Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing

Keval Vora, *University of California, Riverside;* Guoqing Xu, *University of California, Irvine;* Rajiv Gupta, *University of California, Riverside*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/vora

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing

Keval Vora
University of California, Riverside
kvora001@cs.ucr.edu

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Rajiv Gupta
University of California, Riverside
gupta@cs.ucr.edu

Abstract

Single-PC, disk-based processing of big graphs has recently gained much popularity. At the core of an efficient disk-based system is a well-designed partition structure that can minimize random disk accesses. All existing systems use *static partitions* that are created before processing starts. These partitions have static layouts and are loaded entirely into memory in every single iteration even though much of the edge data is not changed across many iterations, causing these unchanged edges to have *zero new impact* on the computation of vertex values.

This work provides a general optimization that removes this I/O inefficiency by employing *dynamic partitions* whose layouts are dynamically adjustable. Our implementation of this optimization in GraphChi — a representative out-of-core vertex-centric graph system — yielded speedups of 1.5—2.8× on six large graphs. Our idea is generally applicable to other systems as well.

1 Introduction

As graphs have become increasingly important in modern computing, developing systems to efficiently process large graphs has been a popular topic in the past few years. While a distributed system is a natural choice for analyzing large amounts of graph data, a recent trend initiated by GraphChi [14] advocates developing *out-of-core* support to process large graphs on a single commodity PC.

Out-of-core graph systems can be classified into two major categories based on their computation styles: *vertex-centric* and *edge-centric*. Vertex-centric computation, that originates from the "think like a vertex" model of Pregel [17], provides an intuitive programming interface. It has been used in most graph systems (*e.g.*, [8, 16, 14, 10]).

Recent efforts (*e.g.*, X-Stream [23] and GridGraph [31]) develop edge-centric computation (or a hybrid processing model) that streams edges into memory to perform vertex updates, exploiting locality for edges at the cost of random accesses to vertices. Since a graph often has many more edges than vertices, an edge-centric system improves performance by reducing random accesses to edges.

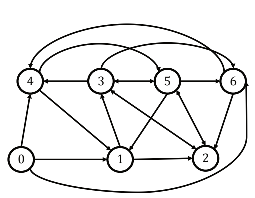
Observation At the heart of both types of systems is a well-designed, disk-based partition structure, along with an efficient *iterative, out-of-core* algorithm that accesses the partition structure to load and process a small portion of the graph at a time and write updates back to disk before proceeding to the next portion. As an example, GraphChi uses a *shard* data structure to represent a graph partition: the graph is split into multiple shards before processing; each shard contains edges whose target vertices belong to the same logical interval. X-Stream [23] partitions vertices into *streaming partitions*. GridGraph [31] constructs 2-dimensional edge blocks to minimize I/O.

Despite much effort to exploit locality in the partition design, existing systems use *static partition layouts*, which are determined before graph processing starts. In every single computational iteration, each partition is loaded entirely into memory, although a large number of edges in the partition are not strictly needed.

Consider an iteration in which the values for only a small subset of vertices are changed. Such iterations are very common when the computation is closer to convergence and values for many vertices have already stabilized. For vertices that are not updated, their values do not need to be pushed along their outgoing edges. Hence, the values associated with these edges remain the same. The processing of such edges (*e.g.*, loading them and reading their values) would be completely redundant in the next iteration because they make zero *new contribution* to the values of their respective target vertices.

Repeatedly loading these edges creates significant I/O inefficiencies, which impacts the overall graph processing performance. This is because data loading often takes a major portion of the graph processing time. As an example, over 50% of the execution time for PageRank is spent on partition loading, and this percentage increases further with the size of the input graph (*cf.* §2).

However, none of the existing out-of-core systems can eliminate the loading of such edges. Both GraphChi and X-Stream support *vertex scheduling*, in which vertices are scheduled to be processed for the next iteration if they have at least one incoming edge whose value is changed in the current iteration. While this approach reduces unnecessary computations, it cannot address the I/O inefficiency: although the value computation for certain vertices can be avoided, shards still need to be entirely loaded to give the system accesses to all vertices and edges. Similarly, GridGraph's 2-D partition structure remains static



(a) Example graph.

Shard 0

Src	Dst	Value
0	1	e_0
1	2	e_1
3	2	e_2
4	1	e_3
5	1	e_4
	2	e_5
6	2	e_6

Shard 1

Src	Dst	Value
0	4	e_7
1	3	e_8
2	3	e_9
	5	e_{10}
3	4	e_{11}
	5	e_{12}
4	5	e_{13}
5	3	e_{14}
6	4	e_{15}

Shard 2

Src	Dst	Value
0	6	e_{16}
3	6	e_{17}
5	6	e_{18}

(b) Shards representation.

Figure 1: An example graph partitioned into shards.

throughout computation regardless of the dynamic behavior of the algorithm — a partition has to be loaded entirely even if only one vertex in it needs to be computed.

Contributions This paper aims to reduce the above I/O inefficiency in out-of-core graph systems by exploring the idea of *dynamic partitions* that are created by omitting the edges that are not updated.

While our idea is applicable to *all* disk-based systems, in this work we focus on dynamically adjusting the *shard structure* used in GraphChi. We choose GraphChi as the starting point because: (1) it is a representative of extensively-used vertex-centric computation; (2) it is under active support and there are a large number of graph programs already implemented in it; and (3) its key algorithm has been incorporated into *GraphLab Create* [1], a commercial product of Dato, which performs both distributed and out-of-core processing. Hence, the goal of this paper is *not* to produce a brand new system that is faster than all existing graph systems, but instead, to show the *generality and effectiveness* of our optimization, which can be implemented in other systems as well.

Challenges Using dynamic partitions requires much more than recognizing unnecessary edges and removing them. There are two main technical challenges that need to be overcome.

The **first challenge** is how to perform vertex computation in the presence of missing edges that are eliminated during the creation of a dynamic partition. Although these edges make no impact on the forward computation, current programming/execution models all assume the presence of all edges of a vertex to perform value updates. To solve the problem, we begin with proposing a delay-based computation model (*cf.* §3) that *delays* the computation of a vertex with a missing edge until a special *shadow iteration* in which all edges are brought into memory from static partitions.

Since delays introduce overhead, to reduce delays, we further propose an *accumulation-based* programming/execution model (*cf.* §4) that enables *incremental vertex computation* by expressing computation in terms of contribution increments flowing through edges. As a result, vertices that *only have missing incoming edges* can be processed instantly without needing to be delayed because the increments from missing incoming edges are guaranteed to be zero. Computation for vertices with missing outgoing edges will still be delayed, but the number of such vertices is often very small.

The **second challenge** is how to efficiently build partitions on the fly. Changing partitions during processing incurs runtime overhead; doing so frequently would potentially make overheads outweigh benefits. We propose an additional optimization (*cf.* §5) that constructs dynamic partitions only during shadow iterations. We show, theoretically (*cf.* §5) and empirically (*cf.* §6), that this optimization leads to I/O reductions rather than overheads.

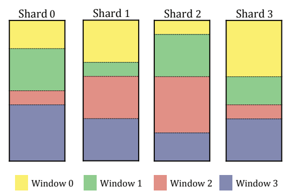
Summary of Results Our experiments with five common graph applications over six real graphs demonstrate that using dynamic shards in GraphChi accelerates the overall processing by up to 2.8× (on average 1.8×). While the accelerated version is still slower than X-Stream in many cases (*cf.* §6.3), this performance gap is reduced by 40% after dynamic partitions are used.

2 The Case for Dynamic Partitions

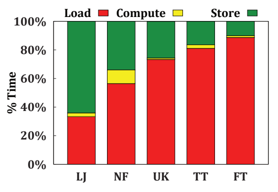
Background A graph $G = (V, E)$ consists of a set of vertices, V, and a set of edges E. The vertices are numbered from 0 to $|V| - 1$. Each edge is a pair of the form $e = (u, v)$, $u, v \in V$. u is the source vertex of e and v is e's destination vertex. e is an incoming edge for v and an outgoing edge for u. The vertex-centric computation model associates a data value with each edge and each vertex; at each vertex, the computation retrieves the values from its incoming edges, invokes an update function on these values to produce the new vertex value, and pushes this value out along its outgoing edges.

The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. There are many programming models developed to support vertex-centric computation, of which the `gather-apply-scatter` (GAS) model is perhaps the most popular one. We will describe the GAS model and how it is adapted to work with dynamic shards in §4. A vertex-centric system iterates around vertices to update their values until a global "fixed-point" is reached.

In GraphChi, the IDs of vertices are split into n disjoint logical intervals, each of which defines a shard. Each shard contains all edge entries whose *target vertices* belong to its defining interval. In other words, the shard only contains incoming edges of the vertices in the interval. As an illustration, given the graph shown in Figure 1a, the distribution of its edges across three shards is shown in Figure 1b where vertices 0–2, 3–5, and 6 are the three intervals that define the shards. If the source of an edge is the same as the previous edge, the edge's *src* field is empty. The goal of such a design is to reduce disk I/O by maximizing sequential disk accesses.

