

# ITERATIVE GRAPH COMPUTATION IN THE BIG DATA ERA

A Dissertation

Presented to the Faculty of the Graduate School  
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of  
Doctor of Philosophy

by

Wenlei Xie

August 2015

© 2015 Wenlei Xie  
ALL RIGHTS RESERVED

# ITERATIVE GRAPH COMPUTATION IN THE BIG DATA ERA

Wenlei Xie, Ph.D.

Cornell University 2015

Iterative graph computation is a key component in many real-world applications, as the graph data model naturally captures complex relationships between entities. The big data era has seen the rise of several new challenges to this classic computation model. In this dissertation we describe three projects that address different aspects of these challenges.

First, because of the increasing volume of data, it is increasingly important to scale iterative graph computation to large graphs. We observe that an important class of graph applications performing little computation per vertex scales poorly when running on multiple cores. These *computationally light* applications are limited by memory access rates, and cannot fully utilize the benefits of multiple cores. We propose a new block-oriented computation model which creates two levels of iterative computation. On each processor, a small block of highly connected vertices is iterated locally, while the blocks are updated iteratively at the global level. We show that block-oriented execution reduces the communication-to-computation ratio and significantly improves the performance of various graph applications.

Second, because of the increasing velocity of data generation, iterative computation over graph streams is an increasingly important problem. Extracting insights from such graph data streams is difficult because most graph mining algorithms are designed for static graph structures. Existing systems are based on either a *snapshot model* or a *sliding window model*. Both of the models embody

a binary view of an edge's role: they either forget old edges beyond a fixed period abruptly, or fail to emphasize recent data sufficiently. We propose a novel *probabilistic edge decay model* that samples edges according to a probability that decreases over time. We exploit the overlap between samples to reduce memory consumption and accelerate the analytic procedure. We design and implement the end-to-end system on top of Spark.

Third, because of the increasing variety of data, people often perform the same iterative computation over a parametric family of graphs. These graphs have the same structure but associate different weights to the same vertices and edges. We focus on the parametric PageRank problem, usually known as personalized PageRank. This method has been widely used to find vertices in a graph that are most relevant to a query or user. However, edge-weighted personalized PageRank has been an open problem for over a decade. We describe the first fast algorithm for edge-weighted personalized PageRank on general graphs. With a reduced model built in the preprocessing stage, we can solve problems in a much smaller reduced space producing good approximate results. This opens opportunities to many applications that were previously infeasible, such as interactive learning-to-rank based on user's feedback.

## BIOGRAPHICAL SKETCH

Wenlei Xie was born in Ningbo, China. He attended Fudan University in Shanghai beginning in 2006. In 2010, Wenlei spent one semester as a visiting student at Microsoft Research Asia. He graduated from Fudan University in 2010 with a B.Sc. in Computer Science.

Since 2010, Wenlei has been studying for his doctorate degree in Computer Science at Cornell University.

To the memory of my grandfather, Maojiang Sang.

## ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to my advisor Johannes Gehrke. He has been always understanding, open to discussion, and full of inspiration and helpful suggestions. He has taught me not only many things about databases, but also how to think as a computer scientist. His insightful advice helped me develop my skills to define and articulate a research question.

I would like to give special thanks to David Bindel. We had numerous discussions on the projects we worked together, and his suggestions from a scientific computing and mathematical perspective are invaluable. I learned a lot about how to do research from these discussions, such as how to locate and address performance bottlenecks in a systematic way.

I have been fortunate to work with Alan Demers. His advice from a system perspective helped me thoroughly understand the experimental results. Through discussions with him, I also learned principles about building large-scale systems.

I am thankful to Robert Kleinberg, for his constructive suggestions, and also his inspiring lectures on algorithm design and analysis.

It was fortunate to have the chance to work with the Big Red Database Group at Cornell. Especially, I would like to thank Guozhang Wang. Through our collaboration in research projects, I learned how to deal with situations when the results are not as expected. I would also like to thank Michael Hay and Lucja Kot for their help and insightful suggestions in our research collaboration, and Gabriel Bender, Tuan Cao, Bailu Ding, Michaela Gotz, Nitin Gupta, Yin Lou, Sudip Roy, Marcos Vaz Salles, Ben Sowell, and Tao Zou for many enlightening discussions.

In addition, I would like to acknowledge my internship mentors. At Google,

Sanjay Agrawal and Kevin Lai taught me how to solve problems in complicated real-world software systems. During my internship at IBM Almaden Research Center, Yuanyuan Tian exposed me to the research problems faced by the leading industry research lab. Srinivas Vemuri introduced me to the challenging graph computation problems at LinkedIn.

My thanks to all the friends and colleagues I met during my graduate studies, especially Shuo Chen and Chenhao Tan.

Finally, I thank my family for their unconditional love and support. I thank my grandfather Maojiang for caring for me when I was a child. I am sad that he could not see me complete this dissertation. His memory will be with me always.

## TABLE OF CONTENTS

Biographical Sketch . . . . .	iii
Dedication . . . . .	iv
Acknowledgements . . . . .	v
Table of Contents . . . . .	vii
List of Tables . . . . .	ix
List of Figures . . . . .	x
<b>1 Introduction</b>	<b>1</b>
1.1 Volume . . . . .	1
1.2 Velocity . . . . .	3
1.3 Variety . . . . .	4
1.4 Dissertation Outline . . . . .	5
<b>2 Fast Iterative Graph Computation with Block Updates</b>	<b>6</b>
2.1 Introduction . . . . .	6
2.2 Block vs. Vertex-at-a-Time . . . . .	10
2.3 Block-Oriented Computation . . . . .	14
2.3.1 Block Formulation . . . . .	15
2.3.2 Per-Block Update . . . . .	16
2.3.3 Two-Level Scheduling . . . . .	20
2.4 Block-aware Execution Engine . . . . .	21
2.4.1 Concurrency Control . . . . .	22
2.4.2 Scheduling . . . . .	24
2.5 Experiments . . . . .	33
2.5.1 Applications . . . . .	34
2.5.2 Experimental Setup . . . . .	35
2.5.3 Results . . . . .	37
2.6 Related Work . . . . .	48
<b>3 Dynamic Interaction Graphs with Probabilistic Edge Decay</b>	<b>51</b>
3.1 Introduction . . . . .	52
3.2 Dynamic Interaction Graphs: Existing Models . . . . .	57
3.3 The PED Model . . . . .	59
3.4 Maintaining Sample Graphs . . . . .	61
3.4.1 Incremental Updating: General Approach . . . . .	61
3.4.2 The Aggregate Graph . . . . .	62
3.4.3 Eager Incremental Updating . . . . .	66
3.4.4 Lazy Incremental Updating . . . . .	67
3.4.5 General Decay Functions . . . . .	68
3.5 Bulk Analysis of Sample Graphs . . . . .	68
3.5.1 Bulk Graph Execution Model . . . . .	69
3.5.2 Incremental Graph Analysis . . . . .	72

3.6	Implementation on Spark . . . . .	73
3.6.1	Spark Overview . . . . .	73
3.6.2	Implementation and Optimization . . . . .	73
3.7	Experimental Evaluation . . . . .	78
3.7.1	Incremental Updating Methods . . . . .	82
3.7.2	Dynamic Graph Analysis . . . . .	86
3.8	Related Work . . . . .	92
<b>4</b>	<b>Interactive Edge-Weighted Personalized PageRank via Model Reduction</b>	<b>95</b>
4.1	Introduction . . . . .	95
4.2	Preliminaries . . . . .	99
4.3	Model Reduction . . . . .	101
4.3.1	Reduced Space Construction . . . . .	101
4.3.2	Extracting Approximations . . . . .	102
4.3.3	Parameterization Forms . . . . .	104
4.3.4	Choosing interpolation equations . . . . .	106
4.4	Learning to Rank . . . . .	109
4.5	Experiments . . . . .	111
4.5.1	Setup . . . . .	111
4.5.2	Preprocessing . . . . .	114
4.5.3	Global PageRank . . . . .	117
4.5.4	Localized PageRank . . . . .	119
4.5.5	Parameter Learning . . . . .	122
4.6	Related Work . . . . .	124
<b>5</b>	<b>Conclusions</b>	<b>126</b>
<b>6</b>	<b>Appendix for Chapter 5</b>	<b>128</b>
6.1	Quasi-optimality . . . . .	128
6.2	Model Reduction with Constraints . . . . .	129
	<b>Bibliography</b>	<b>131</b>

## LIST OF TABLES

2.1	Benefit of Cache Performance . . . . .	18
2.2	Dataset Summary . . . . .	36
3.1	Per-batch time for lazy updating (after 30th batch) . . . . .	85
3.2	Coalesce operations for lazy incremental updating . . . . .	85
4.1	Basic Statistics of Datasets . . . . .	113
4.2	Preprocessing Time on Different Datasets . . . . .	117
4.3	Avg. Running Time per Opt. Iteration . . . . .	124

## LIST OF FIGURES

2.1	Memory Wall for Lightweight Computation . . . . .	7
2.2	Vertex- vs. Block-Oriented Computation . . . . .	14
2.3	Running time vs. inner iterations for PageRank. . . . .	19
2.4	Illustration for Block Level Scheduling . . . . .	28
2.5	Effect of block size. . . . .	38
2.6	Effect of block strategy. . . . .	39
2.7	Effect of static inner scheduling. . . . .	40
2.8	Effect of dynamic inner scheduling. . . . .	46
2.9	Effect of scheduling policies in GraphLab and in GRACE. . . . .	48
3.1	Influence analysis example . . . . .	54
3.2	Example aggregate graph . . . . .	63
3.3	Incremental updating for one edge . . . . .	66
3.4	Overall Data Pipeline of TIDE System . . . . .	74
3.5	Per day batch size of the Twitter dataset . . . . .	78
3.6	Per-batch time for incremental updating . . . . .	82
3.7	Running aggregate graph size . . . . .	82
3.8	Avg per-batch time for incremental updating (after 30th batch) . . . . .	83
3.9	Quality of results for degree centrality . . . . .	86
3.10	Quality of results for Katz centrality . . . . .	87
3.11	Bulk graph execution vs the naive approach . . . . .	91
4.1	Singular Values . . . . .	115
4.2	Running Time on DBLP Graph . . . . .	119
4.3	Accuracy on DBLP Graph . . . . .	119
4.4	Objective Function Value for Parameter Learning . . . . .	119
4.5	Global PageRank on Weibo Graph . . . . .	120
4.6	Localized PageRank with Linear Parameterization . . . . .	121
4.7	Localized PageRank with Scaled-Linear Parameterization . . . . .	122

# CHAPTER 1

## INTRODUCTION

Graphs are versatile tools used to express complex relations, such as friendships in social networks or hyperlinks in the web. Graphs have also been adopted in a variety of other domains, such as recommendation systems, bioinformatics, physical simulations, and computer vision.

Computations on graph data are usually iterative, i.e. they update the graph data repeatedly until a fixed point is reached. Classic examples of iterative graph computation include PageRank, shortest path and coordinate descent. This is because many problems are very difficult or even impossible to solve efficiently with a direct method. For example, exact inference on Markov Random Fields is NP-complete, and so an iterative method called belief propagation is the *de facto* standard algorithm to obtain an approximate result. Iterative methods are also widely used in large-scale PDEs because direct methods may be too computationally expensive, depending on the graph structure.

The emergence of big data poses new challenges to iterative graph computation, in spite of the fact that iterative graph computation has been used and studied for decades. In this dissertation, we discussed these challenges from three different aspects of big data: *volume*, *velocity* and *variety*.

### 1.1 Volume

As more and more graph data is being collected and stored, applications are required to analyze graphs of unprecedented size. Examples of large scale graph computation include 3D model construction over Internet-scale collections of

images [33], social influence analysis on microblogs [21], and anomaly detection over the web and social networks [61]. Graphs with billions of edges are common [104, 61], and graphs with trillions of edges have already been used in industry applications [2]. Because of this explosion in graph size, parallelism is usually exploited. For example, Crandall et al. reported the use of a 200-core Hadoop cluster to solve structure-from-motion to help build 3D models from large unstructured collections of images [33].

