs with a small memory amount, and proposes various system-level optimizations to decently improve its construction time. Next, an ADS estimation framework is presented, allowing users to implement their estimators easily and provides efficient runtime estimation. The evaluation results indicate that Oasis is outstanding for its efficient execution time, low memory usage, and high flexibility.

Acknowledgements

We sincerely thank our shepherd and all the anonymous reviewers for their valuable comments. This work is partially supported by The Research Grants Council (RGC) of Hong Kong SAR (Project No. CUHK14208521) and NSF 2413520. Any opinions, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the RGC and NSF.

References

[1] Takuya Akiba and Yosuke Yano. Compact and scalable graph neighborhood sketching. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 685–694, 2016.

[2] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[4] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Hyperanf: approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 625–634, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] Eliav Buchnik and Edith Cohen. Reverse ranking by graph structure: Model and scalable algorithms. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 51–62, 2016.

[7] Wei Chen, Yifei Yuan, and Li Zhang. Scalable influence maximization in social networks under the linear threshold model. In *2010 IEEE international conference on data mining*, pages 88–97. IEEE, 2010.

[8] Yuhan Chen, Haojie Ye, Sanketh Vedula, Alex Bronstein, Ronald Dreslinski, Trevor Mudge, and Nishil Talati. Demystifying graph sparsification algorithms in graph properties preservation. *Proc. VLDB Endow.*, 17(3):427–440, November 2023.

[9] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[10] Edith Cohen. All-distances sketches, revisited: Hip estimators for massive graphs analysis. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 88–99, 2014.

[11] Edith Cohen, Daniel Delling, Fabian Fuchs, Andrew V. Goldberg, Moises Goldszmidt, and Renato F. Werneck. Scalable similarity estimation in social networks: closeness, node labels, and random edge lengths. In *Proceedings of the First ACM Conference on Online Social Networks*, COSN '13, page 131–142, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*, pages 629–638, 2014.

[13] Edith Cohen and Haim Kaplan. Summarizing data using bottom-k sketches. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 225–234, New York, NY, USA, 2007. Association for Computing Machinery.

[14] Twitter dataset from WebGraph. `http://law.di.unimi.it/webdata/twitter-2010/`, 2010.

[15] Nan Du, Le Song, Manuel Gomez Rodriguez, and Hongyuan Zha. Scalable influence estimation in continuous-time diffusion networks. *Advances in neural information processing systems*, 26, 2013.

[16] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[17] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 71–80, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[19] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella,

Scott Shenker, and Ion Stoica. Bridging the gap: towards approximate graph analytics. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, October 2012. USENIX Association.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[23] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 308–321, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[25] Silviu Maniu and Bogdan Cautis. Network-aware search in social tagging applications: instance optimality versus efficiency. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, page 939–948, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Christopher R Palmer, Phillip B Gibbons, and Christos Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 81–90, 2002.

[27] Iván Palomares, Carlos Porcel, Luiz Pizzato, Ido Guy, and Enrique Herrera-Viedma. Reciprocal recommender systems: Analysis of state-of-art literature, challenges and opportunities towards social recommendation. *Information Fusion*, 69:103–127, 2021.

[28] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 721–732, New York, NY, USA, 2011. Association for Computing Machinery.

[30] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, 50(6), jan 2018.

[31] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412, 2015.

[32] Samsung 970 PRO SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[33] Shazia Tabassum, Fabiola SF Pereira, Sofia Fernandes, and João Gama. Social network analysis: An overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(5):e1256, 2018.

[34] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, volume 1, 2012.

[35] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, jan 2005.

[36] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[37] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. Fora: Simple and effective approximate single-source personalized pagerank. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 505–514, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Duncan J Watts. Networks, dynamics, and the small-world phenomenon. *American Journal of sociology*, 105(2):493–527, 1999.

[39] Sihem Amer Yahia, Michael Benedikt, Laks V. S. Lakshmanan, and Julia Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proc. VLDB Endow.*, 1(1):710–721, aug 2008.

[40] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. Seraph: Towards scalable and efficient fully-external graph computation via on-demand processing. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 373–387, Santa Clara, CA, February 2024. USENIX Association.

[41] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[42] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.

[43] Zeying Zhu, Kan Wu, and Zaoxing Liu. Arya: Arbitrary graph pattern mining with decomposition-based sampling. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1013–1030, Boston, MA, April 2023. USENIX Association.