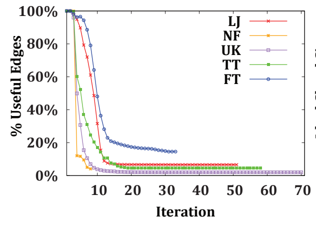


(a) Shards and sliding windows used in GraphChi.

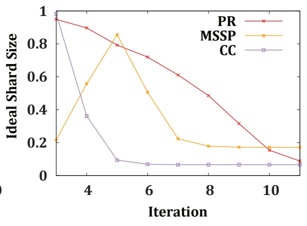


(b) Breakdown of PageRank execution time into load, compute, and store times.

Figure 2: An illustration of sliding windows and the PageRank execution statistics.



(a) Percentages of updated edges across iterations for the PageRank algorithm.



(b) Ideal shard sizes normalized w.r.t. the static shard size for LJ input graph.

Figure 3: Useful data in static shards.

Initial Shards

Shard 0

Src	Dst	Value
0	1	e_0
1	2	e_1
3	2	e_2
4	1	e_3
5	1	e_4
	2	e_5
6	2	e_6

Shard 1

Src	Dst	Value
0	4	e_7
1	3	e_8
2	3	e_9
	5	e_{10}
3	4	e_{11}
	5	e_{12}
4	5	e_{13}
5	3	e_{14}
6	4	e_{15}

Shard 2

Src	Dst	Value
0	6	e_{16}
3	6	e_{17}
5	6	e_{18}

Iteration 3

Shard 0

Src	Dst	Value
1	2	e_1
3	2	e_2
4	1	e_3
5	1	e_4
	2	e_5
6	2	e_6

Shard 1

Src	Dst	Value
1	3	e_8
2	3	e_9
	5	e_{10}
3	4	e_{11}
	5	e_{12}
4	5	e_{13}
5	3	e_{14}
6	4	e_{15}

Shard 2

Src	Dst	Value
3	6	e_{17}
5	6	e_{18}

Iteration 4

Shard 0

Src	Dst	Value
1	2	e_1
5	1	e_4
	2	e_5
6	2	e_6

Shard 1

Src	Dst	Value
1	3	e_8
2	3	e_9
	5	e_{10}
5	3	e_{14}
6	4	e_{15}

Shard 2

Src	Dst	Value
5	6	e_{18}

Iteration 5

Shard 0

Src	Dst	Value
3	2	e_2
4	1	e_3
5	1	e_4
	2	e_5

Shard 1

Src	Dst	Value
2	3	e_9
	5	e_{10}
3	4	e_{11}
	5	e_{12}
4	5	e_{13}
5	3	e_{14}

Shard 2

Src	Dst	Value
3	6	e_{17}
5	6	e_{18}

Figure 4: Dynamic shards for the example graph in Figure 1a created for iteration 3, 4 and 5.

Vertex-centric computation requires the presence of all (in and out) edges of a vertex to be in memory when the update is performed on the vertex. Since edges of a vertex may scatter to different shards, GraphChi uses an efficient parallel sliding window (PSW) algorithm to minimize random disk accesses while loading edges. First, edges in a shard s are sorted on their source vertex IDs. This enables an important property: while edges in s can come out of vertices from different intervals, those whose sources are in the same interval i are located contiguously in the shard defined by i.

When vertices v in the interval of s are processed, GraphChi only needs to load s (*i.e.*, memory shard, containing all v's incoming edges and part of v's outgoing edges) and a small block of edges from each other shard (*i.e.*, sliding shard, containing the rest of v's outgoing edges) – this brings into memory a *complete* set of edges for vertices belonging to the interval.

Figure 2a illustrates GraphChi's edge blocks. The four colors are used, respectively, to mark the blocks of edges in each shard whose sources belong to the four intervals defining these shards.

Motivation While the PSW algorithm leverages disk locality, it suffers from redundancy. During computation, a shard contains edges both with and without updated values. Loading the entire shard in every iteration involves wasteful effort of loading and processing edges that are guaranteed to make zero new contribution to the value computation. This effort is significant because (1) the majority of the graph processing cost comes from the loading phase, and (2) at the end of each iteration, there are a large number of edges whose values are unchanged. Figure 2b shows a breakdown of the execution times of PageRank in GraphChi for five real graphs, from the smallest *LiveJournal* (LJ) with 69M edges to *Friendster* (FT) with 2.6B edges. Further details for these input graphs can be found in Table 3.

In these experiments, the I/O bandwidth was fully utilized. Note that the data loading cost increases as the graph becomes larger – for *Friendster*, data loading contributes to over 85% of the total graph processing time. To understand if the impact of data loading is pervasive, we have also experimented with X-Stream [23]. Our results show that the *scatter* phase in X-Stream, which streams all edges in from disk, takes over 70% of the total processing time for PageRank on these five graphs.

To understand how many edges contain necessary data, we calculate the percentages of edges that have updated values across iterations. These percentages are shown in Figure 3a. The percentage of updated edges drops significantly as the computation progresses and becomes very low when the execution comes close to convergence. Significant I/O reductions can be expected if edges not updated in an iteration are completely eliminated from a shard and not loaded in the next iteration.

Figure 3b illustrates, for three applications PageRank (PR), MultipleSourceShortestPath (MSSP), and ConnectedComponents (CC), how the size of an *ideal* shard changes as computation progresses when the LiveJournal graph is processed. In each iteration, an ideal shard only contains edges that have updated values from the previous iteration. Observe that it is difficult to find a one-size-fits-all static partitioning because, for different algorithms, when and where useful data is produced changes dramatically, and thus different shards are needed.

Overview of Techniques The above observations strongly motivate the need for *dynamic shards* whose layouts can be adapted. Conceptually, for each static shard s and each iteration i in which s is processed, there exists a dynamic shard d_i that contains a subset of edges from s whose values are updated in i.

Figure 4 shows the dynamic shards created for Iteration 3, 4 and 5 during the processing of the example graph shown in Figure 1a. After the 2^{nd} iteration, vertex 0 becomes inactive, and hence, its outgoing edges to 4 and 6 are eliminated from the dynamic shards for the 3^{rd} iteration. Similarly, after the 3^{rd} iteration, the vertices 3 and 4 become inactive, and hence, their outgoing edges are eliminated from the shards for Iteration 4. In Iteration 4, the three shards contain only 10 out of a total of 19 edges. Since loading these 10 edges involves much less I/O than loading the static shards, significant performance improvement can be expected. To realize the benefits of dynamic shards by reducing I/O costs, we have developed three techniques:

(1) Processing Dynamic Shards with Delays – Dynamic shards are iteratively processed like static shards; however, due to missing edges in a dynamic shard, we may have to delay the computation of a vertex. We propose a delay based shard processing algorithm that places delayed vertices in an in-memory delay buffer and periodically performs *shadow iterations* that process the delayed requests by bringing in memory all edges for delayed vertices.

(2) Programming Model for Accumulation-Based Computation – Delaying the computation of a vertex if any of its edge is missing can slow the progress of the algorithm. To overcome this challenge we propose an *accumulation-based* programming model that expresses computation in terms of incremental contributions flowing through edges. This maximizes the processing of a vertex by allowing incremental computations to be performed using available edges and thus minimizes the impact of missing edges.

(3) Optimizing Shard Creation – Finally, we develop a practical strategy for balancing the cost of creating dynamic shards with their benefit from reduced I/O by adapting the frequency of shard creation and controlling when a shadow iteration is triggered.

In subsequent sections we discuss each of the above techniques in detail.

3 Processing Dynamic Shards with Delays

Although dynamic shard provides a promising solution to eliminating redundant loading, an immediate question is how to compute vertex values when edges are missing. To illustrate, consider the following graph edges: $u \rightarrow v \rightarrow w$. Suppose in one iteration the value of v is not changed, which means v becomes inactive and the edge $v \rightarrow w$ is not included in the dynamic shard created for the next iteration. However, the edge $u \rightarrow v$ is still included because a new value is computed for u and pushed out through the edge. This value will be reaching v in the next iteration. In the next iteration, the value of v changes as it receives the new contribution from $u \rightarrow v$. The updated value of v then needs to be pushed out through the edge $v \rightarrow w$, which is, however, not present in memory.

To handle missing edges, we allow a vertex to *delay its computation* if it has a missing edge. The delayed computations are batched together and performed in a special periodically-scheduled iteration called *shadow iteration* where all the (in- and out-) edges of the delayed vertices are brought in memory. We begin by discussing dynamic shard creation and then discuss the handling of missing edges.

Creating Dynamic Shards Each computational iteration in GraphChi is divided into three phases: load, compute, and write-back. We build dynamic shards at the end of the compute phase but before write-back starts. In the compute phase, we track the set of edges that receive new values from their source vertices using a *dirty* mark. During write-back, these dirty edges are written into new shards to be used in the next iteration. Evolving graphs can be supported by marking the dynamically added edges to be dirty and writing them into new dynamic shards.