However, iterative graph computations are also communication intensive, as updates to data associated with vertices and edges depend on data from their neighbors, and the processors assigned to different parts of the graph must communicate after each iteration. Such interprocessor communication is accomplished either through a global memory bus in the single-node multicore environment, or through networks in the distributed environment. The benefits of parallel computation are limited when each vertex or edge update is inexpensive, as processors are usually waiting for data rather than computing on data.

We discuss the challenge posed by the volume of graph data in Chapter 2, with a focus on the multicore environment. In particular, we observe that interprocessor communication becomes a major overhead for a class of *computationally light* applications which perform little computation per vertex or edge. Many important applications, including PageRank, connected components, and coordinate descent, belong to this class. We propose a novel *block-oriented computation model* to reduce the communication-to-computation ratio, and thus significantly improve overall performance. The graph is first partitioned into highly connected blocks in the preprocessing stage. At execution time, the engine considers the block as a scheduling unit, rather than the individual vertex. This

scheduling strategy improves locality by iterating inside the block. Moreover, different schedulers can be used inside a block and across the blocks, which reduces the overall scheduling overhead.

## 1.2 Velocity

Graph data is not only increasing in volume, but it is being generated at increasing rates. With ubiquitous mobile devices, interactions between entities are recorded continuously. Such interactions include message activities between users from social networking sites, browsing and click activities between users and items from electronic commerce companies, and phone calls between people from telecommunication service providers.

There is growing interest in analyzing these graph data streams: enterprises are extracting time-sensitive insights from the data for better recommendations and decision making [67]; and researchers are studying these time-evolving graphs to better understand temporal behaviors of individual nodes [93]. However, analyzing graph data streams is a challenging task, as the majority of graph mining algorithms are designed for static graphs.

We discuss the challenge posed by the velocity of graph data in Chapter 3, focusing on how to analyze graph data streams with existing static graph algorithms. Existing models are based on a binary view of an edge's role in the analysis: an edge is either included for analysis or not. We demonstrate that this simplistic view cannot simultaneously satisfy two important properties, *recency* and *continuity*, required by temporal graph analysis. We show how to break this binary view via a novel probabilistic edge decay model. All edges have

a chance to be considered, while new edges are more likely to be considered. We discuss how to exploit the overlap between sample graphs to significantly accelerate the analysis procedure. We have incorporated these ideas in TIDE, an end-to-end system that handles graph ingestion and query answering. We describe the design and implementation of TIDE in Chapter 3.

### 1.3 Variety

Increasingly, graph data sets contain not only the graph topology, but also rich metadata and semantic information associated with vertices and edges. Consider a social networking site: besides the relationships between users, it also collects all the data created and curated by each individual user, such as posts, chat histories with different users, and behaviors related to different types of articles.

Vertex and edge attributes play a crucial role in applications of graph analysis. PageRank, an approach to ranking vertices in a graph, has been extended by researchers to use vertex and edge information to produce rankings most relevant to a user or query almost since its introduction [39]. The PageRank model is based on a random walk over a graph, and personalized PageRank methods bias this random walk based on the data associated with vertices and edges. For example, in a graph extracted from an object database with type annotations on edges, we may transition through some types of edges more often than other types of edges. However, with this level of personalization, it is difficult to compute PageRank vectors interactively. As a result, existing systems provide only a very limited space of personalization based on offline computations.

In Chapter 4, we discuss the challenge posed by personalizing PageRank results using a wide variety of vertex and edge data. Our method is based on a novel *model reduction* strategy. The PageRank problem is a large linear system which cannot be solved quickly online, as the dimension of the linear system is equal to the number of vertices in graph. However, with a reduced model of much lower dimension constructed offline, we only need to solve a linear system in the reduced space rather than in the original space. This allows iterative personalized PageRank computation that would benefit a variety of applications.

## 1.4 Dissertation Outline

Chapter 2 describes the block execution model to support fast iterative graph computation on multicore machines. Chapter 3 presents TIDE, a distributed system for analyzing dynamic graphs based on a novel probabilistic edge decay model. Chapter 4 describes how to compute edge-weighted personalized PageRank interactively via model reduction. We present related work for each project in its respective chapter. Chapter 5 concludes.

## CHAPTER 2

### FAST ITERATIVE GRAPH COMPUTATION WITH BLOCK UPDATES

Scaling iterative graph processing applications to large graphs is an important problem. Performance is critical, as data scientists need to execute graph programs many times with varying parameters. The need for a high-level, high-performance programming model has inspired much research on graph programming frameworks.

In this chapter, we show that the important class of *computationally light* graph applications – applications that perform little computation per vertex – has severe scalability problems across multiple cores as these applications hit an early “memory wall” that limits their speedup. We propose a novel block-oriented computation model, in which computation is iterated locally over blocks of highly connected nodes, significantly improving the amount of computation per cache miss. Following this model, we describe the design and implementation of a block-aware graph processing runtime that keeps the familiar vertex-centric programming paradigm while reaping the benefits of block-oriented execution. Our experiments show that block-oriented execution significantly improves the performance of our framework for several graph applications.

#### 2.1 Introduction

Graphs express complex data dependencies among entities, so large graphs are a key modeling component for many applications, such as structure from motion [33], community detection [89], physical simulations [97], and link analy-

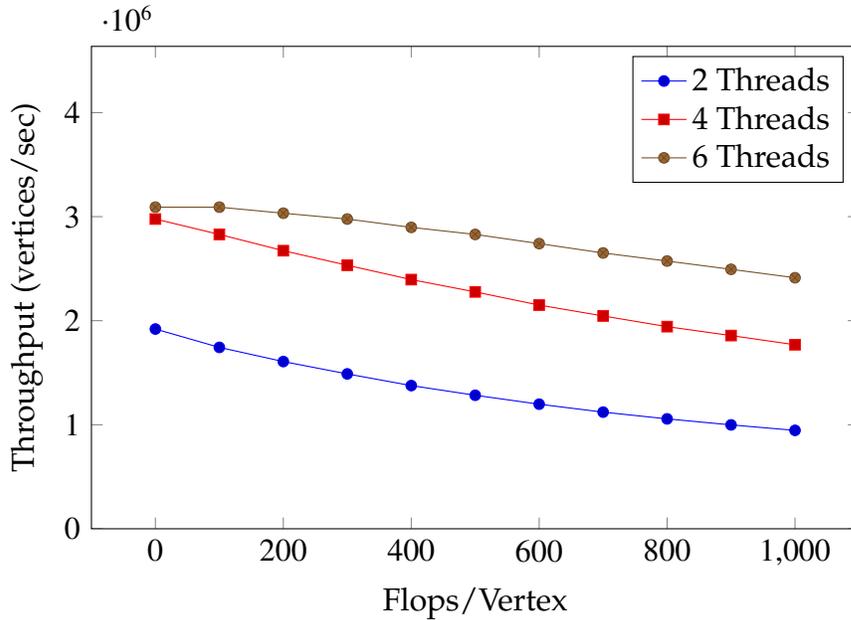


Figure 2.1: Memory Wall for Lightweight Computation

sis [23]. Graph processing usually exploits parallelism, since it is both compute- and memory-intensive. Recently several graph computation frameworks have been introduced with the express goal of helping domain experts develop graph applications quickly [77, 75, 76, 62, 42, 110]. These frameworks present to their users a “think-as-a-vertex” programming model in which each vertex updates its own data based on the data of neighboring vertices. The programming model is coupled with an iterative execution model, which applies the vertex update logic repeatedly until the computation converges. These frameworks have been used successfully in many graph processing applications [77, 75, 62].

Different graph applications perform different amounts of computation per vertex. *Computationally light* applications, such as PageRank, perform tens of floating point operations (flops) per vertex, whereas *computationally heavy* applications such as Belief Propagation, may perform orders of magnitude more work per vertex. This significantly affects performance: A computationally

heavy application can fully utilize several cores, while computationally light applications are limited by memory access rates.

We illustrate this phenomenon by running GRACE, our highly-optimized graph processing engine [110], on a Xeon machine with two sockets with four cores each. The processing threads are evenly distributed across sockets, and there are no bottlenecks due to locking or critical sections [110]. Figure 2.1 shows how computational load, measured in flops per vertex, impacts throughput, measured in vertex updates per second. For computationally heavy applications, adding threads significantly improves throughput. But for computationally light applications, there is negligible improvement in throughput beyond four threads. This is because GRACE has reached the memory bandwidth of the processor; the time spent retrieving vertex and edge data from memory exceeds the time spent computing on the data. Adding more cores only exacerbates this problem.

In this chapter, we propose a novel block-oriented computation model that significantly improves the throughput of computationally light graph applications. Inspired by the block-oriented computation model for matrices and grids from the HPC community [16], we use standard methods to partition the graph into blocks of highly connected vertices [15, 63, 101, 42]. Then, instead of scheduling individual vertices, we schedule blocks of related vertices. This new model opens up two opportunities. First, we can update a block repeatedly to improve locality while accelerating convergence. Repeatedly updating one vertex does not improve convergence, as the neighboring vertices always provide the same data, and thus the computation always produces the same result. But in repeatedly updating a block of connected vertices, each step can

make additional progress. Second, we can use a different scheduler inside a block than the one we use across blocks. For example, we may schedule nodes within blocks in a round-robin fashion, but schedule blocks based on which block contains the node with the worst residual error. Since updating a block is more expensive than updating one vertex, we can afford the overhead of a relatively more expensive scheduler to choose blocks. We show that different scheduling algorithms (of different cost) significantly affect the convergence of graph computations, and thus the overall performance.

We have added support for block-level computation to GRACE through a new block-aware runtime. This runtime has a novel block-level concurrency control protocol based on snapshot isolation; our approach effectively minimizes concurrency overhead. The runtime supports separate scheduling policies for blocks and for nodes within blocks, allowing users to trade off scheduling cost against speed of convergence.

In the next section, we introduce the vertex-centric programming abstraction for iterative graph processing and demonstrate the scalability problems associated with poor locality in the corresponding vertex-oriented computation model. In Section 2.3 we introduce our block-oriented computation model, which has better locality. In Section 2.4, we turn to our block-aware execution engine, and describe in detail the scheduling mechanisms that allow us to maintain fast convergence with low scheduling overhead. We present experimental results in Section 2.5 to demonstrate how our block-oriented computation model can improve update throughput and convergence rates for real-world graph processing applications. We survey related work in Section 2.6.

## 2.2 Block vs. Vertex-at-a-Time

Most parallel graph processing frameworks present to their users a vertex-centric programming abstraction: application logic is written as a local update function to be run at individual vertices. This function is “local” in the sense that it is executed on a single vertex, and updates the vertex data as a function of data at neighboring edges and vertices. In this model, vertices communicate only with their neighbors, either by sending messages or by remote data access. Thus, vertex updates can proceed in parallel, with low-level details of the parallelism handled transparently by the framework.