The shard structure has two main properties contributing to the minimization of random disk accesses: (1) *disjoint edge partitioning* across shards and (2) *ordering of edges* based on source vertex IDs inside each shard. Dynamic shards also follow these two properties: since we do not change the logical intervals defined by static partitioning, the edge disjointness and ordering properties are preserved in the newly generated shards. In other words, for each static shard, we generate a dynamic shard, which contains a subset of edges that are stored in the same order as in the static shard. Although our algorithm is inexpensive, creating dynamic shards for every iteration incurs much time overhead and consumes large disk space. We will discuss an optimization in §5 that can effectively reduce the cost of shard creation.

Processing Dynamic Shards Similar to static shards, dynamic shards can be iteratively processed by invoking the user-defined update function on vertices. Although a dynamic shard contains fewer edges than its static counterpart, the logical interval to which the shard belongs is

Algorithm 1 Algorithm for a shadow iteration.

```
1:  $S = \{S_0, S_1, ..., S_{n-1}\}$: set of $n$ static shards
2:  $DS^i = \{DS^i_0, DS^i_1, ..., DS^i_{n-1}\}$: set of $n$ dynamic shards for
    Iteration $i$
3:  $DS = [DS^0, DS^1, ...]$: vector of dynamic shard sets for Iter-
    ation 0, 1, ...
4:  $V_i$: set of vertex IDs belonging to Interval $i$
5:  $DB$: delay buffer containing IDs of the delayed vertices
6:  $lastShadow$: ID of the last shadow iteration
7:  function SHADOW-PROCESSING(Iteration $ite$)
8:      for each Interval $k$ from 0 to $n$ do
9:          LOAD-ALL-SHARDS($ite, k$)
10:         par-for Vertex $v \in DB \cap V_k$ do
11:             UPDATE($v$) //user-defined vertex function
12:         end par-for
13:         produce $S'_k$ by writing updates to the static shard $S_k$
14:         create a dynamic shard $DS^{ite}_k$ for the next iteration
15:     end for
16:     remove $DS^{lastShadow} \dots DS^{ite-1}$
17:     $lastShadow \leftarrow ite$
18:     clear the delay buffer $DB$
19: end function
20:
21: function LOAD-ALL-SHARDS(Iteration $ite$, Interval $j$)
22:     LOAD-MEMORY-SHARD($S_j$)
23:     par-for Interval $k \in [0, n]$ do
24:         if $k \neq j$ then
25:             LOAD-SLIDING-SHARD($S_k$)
26:         end if
27:     end par-for
28:     for each Iteration $k$ from $lastShadow$ to $ite - 1$ do =
29:         LOAD-MEMORY-SHARD-AND-OVERWRITE($DS^k_j$)
30:         par-for Interval $i \in [0, n]$ do
31:             if $i \neq j$ then
32:                 LOAD-SLIDING-SHARD-AND-OVERWRITE($DS^k_i$)
33:             end if
34:         end par-for
35:         if $k = ite - 1$ then
36:             MARK-DIRTY-EDGES( )
37:         end if
38:     end for
39: end function
```

not changed, that is, the numbers of vertices to be updated when a dynamic shard and its corresponding static shard are processed are the same. However, when a dynamic shard is loaded, it contains only *subset* of edges for vertices in its logical interval. To overcome this challenge, we *delay* the computation of a vertex if it has a missing (incoming or outgoing) edge. The delayed vertices are placed in an in-memory delay buffer. We periodically process these delayed requests by bringing in memory all the incoming and outgoing edges for the vertices in the buffer. This is done in a special *shadow iteration* where static shards are also loaded and updated.

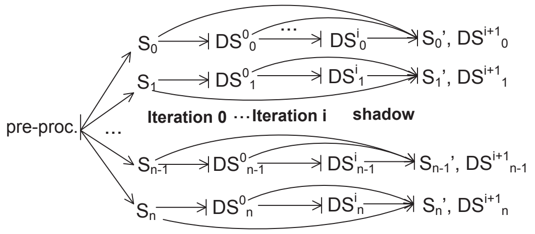


Figure 5: Processing using dynamic shards.

Since a normal iteration has similar semantics as those of iterations in GraphChi, we refer the interested reader to [14] for its details. Here we focus our discussion on shadow iterations. The algorithm of a shadow iteration is shown in Algorithm 1. A key feature of this algorithm is that it loads the static shard (constructed during pre-processing) to which each vertex in the delay buffer belongs to bring into memory all of its incoming and outgoing edges for the vertex computation. This is done by function LOAD-ALL-SHARDS shown in Lines 21–39 (invoked at Line 9).

However, only loading static shards would not solve the problem because they contain out-of-date data for edges that have been updated recently. The most recent data are scattered in the dynamic shards $DS^{lastShadow} \dots DS^{ite-1}$ where *lastShadow* is the ID of the last shadow iteration and *ite* is the ID of the current iteration. As an example, consider Shard 0 in Figure 4. At the end of iteration 5, the most recent data for the edges $1 \rightarrow 2$, $3 \rightarrow 2$, and $0 \rightarrow 1$ are in DS^4_0, DS^5_0, and S_0, respectively, where DS^i_j represents the dynamic shard for interval j created for iteration i, and S_j denotes the static shard for interval j.

To guarantee that most recent updates are retrieved in a shadow iteration, for each interval j, we sequentially load its static shard S_j (Line 22) and dynamic shards created since the last shadow iteration $DS^{lastShadow} \dots DS^{ite-1}$ (Line 29), and let the data loaded later *overwrite* the data loaded earlier for the same edges. LOAD-ALL-SHARDS implements GraphChi's PSW algorithm by loading (static and dynamic) memory shards entirely into memory (Lines 22 and 29) and a sliding window of edge blocks from other (static and dynamic) shards (Lines 23–27 and 30–34). If k becomes the ID of the iteration right before the shadow iteration (Lines 35–37), we mark dirty edges to create new dynamic shards for the next iteration (Line 14).

After the loop at Line 8 terminates, we remove all the intermediate dynamic shards (Line 16) and set *lastShadow* to *ite* (Line 17). These shards are no longer needed, because the static shards are already updated with the most recent values in this iteration (Line 13). One can think of static shards as "checkpoints" of the computation and dynamic shards as intermediate "increments" to the most recent checkpoint. Finally, the delay buffer is cleared.

Figure 5 illustrates the input and output of each computational iteration. Static shards $S_0 \dots S_n$ are statically constructed. Each regular iteration i produces a set of dynamic shards $DS_0^i \dots DS_n^i$, which are fed to the next iteration. A shadow iteration loads all static shards and intermediate dynamic shards, and produces (1) updated static shards $S_0' \dots S_n'$ and (2) new dynamic shards $DS_0^{i+1} \dots DS_n^{i+1}$ to be used for the next iteration.

It may appear that the delay buffer can contain many vertices and consume much memory. However, since the amount of memory needed to represent an incoming edge is higher than that to record a vertex, processing dynamic shards with the delay buffer is actually more memory-efficient compared to processing static shards where all edges are available.

Delaying a vertex computation when *any* of its edge is missing can cause too many vertices to be delayed and negatively impact the the computation progress. For example, when running PageRank on *UKDomain*, *Twitter*, and *Friendster* graphs, immediately after the dynamic shards are created, 64%, 70%, and 73% of active vertices are delayed due to at least one missing incoming or outgoing edge. Frequently running shadow iterations may get data updated quickly at the cost of extra overhead, while doing so infrequently would reduce overhead but slow down the convergence. Hence, along with dynamically capturing the set of edges which reflect change in values, it is important to modify the computation model so that it maximizes computation performed using available values.

§4 presents an optimization for our delay-based computation to limit the number of delayed computations. The optimization allows a common class of graph algorithms to perform vertex computation if a vertex only has missing incoming edges. While the computation for vertices with missing outgoing edges still need to be delayed, the number of such vertices is much smaller, leading to significantly reduced delay overhead.

4 Accumulation-based Computation

This section presents an *accumulation-based* programming/execution model that expresses computation in terms of *incremental contributions* flowing through edges. Our insight is that if a vertex is missing an incoming edge, then the edge is guaranteed to provide zero *new contribution* to the vertex value. If we can design a new model that performs updates based on *contribution increments* instead of actual contributions, the missing incoming edge can be automatically treated as *zero* increment and the vertex computation can be performed without delay.

We discuss our approach based on the popular Gather-Apply-Scatter (GAS) programming model [14, 16, 8]. In the GAS model, vertex computation is divided in three distinct phases: the `gather` phase reads incoming edges and produces an aggregated value using a user-defined aggregation function; this value is fed to the `apply` phase to compute a new value for a vertex; in the `scatter` phase, the value is propagated along the outgoing edges.

4.1 Programming Model

Our accumulation-based model works for a common class of graph algorithms whose GAS computation is *distributive* over aggregation. The user needs to program the GAS functions in a slightly different way to propagate *changes in values* instead of actual values. In other words, the semantics of vertex data remains the same while data on each edge now encodes the delta between the old and the new value of its source vertex. This semantic modification relaxes the requirement that all incoming edges of a vertex have to be present to perform vertex computation.

The new computation semantics requires minor changes to the GAS programming model. (1) `Extract` the gathered value using the old vertex value. This step is essentially an inverse of the `apply` phase that uses its output (*i.e.*, vertex value) to compute its input (*i.e.*, aggregated value). (2) `Gather` edge data (*i.e.*, from present incoming edges) and aggregate it together with the output of `extract`. Since this output represents the contributions of the previously encountered incoming edges, this step incrementally adds new contributions from the present incoming edges to the old contributions. (3) `Apply` the new vertex value using the output of `gather`. (4) `Scatter` the difference between the old and the new vertex values along the outgoing edges.

To turn a GAS program into a new program, one only needs to add an `extract` phase in the beginning that uses a vertex value v to compute *backward* the value g gathered from the incoming edges of the vertex at the time v was computed. g is then aggregated with a value gathered from the present incoming edges to compute a new value for the vertex. To illustrate, consider the PageRank algorithm that has the following GAS functions:

$$\begin{aligned}
&\text{[GATHER]} && sum \leftarrow \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && v.pr \leftarrow (0.15 + 0.85 * sum) / \\
& && \quad v.numOutEdges \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow v.pr
\end{aligned}$$

Adding the `extract` phase produces:

$$\begin{aligned}
&\text{[EXTRACT]} && oldsum \leftarrow (v.pr * v.numOutEdges \\
& && \quad - 0.15)/0.85 \\
&\text{[GATHER]} && newsum \leftarrow oldsum + \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && newpr \leftarrow (0.15 + 0.85 \times newsum) / \\
& && \quad v.numOutEdges; \\
& && oldpr \leftarrow v.pr; \;\; v.pr \leftarrow newpr \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow newpr - oldpr
\end{aligned}$$

In this example, `extract` reverses the PageRank computation to obtain the old aggregated value *oldsum*, on top of which the new contributions of the present incoming edges are added by `gather`. `Apply` keeps its original semantics and computes a new PageRank value. Before this new value is saved on the vertex, the delta between the old and new is computed and propagated along the outgoing edges in `scatter`.

An alternative way to implement the accumulation-based computation is to save the value gathered from incoming edges on each vertex (*e.g.*, *oldsum*) together with the vertex value so that we do not even need the `extract` phase. However, this approach doubles the size of vertex data which also negatively impacts the time cost due to the extremely large numbers of vertices in real-world graphs. In fact, the `extract` phase does not create extra computation in most cases: after simplification and redundancy elimination, the PageRank formulas using the traditional GAS model and the accumulation-based model require the same amount of computation:

$$pr = \begin{cases} 0.15 + 0.85 \times sum & \ldots\text{traditional} \\ v.pr + 0.85 \times sum & \ldots\text{accumulation-based} \end{cases}$$

Impact on the Delay Buffer Since the contribution of each incoming edge can be incrementally added onto the vertex value, this model does not need the presence of all incoming edges to compute vertex values. Hence, it significantly decreases the number of vertices whose computation needs to be delayed, reducing the need to frequently run shadow iterations.

If a vertex has a missing outgoing edge, delay is needed. To illustrate, consider again the $u \rightarrow v \rightarrow w$ example in the beginning of §3. Since the edge $v \rightarrow w$ is missing, although v gets an updated value, the value cannot be pushed out. We have to delay the computation until a shadow iteration in which $v \rightarrow w$ is brought into memory. More precisely, v's `gather` and `apply` can still be executed right away; only its `scatter` operation needs to be delayed, because the target of the `scatter` is unknown due to the missing outgoing edge.

Hence, for each vertex, we execute `gather` and `apply` instantly to obtain the result value r. If the vertex has a missing outgoing edge, the vertex is pushed into the delay buffer together with the value r. Each entry in the buffer now becomes a vertex-value pair. In the next shadow iteration, when this missing edge is brought into memory, r will be pushed through the edge and be propagated.

Since a vertex with missing outgoing edges can be encountered multiple times before a shadow iteration is scheduled, the delay buffer may contain multiple entries for the same vertex, each with a different delta value. Naïvely propagating the most recent increment is incorrect due to the accumulative nature of the model; the consideration of *all* the entries for the vertex is thus required. Hence, we require the developer to provide an additional *aggregation function* that takes as input an ordered list of all delta values for a vertex recorded in the delay buffer and generates the final value that can be propagated to its outgoing edges (details are given in §4.2).

Although our programming model exposes the `extract` phase to the user, not all algorithms need this phase. For example, algorithms such as ShortestPath and ConnectedComponents can be easily coded in a traditional way, that is, edge data still represent actual values (*i.e.*, paths or component IDs) instead of value changes. This is because in those algorithms, vertex values are in *discrete domains* and `gather` is done by monotonically selecting a value from one incoming edge instead of accumulating values from all incoming edge values. For instance, ShortestPath and ConnectedComponents use selection functions (*min*/*max*) to aggregate contributions of incoming edges.

To make the differences between algorithm implementations transparent to the users, we allow users to develop normal GAS functions without thinking about what data to push along edges. The only additional function the user needs to add is `extract`. Depending on whether `extract` is empty, our system *automatically* determines the meaning of edge data and how it is pushed out.

4.2 Model Applicability and Correctness

It is important to understand precisely what algorithms can and cannot be implemented under the accumulation-based model. There are three important questions to ask about applicability: (1) what is the impact of *incremental computation* on graph algorithms, (2) what is the impact of *delay* on those algorithms, and (3) is the computation still correct when vertex updates are delayed?

Impact of Incremental Computation An algorithm can be correctly implemented under our accumulation-based model if the composition of its `apply` and `gather` is distributive on some aggregation function. More formally, if vertex v has n incoming edges $e_1, e_2, \ldots e_n$, v's computation can be expressed under our accumulation-based model iff there exists an aggregation function[1] f *s.t.*

$$\texttt{apply}(\texttt{gather}(e_1,\ldots,e_n)) = f(\texttt{apply}(\texttt{gather}(e_1)),\ldots,\texttt{apply}(\texttt{gather}(e_n)))$$

For most graph algorithms, we can easily find a function f on which their computation is distributive. Table 1 shows a list of 24 graph algorithms studied in recent graph papers and our accumulation-based model works for all but two. For example, one of these two algorithms is GraphColoring, where the color of a vertex is determined

[1]The commutative and associative properties from `gather` get naturally lifted to the aggregation function f.

	V/E	**Iteration**				
		0	**1**	**2**	**3**	**4 (Shadow)**
	u	$[0,I_u]$	$[I_u,I_u]$	$[I_u,a]$	$[a,b]$	$[b,x]$
No Delay	$u \rightarrow v$	$[0,I_u]$	$[I_u,0]$	$[0,a-I_u]$	$[a-I_u,b-a]$	$[b-a,x-b]$
	v	$[0,I_v]$	$[I_v,\texttt{AP}(\texttt{EX}(I_v)+I_u)]$	$[\texttt{AP}(\texttt{EX}(I_v)+I_u), \texttt{AP}(\texttt{EX}(I_v)+I_u)]$	$[\texttt{AP}(\texttt{EX}(I_v)+I_u), \texttt{AP}(\texttt{EX}(I_v)+a)]$	$[\texttt{AP}(\texttt{EX}(I_v)+a), \texttt{AP}(\texttt{EX}(I_v)+b)]$
Delay	$u \rightarrow v$	$[0,I_u]$	$[I_u,0]$	Missing	Missing	$[b-I_u,x-b]$
	v	$[0,I_v]$	$[I_v, \texttt{AP}(\texttt{EX}(I_v)+I_u)]$	$[\texttt{AP}(\texttt{EX}(I_v)+I_u), \texttt{AP}(\texttt{EX}(I_v)+I_u)]$	$[\texttt{AP}(\texttt{EX}(I_v)+I_u), \texttt{AP}(\texttt{EX}(I_v)+I_u)]$	$[\texttt{AP}(\texttt{EX}(I_v)+I_u), \texttt{AP}(\texttt{EX}(I_v)+b)]$

Table 2: A comparison between PageRank executions with and without delays under the accumulation-based model; for each vertex and edge, we use a pair [*a*, *b*] to report its pre- (*a*) and post-iteration (*b*) value. Each vertex *u* (*v*) has a value 0 before it receives an initial value I_u (I_v) in Iteration 0; `EX` and `AP` represent function `Extract` and `Apply`, respectively.