To be more concrete, suppose we are given a directed graph  $G(V, E)$ . Users can define arbitrary attributes on vertices and edges to represent application data. To simplify the discussion, we assume vertex values can be modified, but edge values are read-only. This assumption does not limit expressiveness, since we can store the writable data for edge  $(u, v)$  in vertex  $u$ . We denote the data on vertex  $v$  by  $S_v$ , extending this notation to sets of vertices as well. Similarly, we denote the data on edge  $(u, v)$  by  $S_{(u,v)}$ . The update function for a vertex  $v$  depends only on data on its incoming edges  $NE(v)$ , and on the vertices  $NV(v)$  that connect to  $v$  through  $NE(v)$ . The function `VertexUpdate` that maps the current state  $S_v^{\text{old}}$  of vertex  $v$  to its new state  $S_v^{\text{new}}$  has the following signature:

$$S_v^{\text{new}} = \text{VertexUpdate}(S_v^{\text{old}}, S_{NV(v)}, S_{NE(v)}).$$

During execution, the runtime schedules individual vertex updates. To achieve scalability, existing frameworks either (1) follow the Bulk Synchronous Parallel (BSP) model [105], arranging updates into iterations separated by global synchronization barriers, with updates in one iteration depending only on data

---

**Algorithm 2.1:** Vertex-Oriented Computation Model

---

- 1 Initialize the vertex data ;
- 2 **repeat**
- 3     Get a vertex  $v$  to be updated from the scheduler ;
- 4     Update the data of  $v$  based on  $S_{NV(v)}$  and  $S_{NE(v)}$  ;
- 5     Commit the update to  $S_v$  ;
- 6 **until** *No vertex needs to be updated anymore;*

---

written in the previous iteration [77, 62, 120, 102, 110]; or (2) update vertices asynchronously, based on the most recent data from neighboring vertices, with static or dynamic scheduling of the updates to achieve fast convergence [78, 76, 68]. Algorithm 1 summarizes this vertex-oriented computation model, in which different scheduler implementations can lead to either synchronous or asynchronous execution policies. Whatever scheduler is used, the resulting execution policy is at the granularity of vertices: the processor accesses one vertex at a time, loading the data from the vertex and its neighbors into local cache and triggering the update function `VertexUpdate`. Note for the BSP model the commit is not executed immediately but logged, and will be executed at the synchronization barrier.

The vertex-centric model is a useful programming abstraction, but the corresponding vertex-centric update mechanisms result in poor performance for computationally light graph algorithms, such as PageRank, shortest paths, connected components, and random walks. Such algorithms are communication-bound: computing a vertex update is cheaper than retrieving the required data from neighboring vertices. Thus, these algorithms scale poorly with increasing parallelism. Researchers have proposed ways to reduce the networking over-

head for computationally light applications in distributed memory environments [42, 60]; but we have observed poor scaling even in shared-memory environments, where main memory bandwidth quickly becomes a bottleneck [110].

On the other hand, researchers in high-performance computing (HPC) have studied communication bottlenecks for many years in the context of sparse matrix routines and PDE solvers, which make up a special class of iterative graph processing algorithms [16]. Many of the optimization techniques from this literature apply to other graph algorithms as well.

**Domain Decomposition for Elliptic PDEs.** Large, sparse linear systems and non-linear systems of equations often come from discretized elliptic partial differential equations (PDEs). These linear systems are typically solved by iterative methods [16], the most common of which are Krylov subspace methods with a preconditioning solver to accelerate the rate of convergence. For these problems, domain decomposition is widely used to construct both preconditioners and stand-alone solvers [99]. Domain decomposition methods partition the original domain into disjoint or overlapping subdomains, possibly recursively. The subdomain sizes are typically chosen to minimize inter-processor communication and maintain good cache locality. Each subdomain is updated in a local computation. Because the subdomain solvers are often expensive, these methods tend to have good communication-to-computation ratios; consequently, domain decomposition methods are popular in parallel PDE solvers.

**Block-oriented Scheduling for Eikonal Equations.** The eikonal equation is a nonlinear hyperbolic PDE that describes continuous shortest paths through a medium with varying travel speeds. It is used in many applications, ranging from optics to etch simulation [97]. Various methods with different vertex-

scheduling mechanisms have been proposed for this problem. Fast-Marching methods, like Dijkstra’s algorithm for discrete shortest path problems, dynamically schedule nodes so they are processed in increasing order of distance from the starting set. In contrast, Fast-Sweeping methods update in a fixed order, and so have lower scheduling overhead and more regular access patterns; but they require multiple iterations to converge. As with elliptic PDEs, domain decomposition has been used to parallelize Fast-Sweeping [121]; and recent work has introduced a domain decomposition approach that uses sweeping on subdomains and marching to schedule subdomain solves [25]. It has proven to be very effective, achieving fast convergence rates by using dynamic scheduling for subdomains while maintaining low scheduling overhead by using static scheduling at the per-vertex level.

Blocking is a standard idea that has been applied in many settings. Examples include tile-based Belief Propagation [74], Block Coordinate Descent [20], and cache-aware graph algorithms [84]. Unfortunately, a general data-centric framework for graph structured computations, which would save people from reinventing the wheel, is missing from the literature. In this chapter, we take the first step toward this goal by proposing a general block-oriented computation model for graph computations. As we will show below, our block-oriented computation model still works with a vertex-centric programming abstraction to achieve easy programming.

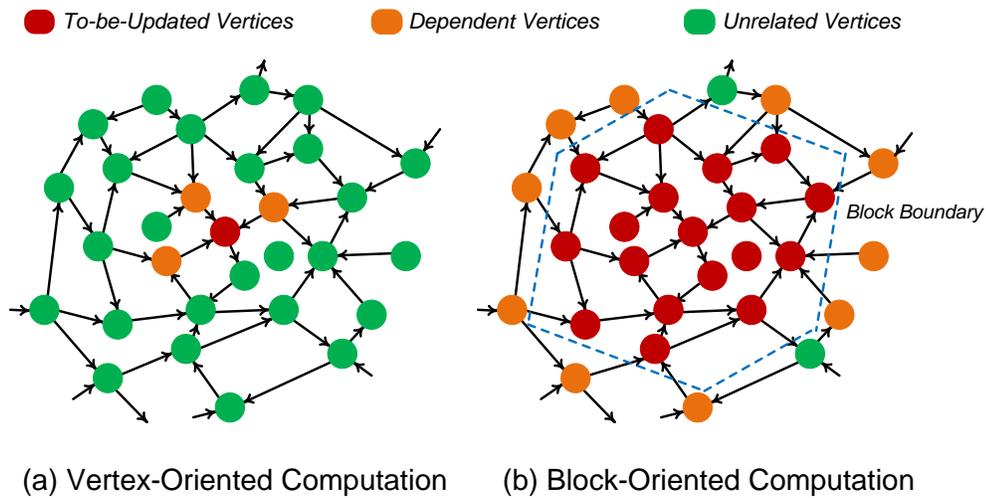


Figure 2.2: Vertex- vs. Block-Oriented Computation

### 2.3 Block-Oriented Computation

The block-oriented computation model is a natural extension to the vertex-oriented computation model. Figure 2.2 illustrates the two models: small red, yellow, and green circles represent vertices to be updated, vertices on which the update depends, and vertices unrelated to the current update. In the vertex-oriented computation model, only one vertex and the data on which it depends are loaded from memory for each update, while in the block-oriented computation model, all the vertices belonging to the same block are loaded from memory and updated together. For a cluster of processors, one can first partition the graph and assign subgraphs to the processors, then further partition the assigned subgraphs into blocks. The subgraphs are chosen to minimize the number of edges between them, so that adjacent vertices are likely to be in the same block.

### 2.3.1 Block Formulation

We initially partition  $G(V, E)$  into disjoint blocks  $B_1(V_1, E_1)$ ,  $B_2(V_2, E_2)$ ,  $\dots$ ,  $B_k(V_k, E_k)$ , where  $k$  should be much greater than the expected degree of parallelism. Block  $B_i$  contains all the edges that originate in  $V_i$ , i.e.  $E_i = \{(u, v) \in E \mid u \in V_i\}$ . Vertices  $V_i$  are the *local vertices* of  $B_i$ ; and edges  $E_i$  are its *local edges*. For a given block  $B$ , we will also use  $V(B)$  to denote its set of local vertices and  $E(B)$  to denote its set of local edges. When updating  $B_i$ , the local edges are read-only and the local vertices are read-write. We define the *incoming boundary vertices* of  $B_i$  as  $NV(B_i) = \{u \in V - V_i \mid (u, v) \in E, v \in V_i\}$ ; and we define the *incoming boundary edges* as This part of the state is read-only when updating  $B_i$ , and can be viewed as part of the input to the update procedure for  $B_i$ . Block  $B_j$  is an *incoming neighbor block* of  $B_i$  if  $B_j$  contains any incoming boundary vertices of  $B_i$ , i.e.  $V_j \cap NV(B_i) \neq \emptyset$

Similarly, we define the *outgoing boundary vertices* of  $B_i$  as  $OV(B_i) = \{v \in V - V_i \mid (u, v) \in E, u \in V_i\}$ , and the *outgoing boundary edges* as  $OE(B_i) = \{(u, v) \in E \mid v \in V - V_i, u \in V_i\}$ . Block  $B_j$  is an *outgoing neighbor block* of  $B_i$  if  $V_j \cap OV(B_i) \neq \emptyset$

The goal of the partitioning is to minimize the number of edges cut by the partition, while making the blocks roughly the same size to facilitate load balance. This is a well-studied problem for which many efficient methods are known [63, 15, 101, 100].

---

**Algorithm 2.2:** Block-Oriented Computation Model

---

- 1 Initialize the graph data ;
  - 2 **repeat**
  - 3     Get a block  $B$  to be updated from scheduler ;
  - 4     Update the data of  $u$  based on its boundary data  $S_{NV(B)}$  and  $S_{NE(B)}$  ;
  - 5     Commit the update to  $S_{V(B)}$  ;
  - 6 **until** *No block need to be updated anymore;*
- 

### 2.3.2 Per-Block Update

We organize computations around a *block update function*; see Algorithm 2. The block update function has the signature

$$S_B^{\text{new}} = \text{BlockUpdate}(S_B^{\text{old}}, S_{NV(B)}, S_{NE(B)}),$$

where  $S_{NV(B)}$  and  $S_{NE(B)}$  denote the data on the boundary vertices and boundary edges of  $B$ . A straightforward way to implement this function would be to let the user directly specify the block-level logic: by analogy to the vertex programming abstraction, we could have a *block-centric programming abstraction*, exposing the block structure, block data, and the dependent boundary data to the user. However, this block programming abstraction would not follow the “think-as-a-vertex” philosophy that has proven so successful in practice [77, 42]. The block-centric programming abstraction is more complicated than the vertex-centric programming abstraction because it introduces an artificial distinction between local and boundary vertices: a local vertex is modifiable and can access its neighbors’ data, while a vertex vertex is read-only and has no access to its neighbors’ data. Moreover, users are already familiar with the vertex-centric programming abstraction; many graph applications have al-

---

**Algorithm 2.3:** Cache-Aware Vertex-Oriented Computation Model

---

```
1 Initialize the vertex data ;
2 repeat
3   Get a set of vertices  $V' \subset V$  from global scheduler, the vertices in  $V'$  are
   closely connected ;
4   Add the scheduled vertices in  $V'$  into the local queue  $Q$  ;
5   while Local queue  $Q$  is not empty do
6      $v = Q.pop()$  ;
7     Update the data of  $v$  based on  $S_{NV(v)}$  and  $S_{NE(v)}$  ;
8     Commit the update to  $S_v$  ;
9   end
10 until No vertex needs to be updated anymore;
```

---

ready been written using it, and it would be inconvenient for users to learn a new abstraction and migrate their existing applications into it. Thus our goal is to let users write vertex-centric programs, then run those programs in the block execution model. To achieve this, our block update function is defined in terms of a (traditional) vertex update function and an optional inner-block scheduling policy `InnerScheduler`:

$$\text{BlockUpdate} = \text{InnerScheduler}(\text{VertexUpdate})$$

The inner-block scheduler iterates over some or all of the vertices inside a block and applies the user-specified `VertexUpdate` function to these vertices, possibly multiple times. For example, a simple inner-block scheduler could update each vertex in the block exactly once in a fixed order, while a more sophisticated scheduler could update vertices repeatedly until the block data converged.

Table 2.1: Benefit of Cache Performance

Scheduler	Time (s)	# Updates	# LLC Misses
Non Cache-Aware	9.52	34,152,807	197,500,000
Cache-Aware	5.15	34,152,807	37,500,000

A key benefit of this block computation model is improved locality of reference: by loading a block of closely connected vertices into the cache, we increase the cache hit rate and thus reduce the average data access time. We demonstrated this effect in the GRACE engine [110] using the experimental setup described in detail in Section 2.5.2. We ran personalized PageRank on the Google graph using both a vertex-centric scheduler (Algorithm 1) and a cache-aware scheduler (Algorithm 3). Algorithm 1 uses the default vertex scheduler in GRACE: it iterates over the vertices in a random order until convergence, with no regard for graph partitioning. The cache-aware scheduler in Algorithm 3 uses the graph partitioning to improve cache performance. Instead of fetching one vertex  $v$  and updating it, the cache-aware vertex scheduler fetches a set  $V'$  of closely-connected vertices, then updates each vertex in turn. Since vertices in  $V'$  share many neighbors, this method achieves better cache utilization. As shown in Table 2.1, the cache-aware vertex scheduler reduces the number of Last-Level-Cache (LLC) misses by about 80% compared to the non-cache-aware vertex scheduler, and reduces the total run time by nearly 50% with the same number of updates. We observed this effect in all the social graphs in our experiments.

Computationally light applications, which exhibit high data access to computation ratios, can run even faster by updating each vertex in a block multiple

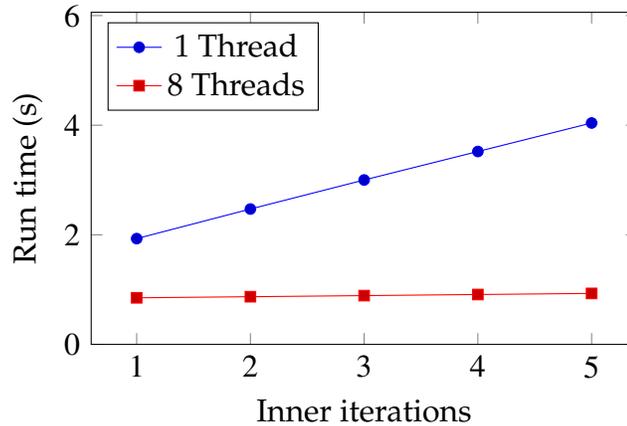


Figure 2.3: Running time vs. inner iterations for PageRank.

times before evicting it from the cache. This reduces the data access to computation ratio, improving end-to-end performance as long as the extra computations make at least some progress towards convergence. In fact, this idea has been used for years in large-scale linear system solvers in the high-performance computing literature [99]. We illustrate this idea again on the Google graph by showing the overall time for ten steps of the computation of personalized PageRank. Figure 2.3 shows how block updates improve the data access to computation ratio. On one thread, each extra sweep over the block increases the overall time by about 530 ms, while on eight threads, which use much more memory bandwidth than one thread, each extra sweep takes only about 20 ms. Thus, for computationally light applications updating data in the cache is cheap. As the amount of parallelism increases and each thread’s share of the available memory bandwidth decreases, we should iterate over vertices that are already in cache multiple times as long as this accelerates convergence.

### 2.3.3 Two-Level Scheduling

In addition to its improved cache behavior resulting from data locality, our block-oriented computation model also enables flexible scheduling of computation both at the block level and within each block. As discussed in Section 2.2, dynamic scheduling of vertex updates can improve convergence. However, this improvement comes at a cost. A vertex-oriented dynamic scheduler usually requires some scheduling metadata on each vertex, such as a scheduling priority value or a flag indicating whether the vertex is in the scheduling queue. Updating this metadata and querying it to make scheduling decisions can be expensive relative to a simple static scheduler that stores no scheduling metadata. For example, a prioritized scheduler maintains priority values for all the vertices and selects the highest priority vertex to be processed next. A common implementation uses a heap-based priority queue, and requires  $\Omega(\log |V|)$  time to update the vertex priority and to pop the highest priority vertex from the queue.

As a result, even if a dynamic scheduler performs fewer vertex updates than a static scheduler, the dynamic scheduler might yield worse overall performance due to the extra scheduling overhead. For example, the Fast-Marching method can be outperformed by the simpler Fast-Sweeping method on problems with constant characteristic directions, because the dynamic scheduling overhead of the Fast-Marching method outweighs its benefit of triggering fewer vertex update functions [25].

For the block-oriented computation model, however, scheduling decisions can be made at the block level instead of the vertex level. This greatly reduces the overhead of dynamic scheduling, since the scheduling metadata only needs

to be maintained per-block. Making scheduling decisions at the block level can be less accurate than making them at the vertex level, and thus result in more vertex updates before convergence, but for computationally light applications this is unlikely to be a problem, since the number of vertex updates alone does not determine the overall performance.

In addition to the block-level scheduler, which chooses the order in which blocks are updated to achieve fast global convergence, we may benefit from an inner-block scheduler that chooses the order in which vertices are updated within the scheduled block. Although high overheads make vertex-level dynamic inner-block schedulers unattractive for computationally light applications, some applications still benefit from various static inner-block schedulers. For example, alternating sweeping ordering within the block can improve the convergence of the fast sweeping method and the Bellman-Ford algorithm [66, 25].

In principle, any vertex scheduling strategy could be used in the inner-block scheduler. In practice, we prefer low-overhead strategies such as static scheduling and FIFO scheduling to keep the total scheduling overhead small.

## 2.4 Block-aware Execution Engine

We now present the design and implementation of a block-aware execution engine that follows the block-oriented computation model of Section 2.3. Our engine builds upon GRACE [110], a scalable parallel graph processing engine. In GRACE, users specify the application logic through a vertex update function as described in Section 2.2. Computation on the graph in GRACE proceeds in

iterations, and a subset of all vertices are processed during an iteration. The selection of the subset of the vertices and their order of processing within an iteration depends on a *scheduling policy* that GRACE also enables the user to define. Thus GRACE cleanly separates the application logic (defined through the vertex update function) from the computation strategy (defined through the scheduling policy).

Adapting GRACE to block-oriented computation in a shared-memory parallel environment poses several challenges. First, we want to isolate users from any low-level details of concurrency control inside the engine; users should simply write their application logic through the vertex update function, and the engine should handle all low-level details associated with parallelism. Second, we now have the opportunity to define two different scheduling policies, one at the level of blocks and another at the level of vertices within a block. In the remainder of this section we describe how we addressed these two issues in our execution engine.

### 2.4.1 Concurrency Control

As described in Section 2.3, we partition the input graph into blocks. We use the popular METIS [63] package for this. We split the iteration-based BSP model of GRACE into two levels, which we call *outer iterations* and *inner iterations*. In each outer iteration we process a subset of the blocks; and in each of these blocks we perform one or more inner iterations to execute the vertex update procedure on a subset (or the whole set) of the vertices of the block. In an outer iteration, each thread repeatedly chooses a block of the graph, reads it from

shared memory into its local cache, and then performs inner iterations before fetching the next block. The blocks are chosen without replacement, so a block can be processed at most once per outer iteration. Since blocks are generated from highly connected vertices, when the update procedure of a vertex needs to access its neighbor vertices, they are likely to be in the same block and hence already resident in the local cache, so for these vertices the threads can directly read their values without generating cache faults.

However, some of the neighbor vertices will be boundary vertices residing in other blocks (see Section 2.3.1). Thus simultaneous application of vertex update procedures inside different blocks can access the same vertex data, causing read/write conflicts. This is similar to the read/write conflicts in the vertex-oriented computation model when neighboring vertices are updated simultaneously, and synchronization is required to avoid such conflicts. This synchronization is usually done at the vertex level. However, for computationally light applications, such fine-grained synchronization introduces overhead comparable to the computation logic itself, significantly degrading performance.

For the block-oriented computation model, we can synchronize at the granularity of blocks instead of vertices. One naive approach would be to refrain from scheduling a block if any of its (incoming or outgoing) neighbor blocks is currently being processed. This locking-based scheme guarantees serializability but severely restricts parallelism: two threads cannot concurrently process vertices  $u$  and  $v$  if they belong to blocks  $B$  and  $B'$  that are neighbor blocks, even if  $u$  and  $v$  themselves are not neighbor vertices and so their update procedures could safely be applied in parallel.

To increase parallelism, our block-aware execution engine implements a sim-

ple form of multi-version concurrency control that allows neighboring blocks to be processed concurrently within an outer loop with guaranteed snapshot isolation [17]. To update a block  $B$ , we create a replica into which the updated block will be written. The thread computing over  $B$  writes into the newly created version of  $B$ , while it and other threads can read from the old version. In this way, reads and writes from different threads occur in different data versions, so no locking is required. After a thread finishes updating a block, the update is committed. Note that since edge data is read-only, we do not need to maintain multiple versions of the edges within the block. Since each block is processed at most once during each outer iteration, maintaining two versions of the data for each block is sufficient to implement this simple multi-version concurrency control. We maintain a bit for each block indicating which of its versions was most recently updated. When a thread begins processing a newly scheduled block  $B$ , it reads and caches the version bits of all the incoming neighbor blocks, and it reads from these versions while processing  $B$ . If a neighbor block  $B'$  is updated while  $B$  is being processed, it does not impact the processing of  $B$  because updates to  $B'$  are made to the version not being used to process  $B$ .

## 2.4.2 Scheduling

As GRACE separates the application logic from scheduling policies, it allows users to specify their own execution policies by relaxing data dependencies. The original GRACE system leverages the message passing programming interface to capture data dependencies. To efficiently support our block-aware execution engine with the underlying snapshot-based concurrency control protocol, we replace the GRACE programming interface by a remote read programming

interface. Nevertheless, the new GRACE system could also support flexible *vertex-oriented* execution policies with a similar customizable execution interface. GRACE's original vertex-oriented runtime maintains a dynamic *scheduling priority* on each vertex to support flexible ordering of vertex updates. Users can instantiate a set of runtime functions to define various scheduling policies by relaxing the data dependency implicitly encoded in messages. We adapt the same idea to let users specify their per-vertex scheduling policies in a remote access programming abstraction. Whenever a vertex  $u$  finishes its update procedure, the following user-specified function is triggered on each one of its outgoing vertices  $v$ :

```
void OnNbrChange(Edge e, Vertex src, Priority prior)
```

In this function, users can update vertex  $v$ 's scheduling priority based on the neighbor's old and new data. For example, to apply Dijkstra's algorithm for the shortest path problem, the vertex's scheduling priority can simply be set to its current tentative distance to the destination, and the above function can be implemented by updating the vertex priority value to the minimum of its current value and the newly updated distance via its changed neighbor:

```
void OnNbrChange(Edge e, Vertex src, Priority p) {  
    VtxData vdata = GetNewData(src);  
    double newDist = vdata.dist + e.cost;  
    if (newDist < GetDstData(e).dist)  
        p.Update(newDist, min);  
}
```

As in the original GRACE runtime, at the beginning of each iteration the

following function is triggered:

```
void OnPrepare(List<Priority> prior)
```

in which users can call engine-provided functions to dynamically select the subset of vertices to be updated in the next iteration. For example `ScheduleAll` can be used to schedule all the vertices that satisfy a user-provided predicate. We refer readers to the original GRACE paper [110] for a detailed description to the functions provided by the engine. As a concrete example, to implement Dijkstra’s algorithm, we can choose the single vertex with the smallest priority value in the `OnPrepare` function. Alternately, since little parallel work is available if we update only one vertex, we could schedule approximately  $r \cdot |V|$  of the vertices with smallest priorities by estimating a threshold from a sample of all the priorities and calling `ScheduleAll` to schedule those vertices with priorities below the estimated threshold. The following code shows the implementation:

```
void OnPrepare(List<Priority> prior) {  
    List<Priority> samples = Sample(prior, m);  
    Sort(samples, <);  
    double threshold = sample[r * m].value;  
    ScheduleAll(PriorLessThan(threshold));  
}
```

### **Block-Level Scheduling**

By instantiating the above `OnNbrChange` and `OnPrepare` functions, users can design flexible per-vertex policies in a remote access programming abstraction. Our block-aware execution engine automatically transforms these policies to

follow the block-oriented computation model. To do this, an addition scheduling priority is maintained for each block. Intuitively, a block update should have high priority when some (or most) of its vertices have high priority. Thus we could estimate the block scheduling priority by aggregating the scheduling priorities of vertices in the block. For example, one way of implementing a block-level Dijkstra-like algorithm would define the block priority as the minimum vertex priority. Then, when a block commits the update, the update function adjusts the priorities of its outgoing neighbors as well as its own priority. Algorithm 4 shows a straightforward block update function that maintains block-level priorities. The priorities of vertices in the block are maintained during the block update (line 6), and these are used to calculate the aggregated priority (line 9). When the block is committed, the priorities of outgoing boundary vertices are updated (line 12), and the new priorities are used to update the aggregated priorities of the outgoing blocks (line 15). Here, the function  $\text{VertexPriorAggr}(B)$  calculates the priority of block  $B$  from the vertex priorities.