Algorithms	**Aggr. Func.** f
Reachability, MaxIndependentSet	or
TriangleCounting, SpMV, PageRank, HeatSimulation, WaveSimulation, NumPaths	sum
WidestPath, Clique	max
ShortestPath, MinmialSpanningTree, BFS, ApproximateDiameter, ConnectedComponents	min
BeliefPropagation	product
BetweennessCentrality, Conductance, NamedEntityRecognition, LDA, ExpectationMaximization, AlternatingLeastSquares	user-defined aggregation function
GraphColoring, CommunityDetection	N/A

Table 1: A list of algorithms used as subjects in the following papers and their aggregation functions if implemented under our model: GraphChi [14], GraphLab [16], ASPIRE [25], X-Stream [23], GridGraph [31], GraphQ [27], GraphX [9], PowerGraph [8], Galois [19], Ligra [24], Cyclops [5], and Chaos [22].

by the colors of all its neighbors (coming through its incoming edges). In this case, it is impossible to compute the final color by applying `gather` and `apply` on different neighbors' colors separately and aggregating these results. For the same reason CommunityDetection cannot be correctly expressed as an incremental computation.

Once function f is found, it can be used to aggregate values from multiple entries of the same vertex in the delay buffer, as described earlier in §4.1. We provide a set of built-in f from which the user can choose, including *and*, *or*, *sum*, *product*, *min*, *max*, *first*, and *last*. For instance, PageRank uses *sum* that produces the final delta by summing up all deltas in the buffer, while ShortestPath only needs to compute the minimum of these deltas using *min*. The user can also implement her own for more complicated algorithms that perform numerical computations.

For graph algorithms with non-distributive `gather` and `apply`, using dynamic partitions has to delay computation for a great number of vertices, making overhead outweigh benefit. In fact, we have implemented GraphColoring in our system and only saw slowdowns in the experiments. Hence, our optimization provides benefit only for distributive graph algorithms.

Impact of Delay To understand the impact of delay, we draw a connection between our computation model with the staleness-based (*i.e.*, relaxed consistency) computation model [25, 6]. The staleness-based model allows computation to be performed on stale values but guarantees correctness by ensuring that all updates are visible at some point during processing (by either using refresh or imposing a staleness upper-bound). This is conceptually similar to our computation model with delays: for vertices with missing outgoing edges, their out-neighbors would operate on stale values until the next shadow iteration.

Since a shadow iteration "refreshes" all stale values, the frequency of performing these shadow iterations bounds the maximum staleness of edge values. Hence, any algorithm that can correctly run under the relaxed consistency model can also safely run under our model. Moreover, the frequency of shadow iterations has no impact on the correctness of such algorithms, as long as they do occur and flush the delayed updates. In fact, all the algorithms in Table 1 would function correctly under our delay-based model. However, their performance can be degraded if they cannot employ incremental computation.

Delay Correctness Argument While our delay-based model shares similarity with the staleness-based model, the correctness of a specific algorithm depends on the aggregation function used for the algorithm. Here we provide a correctness argument for the aggregation functions we developed for the five algorithms used in our evaluation: PageRank, BeliefPropagation, HeatSimulation, ConnectedComponents, and MultipleSourceShortestPath; similar arguments can be used for other algorithms in Table 1.

We first consider our implementation of PageRank that propagates changes in page rank values along edges. Since BeliefPropagation and HeatSimulation perform similar computations, their correctness can be reasoned in the same manner. For a given edge $u \rightarrow v$, Table 2 shows, under the accumulation-based computation, how the values carried by vertices and edges change across iterations with and without delays.

We assume that each vertex u (v) has a value 0 before it is assigned an initial value I_u (I_v) in Iteration 0 and vertex v has only one incoming edge $u \rightarrow v$. At the end of Iteration 0, both vertices have their initial values because the edge does not carry any value in the beginning. We further assume that in Iteration 1, the value of vertex u does not change. That is, at the end of the iteration, u's value is still I_u and, hence, the edge will not be loaded in Iteration 2 and 3 under the delay-based model.

We compare two scenarios in which delay is and is not enabled and demonstrate that the same value is computed for v in both scenarios. Without delay, the edge value in each iteration always reflects the change in u's values. v's value is determined by the four functions described earlier. For example, since the value carried by the edge at the end of Iteration 0 is I_u, v's value in Iteration 1 is updated to `apply(gather(extract(`I_v`), `I_u`)))`. As `gather` is `sum` in PageRank, this value reduces to `AP(EX(`I_v`) + `I_u`)`. In Iteration 2, the value from the edge is 0 and thus v's value becomes `AP(EX(AP(EX(`I_v`) + `I_u`)) + 0)`. Because `EX` is an inverse function of `AP`, this value is thus still `AP(EX(`I_v`) + `I_u`)`. Using the same calculation, we can easily see that in Iteration 4 v's value is updated to `AP(EX(`I_v`) + `b`)`.

With delay, the edge will be missing in Iteration 2 and 3, and hence, we add two entries (u, $a - I_u$) and (u, $b - a$) into the delay buffer. During the shadow iteration, the edge is loaded back into memory. The aggregation function `sum` is then applied on these two entries, resulting in value $b - I_u$. This value is pushed along $u \rightarrow v$, leading to the computation of the following value for v:

$$\begin{aligned}&\texttt{AP}(\texttt{EX}(\texttt{AP}(\texttt{EX}(I_v) + I_u)) + (b - I_u))\\ \Rightarrow\ &\texttt{AP}(\texttt{EX}(I_v) + I_u + b - I_u)\\ \Rightarrow\ &\texttt{AP}(\texttt{EX}(I_v) + b)\end{aligned}$$

which is the same as the value computed without delay.

This informal correctness argument can be used as the base case for a formal proof by induction on iterations. This proof is omitted from the paper due to space limitations. Although we have one missing edge in this example, the argument can be easily extended to handle multiple missing edges since the `gather` function is associative.

For ShortestPaths and ConnectedComponents, they do not have an `extract` function and their contributions are gathered by the selection function *min*. Since a dynamic shard can never have edges that are not part of its corresponding static shard, vertex values (*e.g.*, representing path and component IDs) in the presence of missing edges are always greater than or equal to their actual values. It is thus easy to see that the aggregation function *min* ensures that during the shadow iteration the value a of each vertex will be appropriately overridden by the minimum value b of the delayed updates for the vertex if $b \leq a$.

4.3 Generalization to Edge-Centricity

Note that the dynamic partitioning techniques presented in this work can be easily applied to edge-centric systems. For example, X-Stream [23] uses an unordered edge list and a scatter-gather computational model, which first streams in the edges to generate updates, and then streams in the generated updates to compute vertex values. To enable dynamic partitioning, dynamic edge lists can be constructed based on the set of changed vertices from the previous iterations. This can be done during the scatter phase by writing to disk the required edges whose vertices are marked dirty.

Hence, later iterations will stream in smaller edge lists that mainly contain the necessary edges. Similarly to processing dynamic shards, computations in the presence of missing edges can be delayed during the gather phase when the upcoming scatter phase cannot stream in the required edges. These delayed computations can be periodically flushed by processing them during shadow iterations in which the original edge list is made available.

GridGraph [31] is a recent graph system that uses a similar graph representation as used in GraphChi. Hence, our shard-based techniques can be applied directly to partitions in GridGraph. As GridGraph uses very large static partitions (that can accommodate tens of millions of edges), larger performance benefit may be seen if our optimization is added. Dynamic partitions can be generated when edges are streamed in; computation that needs to be delayed due to missing edges can be detected when vertices are streamed in.

5 Optimizing Shard Creation

To maximize net gains, it is important to find a sweet spot between the cost of creating a dynamic shard and the I/O reduction it provides. This section discusses an optimization and analyzes its performance benefit.

5.1 Optimization

Creating a dynamic shard at each iteration is an overkill because many newly created dynamic shards provide only small additional reduction in I/O that does not justify the cost of creating them. Therefore, we create a new dynamic shard after several iterations, allowing the creation overhead to be easily offset by the I/O savings.

Furthermore, to maximize edge reuse and reduce delay frequencies, it is useful to include into dynamic shards edges that may be used in *multiple* subsequent iterations. We found that using shadow iterations to create dynamic shards strikes a balance between I/O reduction and overhead of delaying computations – new shards are created only during shadow iterations; we treat edges that were updated after the previous shadow iteration as dirty and include them all in the new dynamic shards. The intuition here is that by considering an "iteration window" rather than one single iteration, we can accurately identify edges whose data have truly stabilized, thereby simultaneously reducing I/O and delays.

The first shadow iteration is triggered when the percentage of updated edges p in an iteration drops below a threshold value. The frequency of subsequent shadow iterations depends upon the size of the delay buffer d — when the buffer size exceeds a threshold, a shadow iteration is triggered. Hence, the frequency of shard creation is adaptively determined, in response to the progress towards convergence. We used $p = 30\%$ and $d = 100$KB in our experiments and found them to be effective.