Figure 2.4 illustrates the block level scheduling of a shortest path computation in a path with four blocks: the upper, left, middle and right block. Each block is a triangle with unit-weight edges, and each vertex is labeled with its current distance estimate. Block updates are scheduled in descending priority value. In the first state shown (Figure 2.4(a)) the upper block has just been updated, and priorities for the remaining three blocks have been calculated. The middle block has the lowest block priority, so it is scheduled next. When the update to the middle block is committed, the priorities of the neighboring blocks are adjusted, so that the left block now has a lower priority than the right block (Figure 2.4(b)). The left block is thus processed next, and once the update has

been committed, the state is as shown in Figure 2.4(c).

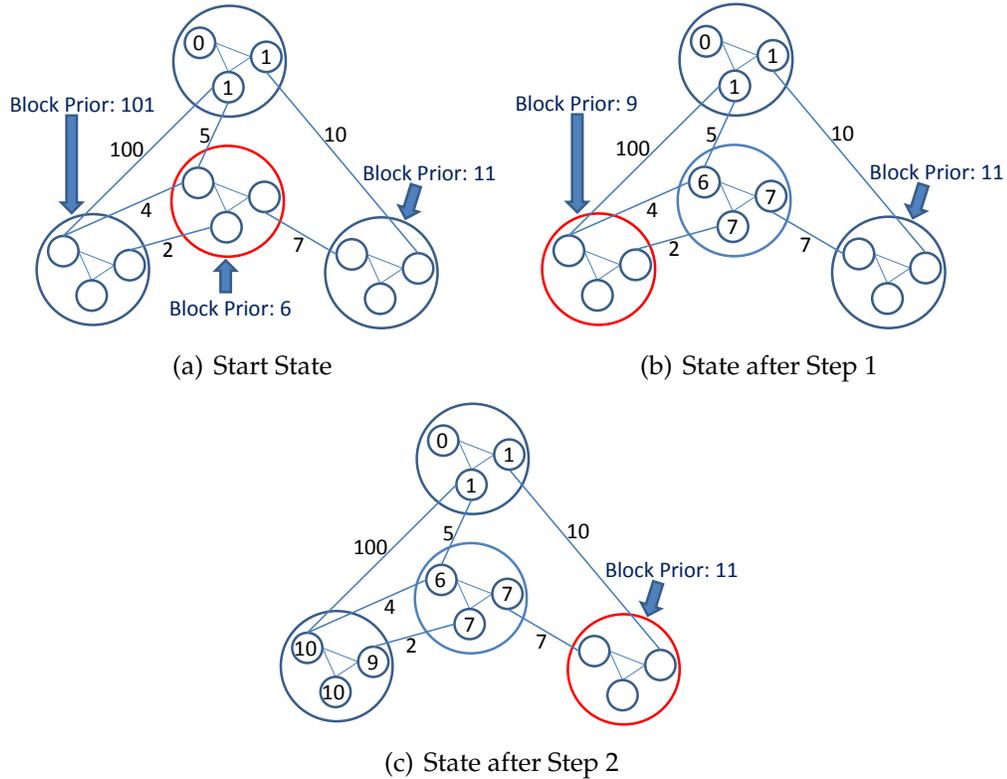


Figure 2.4: Illustration for Block Level Scheduling

Because it works at a coarse granularity and ignores potential dependencies among vertices, block-level scheduling is in general less accurate than vertex-level scheduling. However, block-level scheduling results in less scheduling overhead and better cache utilization, and researchers have successfully used block scheduling with block priorities defined by aggregation in several applications [25, 47, 113]. Our framework is general enough to support all the aggregation methods used in these papers. However, for some applications, these aggregates may not be suitable, and users will need to define application-specific block priority functions.

We note two general optimizations to this straightforward implementation.

---

**Algorithm 2.4:** Block Update with Priority Maintenance

---

**Input:** Block  $B$

- 1 **while** *Inner scheduling queue is not empty* **do**
- 2     Get a vertex  $u$  from inner scheduler ;
- 3     Reset  $u$ .prior ;
- 4     Perform vertex update on  $u$  ;
- 5     **foreach**  $e = (u, v) \in E(B), v \in V(B)$  **do**
- 6         OnNbrChange( $e, u, v$ .prior) ;
- 7     **end**
- 8 **end**
- 9  $B$ .prior = VertexPriorAggr( $B$ ) ;
- 10 Commit block update ;
- 11 **foreach**  $e = (u, v) \in E(B), v \in V(B'), B' \neq B$  **do**
- 12     OnNbrChange( $e, u, v$ .prior) ;
- 13 **end**
- 14 **foreach**  $B$ 's outgoing block  $B'$  **do**
- 15      $B'$ .prior = VertexPriorAggr( $B'$ ) ;
- 16 **end**

---

First, many aggregation functions can be maintained incrementally [50]. For example, to maintain the sum of the priorities, we could simply subtract the old priority from the sum and add the new priority into it. In such cases, line 9 and line 15 could be replaced by incremental updates. Second, maintaining vertex priorities in the block after each individual vertex update (line 6) is sometimes unnecessary. If the blocks are run to convergence – a reasonable choice for many applications, as our experiments will show – we do not need to maintain vertex

scheduling priorities during block update, which means line 6 could be skipped. Alternatively, if static inner scheduling is used for block updates, the vertex-level scheduling priorities need to be updated only in the last inner iteration. In other words, line 6 would only be executed for the last inner iteration. Both these optimizations are implemented in our system.

As a special case of the second optimization, we observed that for many applications the user-specified `OnNbrChange` function actually maintains the scheduling priority as an aggregation. For example, for the Dijkstra algorithm, the `OnNbrChange` function maintains the *MIN* aggregate over the tentative distances. If the same aggregate is used to define block scheduling priority the runtime can skip updating the vertex priority and update the block priority directly when the second optimization is enabled. To implement this, lines 3, 9 and 15 would be skipped. Line 6 would change to `OnNbrChange(e, u, B.prior)` and line 12 would change to `OnNbrChange(e, u, B'.prior)`. This optimization is supported by the GRACE engine, but must be enabled explicitly.

Using the block priorities, it is straightforward to dynamically select a subset of blocks for each iteration. The engine simply passes the scheduling information to the user-defined `OnPrepare` to decide the blocks scheduled for the next iteration.

### **Inner-block Scheduling**

As discussed in Section 2.3, in addition to block-level scheduling we can have low-overhead vertex schedulers within a block. We currently support both static and dynamic inner-block scheduling. To use them, users just need to spec-

ify the inner scheduling policy and the parameters.

**Inside-Block Ordering.** Inside a block, vertex updates are done sequentially. Since only one assigned thread is responsible for updating the block, there are no update conflicts inside the block. As each update is immediately visible to later updates, the update ordering for each inner iteration can make a significant difference. Currently, we provide two pre-defined static inner schedulers: fixed-order sweeping and alternating sweeping. For fixed-ordering sweeping, the engine updates vertices in the same order in each inner iteration, while alternating sweeping reverses the order between inner iterations.

**Dynamic Inner Scheduling.** Instead of statically sweeping over the vertices inside the block for a fixed number of sweeps or until convergence, we may use dynamic inner scheduling, potentially eliminating update function calls for vertices whose neighbors have unchanged data. To this end, we have implemented a low-overhead dynamic inner scheduler. Each block has a queue of vertices whose neighbor data has changed since the last update. Every incoming boundary vertex also has an extra outer scheduling bit. For a given block  $B$  and incoming boundary vertex  $v$ , this outer scheduling bit is set if the data on any of  $v$ 's incoming neighbors outside block  $B$  have changed.

When block  $B$  is chosen to be updated, all the vertices with outer scheduling bits set are added to the queue. The GRACE engine then repeatedly pops a vertex  $u$  from the queue and invokes the update function on it. If the vertex data of  $u$  changes, the engine pushes all of  $u$ 's outgoing vertices inside the block  $B$  onto the queue. The block update continues until the queue is empty, or a pre-defined maximum number of updates is reached. In the latter case the vertices remaining in the queue wait for the next time that block  $B$  is chosen to be

updated.

Once block  $B$  commits, the GRACE engine iterates over all outgoing boundary vertices  $u$  that have changed, setting the outer scheduling bit for all vertices  $v$  s.t.  $(u, v) \in E, v \notin B$ .

**Maximum Number of Inner Iterations.** We need to decide the maximum number of inner iterations per block update. At one extreme, we can perform only a single iteration; i.e., update each vertex once per block update. This is similar to the traditional vertex execution model, except it yields better cache performance because we usually read the same vertex data repeatedly when updating a block. At the other extreme, we can set this number to infinity, so that each block is repeatedly updated until it converges for its current boundary values. How to make this choice depends on the ratio of data access to computation cost and on how much additional inner iterations benefit convergence. If little computation is done for each byte fetched from memory and each inner iteration significantly accelerates global convergence, the number of inner iterations should be large. On the other hand, if the computation is heavy, or if additional inner iterations do not accelerate global convergence very much, the maximum number of iterations should be small.

Given the fixed boundary data  $S_{NV(B)}$  and  $S_{NE(B)}$ , it is possible for a block to converge before the maximum number of inner iterations is reached. In this case the block update can be terminated. This situation is likely when the global graph data is close to convergence.

## 2.5 Experiments

We added block updates to GRACE, a shared-memory parallel graph processing framework implemented in C++ with pthreads [110]. We did not change its vertex-oriented programming model, but modified the runtime as described in Section 2.4 to support block-aware execution. Although this preliminary implementation is not (yet) a general engine – currently the block level schedulers are manually coded, based on the scheduling policies used in the experiments – it can already demonstrate the key techniques described in Section 2.4 and hence be used to validate their performance benefits.

The original GRACE runtime provides two dynamic scheduling policies: Eager and Prior. Users need only provide an application-specific function to compute priorities. Both policies schedule only a subset of the vertices in each tick: for the Eager policy, a vertex is scheduled if a neighbor’s data has changed; and for the Prior policy, only the  $r \cdot |V|$  highest-priority vertices are scheduled for update, where  $r$  is a configurable selection ratio. Both of them are extended to be executed with the block-oriented computation model as we discussed in Section 2.4.

Our experimental evaluation has three goals. First, we want to verify that our block-oriented computation model can improve end-to-end performance compared with the vertex-oriented computation model. Second, we want to show that this end-to-end performance gain comes from both better cache behavior and lower scheduling overhead. Last, we want to evaluate the effect of inner-block scheduling policies on the convergence rate.

## 2.5.1 Applications

**Personalized PageRank.** Our first application, Personalized PageRank (PPR), is a PageRank computation augmented with a personal preference vector [58]. We randomly generated a sparse personalization vector in which all but 1% of the entries are zeros. The standard iterative algorithm is a Richardson iteration for solving a linear system. The natural algorithm in GRACE is a Jacobi or Gauss-Seidel iteration, while the natural approach in our block-aware execution engine is a block Jacobi or block Gauss-Seidel iteration [41, 16]. We declare convergence when all the vertex updates in an iteration are less than a tolerance of  $10^{-3}$ .

**Single-Source Shortest Path.** Our second application is Single-Source Shortest Paths (SSSP), where each vertex repeatedly updates its own distance based on the neighbors' distances from the source. GRACE's original vertex-oriented execution with eager scheduling corresponds to the Bellman-Ford algorithm, and its prioritized scheduling corresponds to Dijkstra's algorithm. We are not aware of any algorithms in the literature that correspond to approaches in our block-aware execution engine, though there has been work on similar blocked algorithms for the related problem of solving the eikonal equation [25]. All the variants of this algorithm converge exactly after finitely many steps, and we declare convergence when no vertex is updated in an iteration.

**Etch Simulation.** Our third application is a three-dimensional Etch Simulation (Etch Sim) based on an eikonal equation model [97]. The simulation domain is discretized into a 3D grid and represented as a graph. The time at which an etch front passes a vertex can be computed iteratively based on when the front reaches its neighbors. The vertex-oriented execution engine with eager schedul-

ing in GRACE corresponds to the Fast Sweeping method for solving the equations, while its original prioritized scheduling corresponds to the Fast Marching method. Our block-aware execution engine with block-level eager scheduling corresponds to the Fast Sweeping method with Domain Decomposition [99], while our block-aware execution engine with block-level prioritized scheduling corresponds to the Heap-Cell method [25]. As with SSSP, all these algorithmic variants converge exactly after finitely many steps, and we declare convergence when no vertex is updated in an iteration.

## 2.5.2 Experimental Setup

**Machine.** We ran all our experiments using an 8-core computer with 2 Intel Xeon L5335 quad-core processors and 32GB RAM.

**Datasets.** Table 2.2 summarizes the datasets we used for our applications. For PPR, we used a coauthor graph from DBLP collected in Oct 2011, which has about 1 million vertices and 7 million edges. We also used a web graph released by Google, which contains about 880,000 vertices and 5 million edges. For Shortest Path, we used a social graph from LiveJournal with about 5 million vertices and 70 million edges. For the Etch Simulation application, we constructed a 3D grid that has  $120 \times 120 \times 120$  vertices. Finally, we demonstrate the performance of our system on a larger example, a web graph of the .uk domain crawled in 2002. This graph contains about 18 million vertices and 300 million edges. Vertices are ordered randomly in all the datasets except the 3D grid dataset.

**Partition Time.** For the DBLP, Google, and LiveJournal graphs, we used METIS [63] to partition the graph into blocks of around 100. For the .uk web

Table 2.2: Dataset Summary

<b>Data Set</b>	<b># Vertices</b>	<b># Edges</b>	<b>Partition Time(s)</b>	<b>Application</b>
DBLP	967,535	7,049,736	38	PPR
Web-Google	875,713	5,105,039	34	PPR
LiveJournal	4,847,571	68,993,773	659	SSSP
3D Grid	1,728,000	9,858,192	N/A	Etch Sim
UK02	18,520,486	298,113,762	1034	PPR

graph, we partitioned into blocks of size 400. We report the partitioning times in Table 2.2. While partitioning is itself expensive, our focus is problems where the partitioning work will be amortized over many executions of the main computation. Given that relatively small blocks appear useful in practice, recently-developed fast algorithms for bottom-up graph clustering [100] may perform better on these problems.

**Scheduling.** As mentioned in Section 2.4, static scheduling and two common dynamic scheduling policies (Eager and Prior) are implemented in the GRACE runtime. To use Prior scheduling, users must also provide the application-specific priority calculation. For the Eager scheduling policy, the scheduling priority for a vertex is a boolean value indicating whether any neighboring vertex data has changed. Thus we use boolean OR as the block priority aggregation, which means a block would be scheduled if its boundary data has changed, or its last update did not run to convergence. For the Prior scheduling policy, each vertex holds a float value to indicate its priority. The priority aggregation used for SSSP and EtchSim is MIN in our experiments, while the aggregation used for

PPR is SUM. Notice that since we use MIN for SSSP, it is eligible for the direct block-priority update optimization described in Section 2.4.

### 2.5.3 Results

#### Block Size

The Etch Simulation application has a natural grid structure, and we used subgrids of size  $b \times b \times b$  as blocks, while for other applications we used METIS to partition the graph into blocks of roughly equal size. The best block size depends on many factors, including characteristics of the machine, characteristics of the data and application, how the graph is partitioned, and how block updates are scheduled. In our applications, the performance was only moderately sensitive to the block size, as we illustrate in Figure 2.5. Because block sizes between 100 and 400 performed well for these examples, we chose a default block size of 100 for the PPR and SSSP test cases except the UK dataset, and used a  $5 \times 5 \times 5$  sub-grid as a block for the Etch Simulation. For the much larger UK dataset, we set the block size to be 400.

#### End-to-End Performance

To see how the block-oriented computation model performs compared to the vertex-oriented computation model, we ran each of our example applications under different scheduling policies. The mean run times for the scheduling policies are shown in Figure 2.6; the bars labeled Vertex and BlockCvg correspond respectively to applying this schedule to individual vertex updates and to block

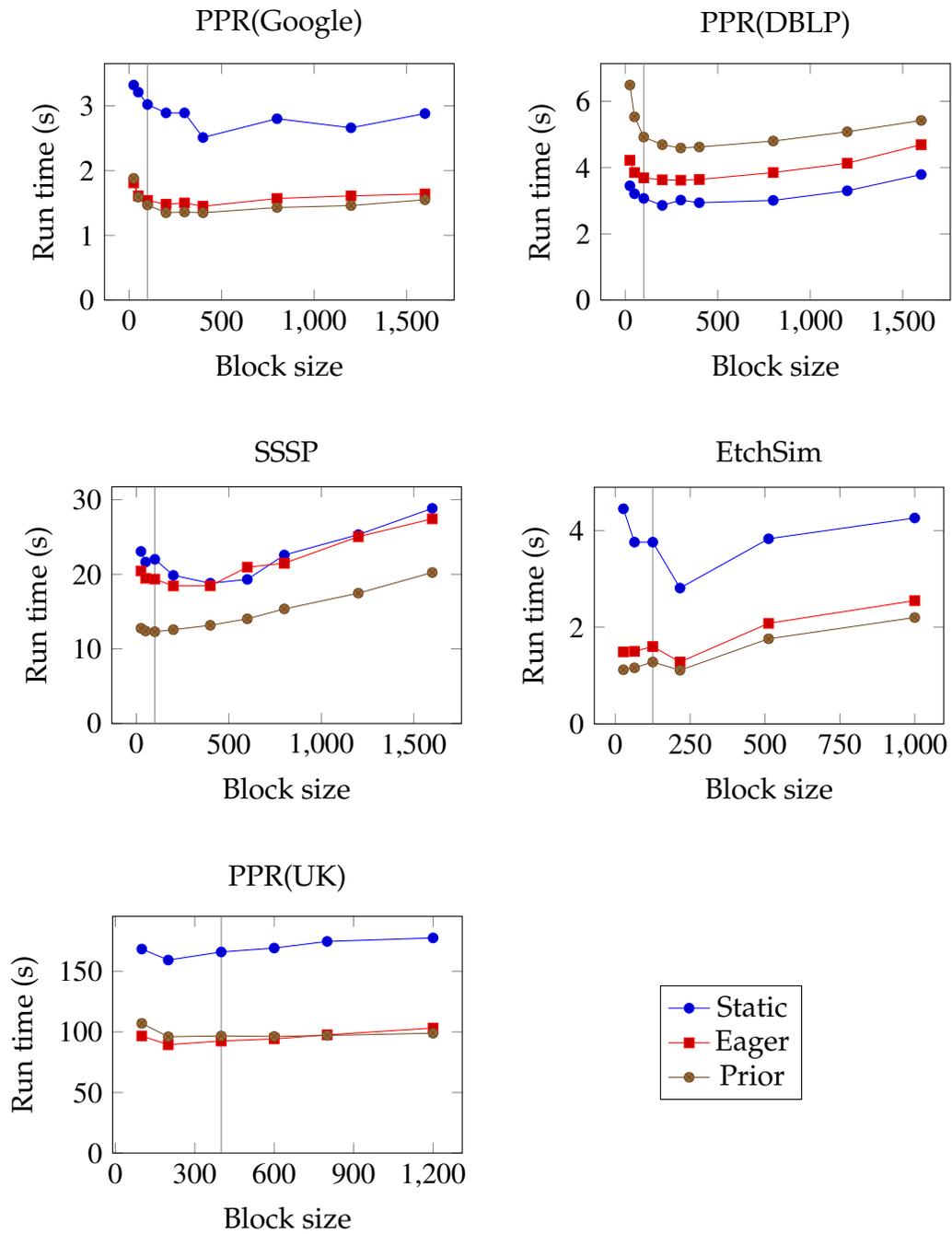


Figure 2.5: Effect of block size.

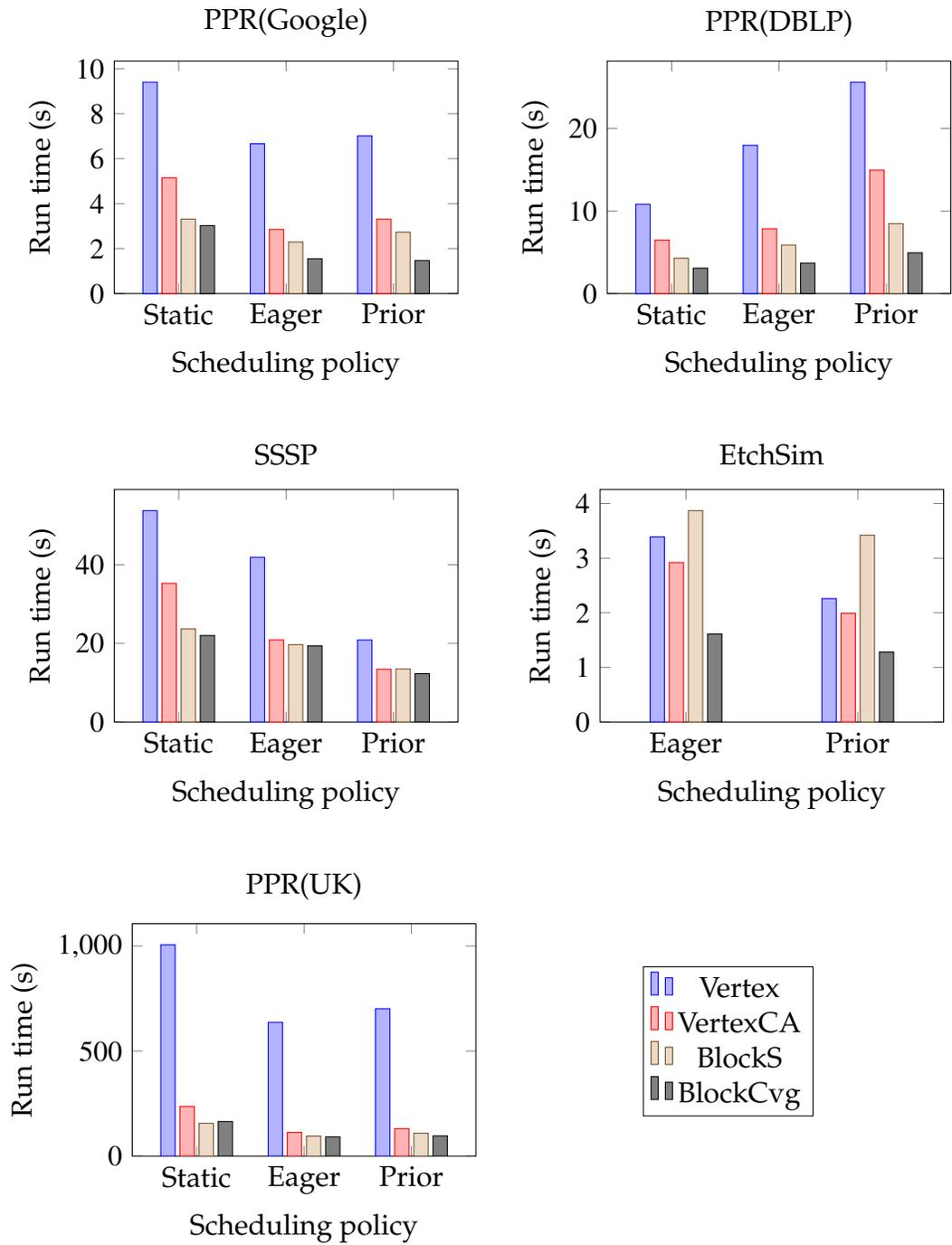


Figure 2.6: Effect of block strategy.