5.2 I/O Analysis

We next provide a rigorous analysis of the I/O costs. We show that the overhead of shard loading in shadow iterations can be easily offset from the I/O savings in regular non-shadow iterations. We analyze the I/O cost in terms of the number of data blocks transferred between disk and memory. Let b be the size of a block in terms of the number of edges and E be the edge set of the input graph. Let AE_i (*i.e.*, active edge set) represent the set of edges in the dynamic shards created for iteration i. Here we analyze the cost of regular iterations and shadow iterations separately for iteration i.

During regular iterations, processing is done using the static shards in the first iteration and most recently created dynamic shards during later iterations. Each edge can be read at most twice (*i.e.*, when its source and target vertices are processed) and written once (*i.e.*, when the value of its source vertex is pushed along the edge). Thus,

$$C_i \leq \begin{cases} \frac{3|E|}{b} & \text{with static shards} \\ \frac{3|AE_i|}{b} & \text{with dynamic shards} \end{cases} \quad (1)$$

In a shadow iteration, the static shards and all intermediate dynamic shards are read, the updated edges are written back to static shards, and a new set of dynamic shards are created for the next iteration. Since we only append edges onto existing dynamic shards in regular iterations, there is only one set of dynamic shards between any consecutive shadow iterations. Hence, the I/O cost is:

$$C_i \leq \frac{3|E|}{b} + \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b} \quad (2)$$

where AE_{LS} is the set of edges in the dynamic shards created by the last shadow iteration. Clearly, C_i is larger than the cost of static shard based processing (*i.e.*, $\frac{3|E|}{b}$).

Eq. 1 and Eq. 2 provide a useful insight on how the overhead of a shadow iteration can be amortized across regular iterations. Based on Eq. 2, the extra I/O cost of a shadow iteration over a regular static-shard-based iteration is $\frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$. Based on Eq. 1, the I/O saving achieved by using dynamic shards in a regular iteration is $(\frac{3|E|}{b} - \frac{3|AE_i|}{b})$.

We assume that d shadow iterations have been performed before the current iteration i and hence, the frequency of shadow iterations is $\frac{i}{d}$ (for simplicity, we assume i is multiple of d). This means, shadow iteration occurs once every $\frac{i}{d} - 1$ regular iterations.

In order for the overhead of a shadow iteration to be wiped off by the savings in regular iterations, we need:

$$(\frac{i}{d} - 1) \times (\frac{3|E|}{b} - \frac{3|AE_i|}{b}) \geq \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$$

After simplification, we need to show:

$$(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_{LS}| \geq 0 \quad (3)$$

Since AE_{LS} is the set of edges in the dynamic shards before Iteration i, we have $|AE_{LS}| \leq |AE_i|$ as we only append edges after that shadow iteration. We thus need to show:

$$(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_i| \geq 0$$
$$\Longrightarrow \frac{|E|}{|AE_i|} \geq \frac{\frac{i}{d}}{\frac{i}{d} - 1}$$

The above inequality typically holds for any frequency of shadow iterations $\frac{i}{d} > 1$. For example, if the frequency of shadow iterations $\frac{i}{d}$ is 3, $\frac{|E|}{|AE_i|} \geq 1.5$ means that as long as the total size of static shards is 1.5 times larger than the total size of (any set of) dynamic shards, I/O efficiency can be achieved by our optimization. As shown in Figure 3a, after about 10 iterations, the percentage of updated edges in each iteration goes below 15%. Although unnecessary edges are not removed in each iteration, the ratio between $|E|$ and $|AE_i|$ is often much larger than 1.5, which explains the I/O reduction from a theoretical viewpoint.

6 Evaluation

Our evaluation uses five applications including PageRank (PR) [20], MultipleSourceShortestPath (MSSP), Belief-Propagation (BP) [11], ConnectedComponents (CC) [30], and HeatSimulation (HS). They belong to different domains such as social network analysis, machine learning, and scientific simulation. They were implemented using our accumulation-based GAS programming model. Six real-world graphs, shown in Table 3, were chosen as inputs for our experiments.

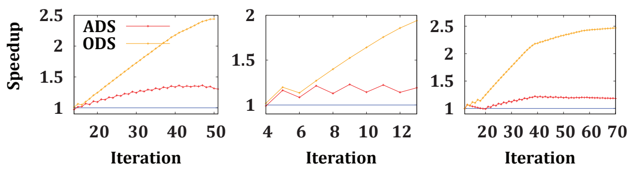
All experiments were conducted on an 8-core commodity Dell machine with 8GB main memory, running Ubuntu 14.04 kernel 3.16, a representative of low-end PCs regular users have access to. Standard Dell 500GB 7.2K RPM HDD and Dell 400GB SSD were used as secondary

Inputs	Type	#Vertices	#Edges	PMSize	#SS
LiveJournal (LJ) [2]	Social Network	4.8M	69M	1.3GB	3
Netflix (NF) [3]	Recomm. System	0.5M	99M	1.6GB	20
UKDoman (UK) [4]	Web Graph	39.5M	1.0B	16.9GB	20
Twitter (TT) [13]	Social Network	41.7M	1.5B	36.3GB	40
Friendster (FT) [7]	Social Network	68.3M	2.6B	71.6GB	80
YahooWeb (YW) [28]	Web Graph	1.4B	6.6B	151.3GB	120

Table 3: Input graphs used; **PMSize** and **SS** report the peak in-memory size of each graph structure (without edge values) and the number of static shards created in GraphChi, respectively. The in-memory size of a graph is measured as the maximum memory consumption of a graph across the five applications; LJ and NF are relatively small graphs while UK, TT, FT, YW are **billion-edge graphs larger than the 8GB memory size**; YW is the **largest real-world graph** publicly available; all graphs have highly skewed power-law degree distributions.

G	Version	PR	BP	HS	MSSP	CC
	BL	630	639	905	520	291
LJ	**ADS**	483	426	869	535	296
	ODS	258	383	321	551	263
	BL	189	876	238	1,799	190
NF	**ADS**	174	597	196	1,563	177
	ODS	158	568	164	1,436	178
	BL	31,616	19,486	21,620	74,566	14,346
UK	**ADS**	23,332	15,593	35,200	76,707	14,742
	ODS	14,874	14,227	12,388	67,637	12,814
	BL	83,676	47,004	75,539	109,010	22,650
TT	**ADS**	61,994	38,148	67,522	97,132	21,522
	ODS	47,626	28,434	30,601	84,058	21,589
	BL	130,928	100,690	159,008	146,518	50,762
FT	**ADS**	85,788	84,502	176,767	143,798	50,831
	ODS	87,112	51,905	63,120	127,168	42,956

Table 4: A comparison on execution time (seconds) among **Baseline** (BL), **ADS**, and **ODS**.



(a) PR on LJ. (b) BP on FT. (c) HS on TT.

Figure 6: Speedups achieved per iteration.

storage, both of which were connected via SATA 3.0Gb/s interface. File system caches were flushed before running experiments to make different executions comparable.

Two relatively small graphs LJ and NF were chosen to understand the scalability trend of our technique. The other four graphs UK, TT, FT, and YW are larger than memory by 2.4×, 5.2×, 10.2×, and 21.6× respectively.

6.1 Overall Performance

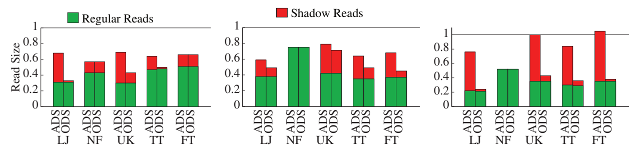
We compared our modified GraphChi extensively with the **Baseline (BL)** GraphChi that processes static shards in parallel. To provide a better understanding of the impact of the shard creation optimization stated in §5, we made two modifications, one that creates dynamic shards aggressively (**ADS**) and a second that uses the optimization in §5 (**ODS**). We first report the performance of our algorithms over the first five graphs on HDD in Table 4.

We ran each program until it converged to evaluate the full impact of our I/O optimization. We observed that for each program the numbers of iterations taken by **Baseline** and **ODS** are almost the same. That is, despite the delays needed due to missing edges, the accumulation-based computation and shard creation optimizations minimize the vertices that need to be delayed, yielding the same convergence speed in **ODS**. **ADS** can increase the number of iterations in a few cases due to the delayed convergence. Due to space limitations, the iteration numbers are omitted from the paper. On average, **ADS** and **ODS** achieve an up to 1.2× and 1.8× speedup over *Baseline*.

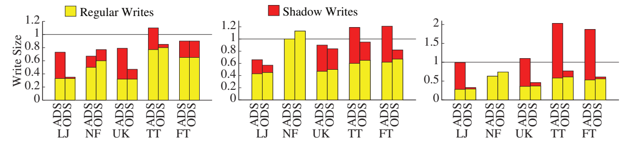
PR, BP, and HS are computation-intensive programs and they operate on large working sets. For these three programs, on average **ADS** speeds up graph processing by 1.53×, 1.50× and 1.22×, respectively. **ODS** performs much better providing speedups of 2.44×, 1.94×, and 2.82× respectively. The optimized version **ODS** performs better than the aggressive version **ADS** because **ODS** is likely to eliminate edges after the computation of their source vertices becomes stable, and thus edges that will be useful in a few iterations are likely to be preserved in dynamic shards. **ODS** consistently outperforms the baseline. While **ADS** outperforms the baseline in most cases, eliminating edges aggressively delays the algorithm convergence for HS on UK (*i.e.*, by 20% more iterations).