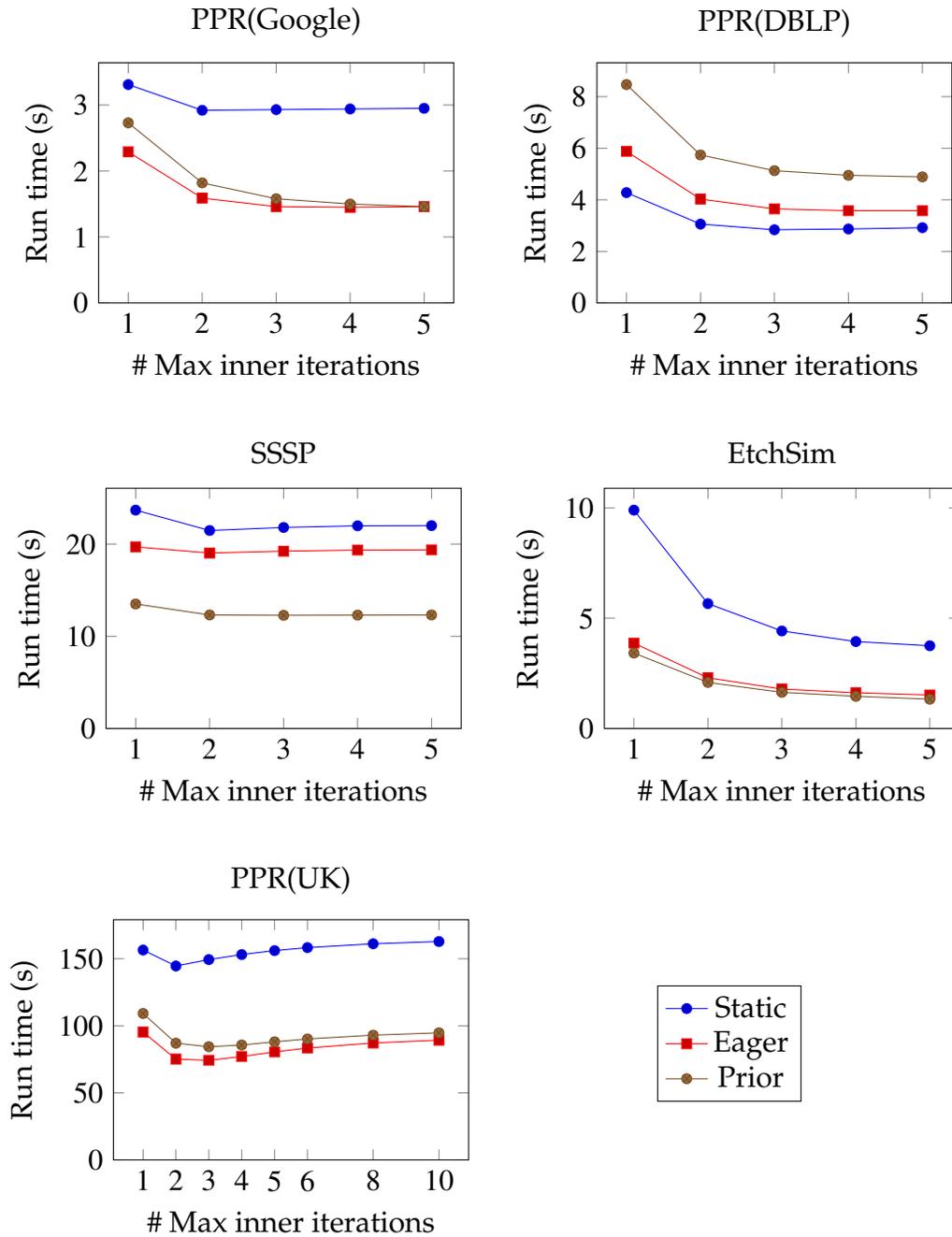


Figure 2.7: Effect of static inner scheduling.

updates. Here when a block is scheduled, each vertex is repeatedly updated until the block data converges. We omit the time for the static scheduling policy for the Etch Simulation application from Figure 2.6. While a well-chosen static schedule leads to very fast convergence for this problem [121], the naive static schedule in our current implementation takes much longer than the two dynamic scheduling policies: 15.9s for the vertex model and about 3.6s for the block model.

In our experiments, the best scheduling policy under the vertex model is also the best scheduling policy under the block model. More specifically, for the SSSP and Etch Simulation applications, the best scheduling policy is the prioritized policy. For the PPR application, the best scheduling policy depends on the dataset characteristics. On the Google and UK graph, dynamic scheduling performs better than static scheduling, while on the DBLP graph, the static scheduling policy performs best. This is because the DBLP graph has a much larger clustering coefficient than the Google and UK graph. Thus the computational savings due to dynamic scheduling policies are smaller and are outweighed by the high overhead of the dynamic schedulers themselves. Moreover, dynamic scheduling policies tend to schedule vertices with higher degrees in the PPR application, which makes the vertex updates more expensive.

In general, the blocked computation outperforms the corresponding vertex-centric computation under each scheduling policy. In the PPR application, our block engine runs  $3.5\times - 7.0\times$  faster than the vertex-centric computation for the best scheduling policy. For the SSSP and Etch Simulation applications, we cut the run time roughly in half. Also, we observed that the block-oriented computation model is more robust to the “wrong choice” of scheduling policy. For

example, the vertex-centric prioritized scheduler is about  $2.5\times$  slower than the vertex-centric static scheduler, but the block prioritized scheduler is about 60% slower than the block static scheduler. This is because by scheduling blocks rather than vertices, we significantly reduce the total scheduling overhead.

### **Analysis of Block Processing Strategies**

Recall that the block-oriented computation model has three main benefits: (1) It has a better memory access pattern due to visiting vertices in the same block together. (2) It has reduced overhead for isolation due to providing consistent snapshots at block level instead of at vertex level. (3) It can achieve better cache performance by doing multiple iterations in a block. To understand how each of these benefits contributes to improved end-to-end performance, we analyzed the run time for each application in two execution models that are hybrids of the pure vertex-oriented computation model and the pure block-oriented computation model used in general performance comparison. We show the running times of all these schedulers in Figure 2.6.

To understand how the memory access pattern affects performance, we used the cache aware vertex-oriented computation model (VertexCA) introduced in Section 2.2. Recall this execution model still updates one vertex at a time, and makes scheduling decisions at the vertex level. However, it is aware of the graph partitioning and updates the vertices in block order; i.e., it updates all the scheduled vertices of a given block before proceeding to the next block. By doing so it achieves better temporal locality. We also report the running time for two different inner-block schedulers: the simple block model, which just sweeps all the vertices once (BlockS), and the convergent block model, which iteratively

updates the block data until it converges (BlockCvg). Note that the major difference between the simple block model and the cache-aware vertex model is the isolation and scheduling overhead. GRACE uses snapshot isolation; thus, before updating a vertex or block the engine is responsible for choosing the right version of the data to be passed to the update function. By making this decision at the block level rather than at the vertex level, the average overhead is greatly reduced. Finally, the difference between the simple block model and the convergent block model is that the CPU is able to do more work on data residing in the cache when the memory is saturated.

With the cache-aware vertex model, we observed a significant reduction in running time of the PPR and SSSP applications, achieving savings of 36% to 52% on the best scheduling policy except for the UK graph. We observed much more saving on the larger UK graph, in which the cache-aware vertex model is more than 5 times faster than the vertex model. In contrast, the cache-aware vertex model only saves about 10% of the time for the prioritized policy in Etch Simulation application. This is because the grid-structure graph used in the Etch Simulation already has a regular memory access pattern, while the memory access pattern for arbitrary graphs could be quite random.

Switching from the cache-aware vertex model to the convergent block model, we save about half the time for PPR on the DBLP and Google graphs. We see similar savings for the Etch Simulation application, but for different reasons. For PPR, performance improves from the cache-aware vertex model to the simple block model and finally to the convergent block model. However, for Etch Simulation, the simple block model has worse performance than the cache-aware vertex model, while the convergent block model has much better

performance. This is because scheduling at the block level wastes many vertex updates in this case. For PPR on the UK graph, switching from VertexCA to BlockCvg reduces the running time by about 20% for the eager scheduling policy, which is not as significant as PPR on other two datasets. This is because running until convergence inside a block gives only a slight improvement. Our experiments show an inexact block solve improves the overall running time on this dataset. For SSSP, BlockS has roughly the same running time as VertexCA, while BlockCvg only improves the performance by 10%. This is because social networks obeying a power-law are hard to partition – in the partitions we used in our experiments, more than half the edges are cutting edges. Thus updating the block until convergence contributes little to achieving global convergence. Researchers have shown that overlapping partitions would help this problem [8], and some recent emergent graph processing frameworks such as PowerGraph [42] have designed their programming interfaces to naturally support computation on overlapping partitions. We expect this block computation model would have more benefits on social graph computations for these frameworks. We also observed that the direct block-priority update optimization to BlockCvg reduces the running time from 12.3 seconds to 10.5 seconds, and makes BlockCvg 22% faster than VertexCA.

### **Effect of Inner-Block Scheduling**

In this subsection we focus on the effect of inner-block scheduling. In particular, we have seen that updating each vertex multiple times in a single block update often improves performance. For example, if the boundary data of the block has already converged, then iterating over the vertices until the block data

converges is a natural way to define the block update function. However, this could be a poor choice if the boundary data of the block is incorrect. The tradeoff is that doing more updates inside the block leads to better cache performance, but it could waste CPU time if the boundary data has not converged. As we mentioned in Section 2.4, we can set a maximum number of inner iterations  $I_\theta$ , and terminate the block update after  $I_\theta$  sweeps even if the block has not yet converged.

To understand this tradeoff, we plot running time against the number of inner-block iterations  $I_\theta$  in Figure 2.7. For PPR, the best performance occurs around  $I_\theta = 3$ , and after that the running time remains the same as running until convergence for both DBLP and Google datasets. However, for the UK dataset, further increasing  $I_\theta$  increases the overall run time significantly. Specifically, running until convergence is 24% slower than setting  $I_\theta = 3$ . On the other hand, for the Etch Simulation application, more inner iterations always yields better performance. We believe that besides the application characteristics, the higher diameter of the graph also favors more inner iterations because they help information propagate across the graph faster.

Dynamic scheduling may also be used inside blocks to reduce the number of updates, at the cost of paying some extra scheduling overhead. To study this tradeoff, we compare the run times for static and dynamic inner scheduling in Figure 2.8. For all three applications, dynamic inner scheduling reduces the number of vertex updates; nonetheless, static scheduling outperforms dynamic scheduling for the SSSP problem, because the computational saving is outweighed by the scheduling overhead. For PPR, we observed that dynamic inner scheduling is slightly faster than the static inner scheduling on Google

graph, while it is slower than the static inner scheduling on the DBLP graph. However, dynamic inner scheduling yields nearly 25% improvement in the Etch Simulation application, as the vertex update function is slightly computationally heavier than the previous two applications and the convergence for this problem is particularly sensitive to update order.