MSSP and CC require less computation and they operate on smaller and constantly changing working sets. Small benefits were seen from both **ADS** (1.15× speedup) and **ODS** (1.30× speedup), because eliminating edges achieves I/O efficiency at the cost of locality.

Figure 6 reports a breakdown of speedups on iterations for PR, BP, and HS. Two major observations can



(a) Normalized read size for PR. (b) Normalized read size for BP. (c) Normalized read size for HS.



(d) Normalized write size for PR. (e) Normalized write size for BP. (f) Normalized write size for HS.

Figure 7: Read and write size for different benchmarks normalized w.r.t. the baseline.

be made here. First, the performance improvement increases as the computation progresses, which confirms our intuition that the amount of useful data decreases as the computation comes close to the convergence. Second, the improvements from **ADS** exhibit a saw-tooth curve, showing the need of the optimizations in **ODS**: frequent drops in speedups are due to frequent shard creation and shadow iterations. These time reductions are entirely due to reduced I/O because the numbers of iterations taken by **ODS** and **Baseline** are almost always the same.

	PR	**BP**	**HS**
BL	153h:33m	80h:19m	147h:48m
ODS	92h:26m	54h:29m	92h:7m
Speedup	**1.66×**	**1.47×**	**1.60×**

Table 5: PR, BP and HS on YW.

Since the YW graph is much larger and takes much longer to run, we evaluate **ODS** for PR, BP and HS whose performance is reported in Table 5. **ODS** achieves a 1.47 – 1.60× speedup over **Baseline** for PR, BP and HS.

Performance on SSD To understand whether the proposed optimization is still effective when high-bandwidth **SSD** is used, we ran experiments for PR and BP on a machine with the same configuration except that SSD is employed to store shards. We found that the performance benefits are consistent when SSD is employed: on average, **ADS** accelerates PR, BP and HS by 1.25×, 1.18× and 1.14× respectively, whereas **ODS** speeds them up by 1.67×, 1.52× and 1.91× respectively.

Our techniques are independent of the storage type and the performance benefits are mainly achieved by reducing shard loading time. This roughly explains why a lower benefit is seen on SSD than on HDD – for example, compared to HDD, the loading time for FT on SSD decreases by 8%, 11% and 7% for PR, BP and HS, respectively.

6.2 I/O Analysis

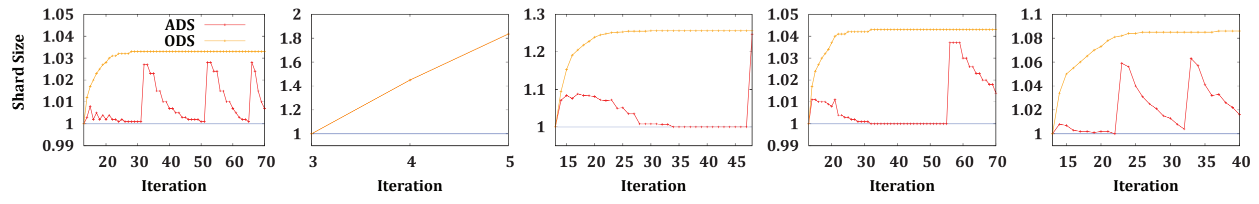
Data Read/Written Figure 7 shows the amount of data read and written during the graph processing in the modified GraphChi, normalized *w.r.t. Baseline*. Reads and writes that occur during shadow iterations are termed *shadow reads* and *shadow writes*. No shadow iteration has occurred when some applications were executed on the Netflix graph (*e.g.*, in Figures 7 (b), (c), (e), and (f)), because processing converges quickly and dynamic shards created once are able to capture the active set of edges until the end of execution.

Due to space limitations, we only show results for PR, BP and HS; similar observations can be made for the other applications. Clearly, **ODS** reads/writes much less data than both **Baseline** and **ADS**. Although shadow iterations incur additional I/O, this overhead can be successfully offset from the savings in regular iterations. **ADS** needs to read and write more data than **Baseline** in some cases (*e.g.*, Friendster in Figure 7c, Twitter in Figure 7d and Figure 7e). This shows that creating dynamic shards too frequently can negatively impact performance.

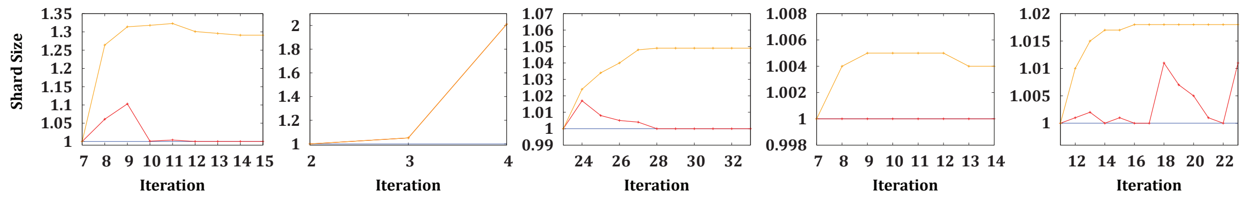
Size of Dynamic Shards To understand how well **ADS** and **ODS** create dynamic shards, we compare the sizes of intermediate dynamic shards created using these two strategies. Figure 8 shows the change of the sizes of dynamic shards as the computation progresses, normalized *w.r.t.* the size of an ideal shard. The ideal shard for a given iteration includes only the edges which were updated in the previous iteration, and hence, it contains the minimum set of edges necessary for the next iteration. Note that for both **ADS** and **ODS**, their shard sizes are close to the ideal sizes. In most cases, the differences are within 10%.

It is also expected that shards created by **ODS** are often larger than those created by **ADS**. Note that patterns exist in shard size changes for **ADS** such as HS on LJ (Figure 8a) and FT (Figure 8e). This is because the processing of delayed operations (in shadow iterations) over high-degree vertices causes many edges to become active and be included in new dynamic shards.

Edge Utilization Figure 9a reports the average *edge utilization rates* (EUR) for **ADS** and **ODS**, and compares them with that of **Baseline**. The average edge utilization rate is defined as the percentage of updated edges in a dynamic shard, averaged across iterations. Using dynamic shards highly improves the edge utilization: the EURs for **ADS** and **ODS** are between 55% and 92%. For CC on NF, the utilization rate is 100% even for **ODS**, because computation converges quickly and dynamic shards are created only once. Clearly, **ADS** has higher EURs than **ODS** because of its aggressive shard creation strategy. Using static shards throughout the execution leads to a very low EUR for **Baseline**.

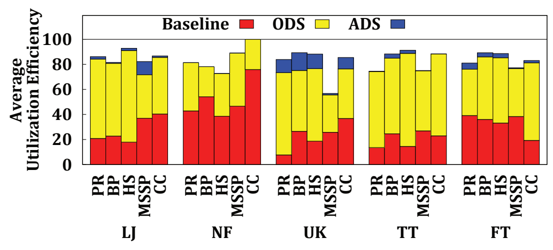


(a) HS on LJ (b) HS on NF (c) HS on UK (d) HS on TT (e) HS on FT

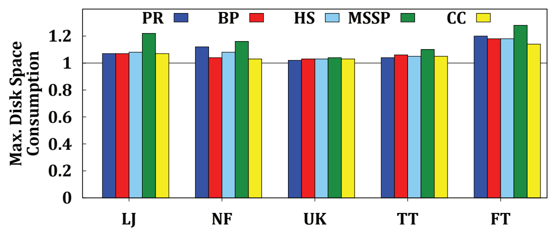


(f) MSSP on LJ (g) MSSP on NF (h) MSSP on UK (i) MSSP on TT (j) MSSP on FT

Figure 8: The dynamic shard sizes normalized *w.r.t.* the ideal shard sizes as the algorithm progresses.



(a) Edge utilization rates.



(b) Max disk space used.

Figure 9: Edge and disk utilization statistics.

Disk Space Consumption Figure 9b reports the maximum disk space needed to process dynamic shards normalized *w.r.t.* that needed by **Baseline**. Since we create and use dynamic shards only after vertex computations start stabilizing, the actual disk space it requires is very close to (but higher than) that required by **Baseline**. This can be seen in Figure 9b where the disk consumption increases by 2-28%. Note that the maximum disk space needed is similar for **ADS** and **ODS**, because dynamic shards created for the first time take most space; subsequent shards are either smaller (for **ADS**), or additionally include a small set of active edges (for **ODS**), which is insignificant to affect the ratio.

Delay Buffer Size With the help of the accumulation-based computation, the delay buffer often stays small throughout the execution. Its size is typically less than few 100KBs. The peak consumption was seen when ConnectedComponent was run on the Friendster graph, and the buffer size was 1.5MB.

6.3 Comparisons with X-Stream

Figure 10 compares the speedups and the per-iteration savings achieved by **ODS** and **X-Stream** over **Baseline** when running PR on large graphs. The saving per iteration was obtained by (1) calculating, for each iteration in which dynamic shards are created, $\frac{Baseline-ODS}{Baseline}$, and (2) taking an average across savings in all such iterations. While the per-iteration savings achieved by dynamic shards are higher than those by X-Stream, **ODS** is overall slower than X-Stream (*i.e.*, **ODS** outperforms X-Stream on UK but underperforms it on other graphs).

This is largely expected due to the fundamentally different designs of the vertex- and edge-centric computation models. Our optimization is implemented in GraphChi, which is designed to scan the whole graph multiple times during each iteration, while X-Stream streams edges in and thus only needs one single scan. Hence, although our optimization reduces much of GraphChi's loading time, this reduction is not big enough to offset the time spent on extra graph scans. Furthermore, in order to avoid capturing a large and frequently changing edge set (as described in §5.1), our optimization for creating and using dynamic

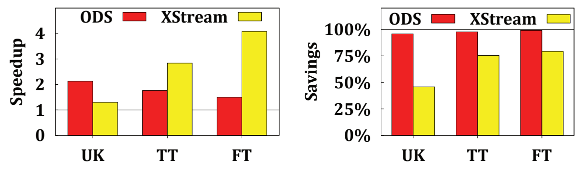


Figure 10: Speedups achieved (left) and per-iteration savings in execution time achieved (right) by **ODS** and **X-Stream** over **Baseline** using PR.

shards gets activated after a certain number of iterations (*e.g.*, 20 and 14 for TT and FT, respectively), and these (beginning) iterations do not get optimized.

Although X-Stream has better performance, the proposed optimization is still useful in practice for two main reasons. First, there are many vertex-centric systems being actively used. Our results show that the use of dynamic shards in GraphChi has significantly reduced the performance gap between edge-centricity and vertex-centricity (from 2.74× to 1.65×). Second, our performance gains are achieved only by avoiding the loading of edges that do not carry updated values and this type of inefficiency also exists in edge-centric systems. Speedups should be expected when future work optimizes edge-centric systems using mechanisms proposed in §4.3.

7 Related Work

Single-PC Disk-based Graph Processing Single-PC graph processing systems [14, 23, 31, 27, 15, 29, 10] have become popular as they do not need expensive computing resources and free developers from managing clusters and developing/maintaining distributed programs.

GraphChi [14] pioneered single-PC-based out-of-core graph processing systems. As mentioned in §1, shards are created during pre-processing and never changed during graph computation, resulting in wasteful I/O. This work exploits dynamic shards whose data can be dynamically adjustable to reduce I/O.

Efforts have been made to reduce I/O using semi-external memory and SSDs. Bishard Parallel Processor [18] aims to reduce non-sequential I/O by using separate shards to contain incoming and outgoing edges. This requires replication of all edges in the graph, leading to disk space blowup. X-Stream [23] uses an edge-centric approach in order to minimize random disk accesses. In every iteration, it streams and processes the entire unordered list of edges during the `scatter` phase and applies updates to vertices in the `gather` phase.

Using our approach, *dynamic edge-lists* can be created to reduce wasteful I/O in the `scatter` phase of X-Stream. GridGraph [31] uses partitioned vertex chunks and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. It enables selective scheduling by eliminating processing of edge blocks for which vertices in the corresponding chunks are not scheduled. However, the two-level partitioning is still done statically. Conceptually, making partitions dynamic would provide additional benefit over the 2-level partitioning.

FlashGraph [29] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built based on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [10] is an out-of-core computation engine for graph database to process graphs using SSDs. Since TurboGraph uses an adjacency list based representation, algorithms need to be expressed as sparse matrix-vector multiplication, which has a limited applicability because certain algorithms such as triangle counting cannot be expressed in this manner. Work from [21] uses an asynchronous approach to execute graph traversal algorithms with semi-external memory. It utilizes in-memory prioritized visitor queues to execute algorithms like breadth-first search and shortest paths.

Shared Memory and Distributed Graph Systems Google's Pregel [17] provides a synchronous vertex centric framework for large scale graph processing. GraphLab [16] is a framework for distributed asynchronous execution of machine learning and data mining algorithms on graphs. PowerGraph [8] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. Cyclops [5] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. GraphX [9] maps graph processing back to the dataflow framework and presents an abstraction that can be easily implemented using common dataflow operators. Chaos [22] utilizes disk space on multiple machines to scale graph processing.

Ligra [24] provides a shared memory abstraction for vertex algorithms. The abstraction is particularly suitable for graph traversal. Work from [19] presents a shared-memory implementation of graph DSLs on a generalized Galois system, which has been shown to outperform their original implementations. GRACE [26], a shared-memory system, processes graphs based on message passing and supports asynchronous execution by using stale messages. Orthogonal to these shared memory systems, this work aims to improve the I/O efficiency of disk-based graph systems. Graph reduction techniques [12] can be used to further reduce I/O by processing a small subgraph first and then feeding values to the original graph.

8 Conclusion

We present an optimization that dynamically changes the layout of a partition structure to reduce I/O for disk-based graph systems. Our experiments with GraphChi demonstrate that this optimization has significantly shortened its I/O time and improved its overall performance.

Acknowledgments

We would like to thank our shepherd Byung-Gon Chun as well as the anonymous reviewers for their valuable and thorough comments. This work is supported by NSF grants CCF-1524852 and CCF-1318103 to UC Riverside, NSF grants CNS-1321179 and CCF-1409423 to UC Irvine, and ONR grant N00014-16-1-2149 to UC Irvine.

References

[1] GraphLab Create. https://dato.com/products/create/, 2016.

[2] Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X. Group formation in large social networks: Membership, growth, and evolution. In *KDD* (2006), pp. 44–54.

[3] Bennett, J., and Lanning, S. The netflix prize. In *Proceedings of KDD cup and workshop* (2007), vol. 2007, p. 35.

[4] Boldi, P., and Vigna, S. The WebGraph framework I: Compression techniques. In *WWW* (2004), pp. 595–601.

[5] Chen, R., Ding, X., Wang, P., Chen, H., Zang, B., and Guan, H. Computation and communication efficient graph processing with distributed immutable view. In *HPDC* (2014), pp. 215–226.

[6] Cui, H., Cipar, J., Ho, Q., Kim, J. K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC* (2014), pp. 37–48.

[7] Friendster network dataset, 2015.

[8] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), pp. 17–30.

[9] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In *OSDI* (2014), pp. 599–613.

[10] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD* (2013), pp. 77–85.

[11] Kang, U., Horng, D., and Faloutsos, C. Inference of beliefs on billion-scale graphs. In *Large-scale Data Mining: Theory and Applications* (2010).

[12] Kusum, A., Vora, K., Gupta, R., and Neamtiu, I. Efficient processing of large graphs via input reduction. In *ACM HPDC* (2016).

[13] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW* (2010), pp. 591–600.

[14] Kyrola, A., Blelloch, G., and Guestrin, C. GraphChi : Large-scale graph computation on just a PC. In *OSDI*, pp. 31–46.

[15] Lin, Z., Kahng, M., Sabrin, K. M., Chau, D. H. P., Lee, H., , and Kang, U. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData* (2014), pp. 159–164.

[16] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[17] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., Czajkowski, G., and Inc, G. Pregel: A system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[18] Najeebullah, K., Khan, K. U., Nawaz, W., and Lee, Y.-K. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering 9*, 2 (2014), 199–212.

[19] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *SOSP* (2013), pp. 456–471.

[20] Page, L., Brin, S., Motwani, R., and Winograd, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford University, 1998.

[21] Pearce, R., Gokhale, M., and Amato, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (2010), pp. 1–11.

[22] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: Scale-out graph processing from secondary storage. In *SOSP* (2015), pp. 410–424.

[23] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP* (2013), pp. 472–488.

[24] Shun, J., and Blelloch, G. E. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP* (2013), pp. 135–146.

[25] Vora, K., Koduru, S. C., and Gupta, R. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA* (2014), pp. 861–878.

[26] Wang, G., Xie, W., Demers, A., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[27] Wang, K., Xu, G., Su, Z., and Liu, Y. D. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC* (2015), pp. 387–401.

[28] Yahoo! Webscope Program. `http://webscope.sandbox.yahoo.com/`.

[29] Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C. E., and Szalay, A. S. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST* (2015), pp. 45–58.

[30] Zhu, X., and Ghahramani, Z. Learning from labeled and unlabeled data with label propagation. Tech. Rep. CALD-02-107, Carnegie Mellon University, 2002.

[31] Zhu, X., Han, W., and Chen, W. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015), pp. 375–386.