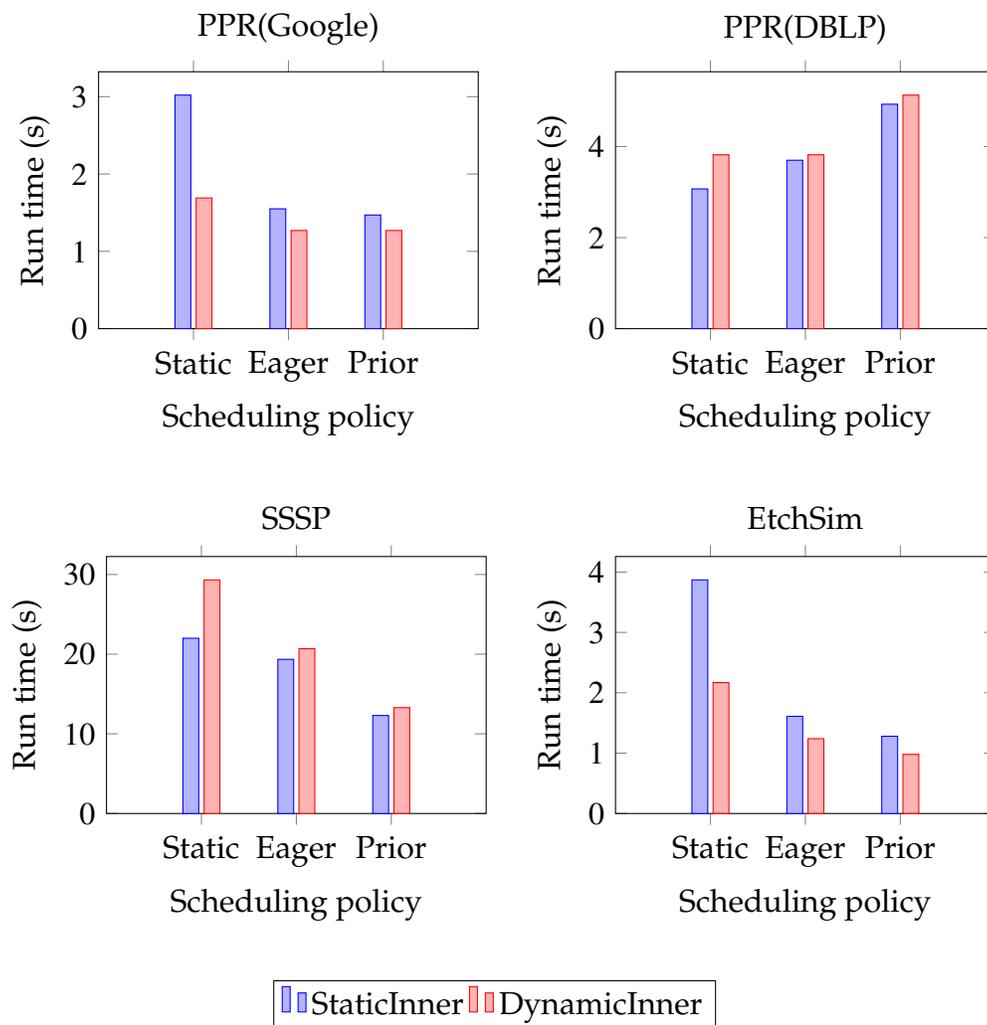


Figure 2.8: Effect of dynamic inner scheduling.

## Comparison with GraphLab

To evaluate the performance of the vertex execution model implemented in GRACE, we compare the running time of GRACE with the GraphLab shared-memory multicore version on all three applications under different scheduling policies. Recall that GraphLab provides two different isolation levels for concurrent vertex updates: vertex consistency, which allows two neighboring vertices to update simultaneously, and edge consistency, which guarantees serializability of updates. As pointed out by the GraphLab authors, for some graph applications vertex consistency can produce inaccurate results, as it does not avoid read/write conflicts on neighboring vertices [75, 76]. We report GraphLab’s run time under both isolation levels in Figure 2.9.

For Eager and Prior scheduling, GRACE’s run time is between that of GraphLab with vertex consistency and GraphLab with edge consistency. This is because the isolation level used by GRACE – snapshot isolation – is between GraphLab’s vertex consistency and GraphLab’s edge consistency. The only exception is the Eager scheduling for eikonal equation, in which GRACE is faster than GraphLab whether vertex or edge consistency is used. This is because the corresponding scheduler in GraphLab executes more than twice as many updates as GRACE.

For Prior scheduling, GRACE is always faster than GraphLab, because it makes the scheduling decision by iterations rather than by vertex [110]. Thus we expect the block model could benefit GraphLab even more than GRACE, since GraphLab has higher scheduling overhead.

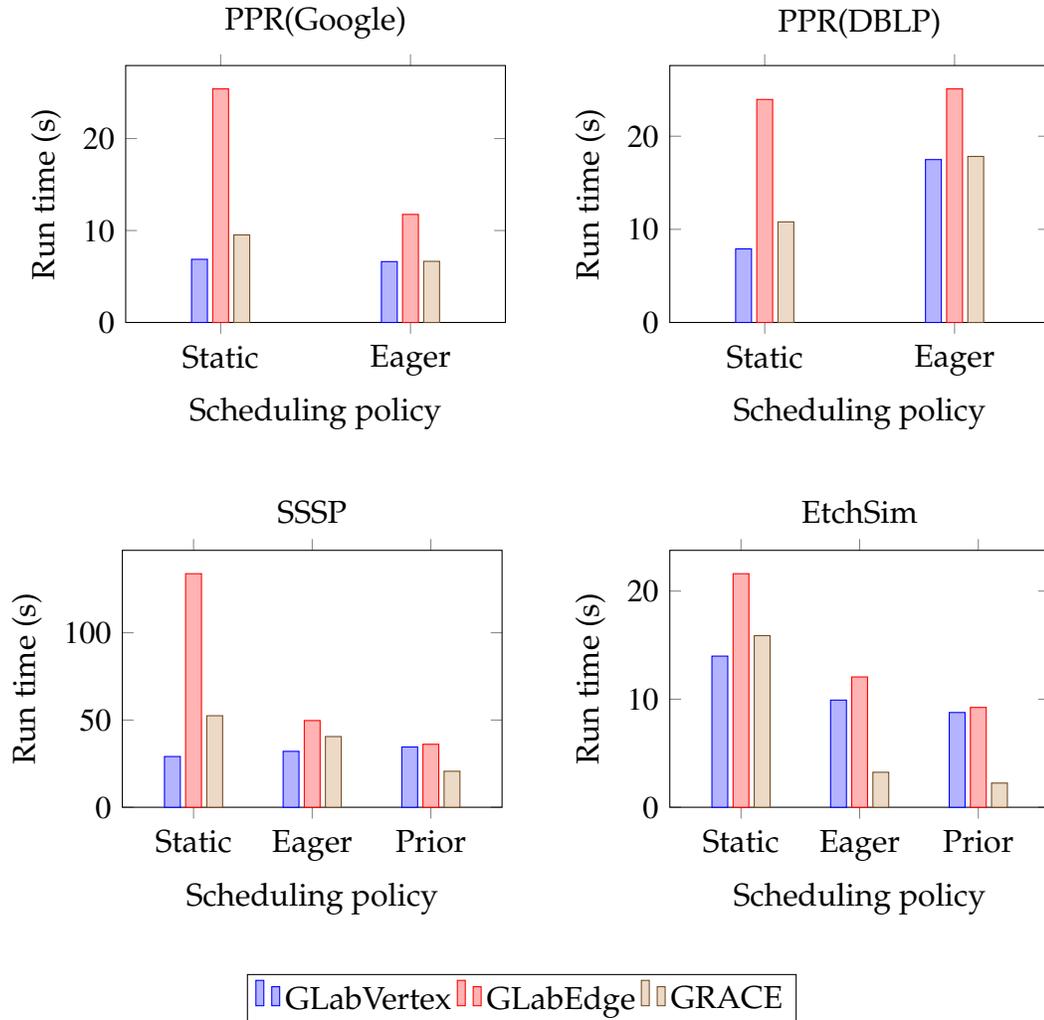


Figure 2.9: Effect of scheduling policies in GraphLab and in GRACE.

## 2.6 Related Work

Most existing graph programming frameworks can be categorized into two groups. The first group are mainly designed for distributed memory environments and are based on the Bulk Synchronous Parallel (BSP) [105] model, which allows processors to compute independently within each iteration and uses global synchronization between iterations for processors to communicate. Example frameworks in this group include PEGASUS [62], Pregel [77],

PrIter [120], AsyncMR [60] and Naiad [78], which are mostly built on MapReduce or DryadLINQ with a higher level programming model. The second group of frameworks are mainly designed for multi-core shared memory architectures and usually do not apply global synchronization barriers for threads to communicate. Instead, threads can proceed asynchronously, which enables various scheduling of computation tasks to achieve better convergence. Consistency is guaranteed either by a fine-grained locking mechanism to synchronize shared data access or by requiring all operations to be commutative and able to be rolled back in case of a data race. Frameworks in this group include GRACE [110], GraphLab [75], GraphChi [70] and Galois [68]. One note here is that GraphChi is designed to store graphs out-of-core and its “sliding window” idea bears some resemblance to our cache-aware approaches.

Nevertheless, both these groups provide a local vertex computational interface to users to code their application logic, and both groups execute the coded applications in a per-vertex update manner. As we have illustrated in Section 2.1, such a per-vertex update model, while appropriate for programming, is not an ideal execution model for computationally light applications due to its poor locality and high demand for memory bandwidth.

The idea of having more local computations to reduce the communication overhead has also been suggested in the AsyncMR framework [60]. However, since the AsyncMR framework follows the MapReduce model, the main communication overhead is global synchronization. We are applying this optimization technology to GRACE, an asynchronous framework in which global synchronization is not the main communication overhead. In addition, because of its underlying MapReduce framework, AsyncMR does not support flexible

dynamic outer scheduling.

Blocking is widely used in high-performance computing to improve memory access patterns. A textbook example is blocked matrix multiplication [85, Section 5.3], which is the basic building block for high-performance dense linear algebra libraries like LAPACK [9]. For large problems, such cache-blocked dense linear algebra codes typically use  $O(\sqrt{M})$  floating point operations per cache miss, where  $M$  is the cache size [34]. Similar optimizations apply to graph algorithms such as Floyd-Warshall that are structurally similar to dense linear algebra operations [84]. In contrast, sparse matrix algorithms have relatively irregular memory access patterns, and it is more difficult to block them for efficient cache use. Recent research addresses this to some extent by automatically reorganizing the graph data structures used to store sparse matrices in order to optimize key operations such as sparse matrix-vector multiplication [57, 109, 83]. Blocking is also used for improved locality and convergence in many iterative solvers for linear and nonlinear equations and optimization problems; examples include block Jacobi and block Gauss-Seidel methods for accelerating iterative solvers [16], domain decomposition and substructuring methods used in linear and nonlinear PDE solvers [99], and block coordinate descent methods in optimization [20], etc. Because the local block updates are relatively expensive, these methods often achieve good communication-to-computation ratios [49].

CHAPTER 3  
DYNAMIC INTERACTION GRAPHS WITH PROBABILISTIC EDGE  
DECAY

A large scale network of social interactions, such as mentions in Twitter, can often be modeled as a “dynamic interaction graph” in which new interactions (edges) are continually added over time. Existing systems for extracting timely insights from such graphs are based on either a cumulative “snapshot” model or a “sliding window” model. The former model does not sufficiently emphasize recent interactions. The latter model abruptly forgets past interactions, leading to discontinuities in which, e.g., the graph analysis completely ignores historically important influencers who have temporarily gone dormant. We introduce TIDE, a distributed system for analyzing dynamic graphs that employs a new “probabilistic edge decay” (PED) model. In this model, the graph analysis algorithm of interest is applied at each time step to one or more graphs obtained as samples from the current “snapshot” graph that comprises all interactions that have occurred so far. The probability that a given edge of the snapshot graph is included in a sample decays over time according to a user specified decay function. The PED model allows controlled trade-offs between recency and continuity, and allows existing analysis algorithms for static graphs to be applied to dynamic graphs essentially without change. For the important class of exponential decay functions, we provide efficient methods that leverage past samples to incrementally generate new samples as time advances. We also exploit the large degree of overlap between samples to reduce memory consumption from  $O(N)$  to  $O(\log N)$  when maintaining  $N$  sample graphs. Finally, we provide bulk-execution methods for applying graph algorithms to multiple sample graphs simultaneously without requiring any changes to existing

graph-processing APIs. Experiments on a real Twitter dataset demonstrate the effectiveness and efficiency of our TIDE prototype, which is built on top of the Spark distributed computing framework.

### 3.1 Introduction

In a world of booming social networking services and pervasive mobile devices, electronic records of social interactions between people are being generated at ever-increasing rates. A network of social interactions often can be modeled as a “dynamic interaction graph” in which new interactions, represented by edges, are continually added. Examples include phone-call graphs generated by telecommunication service providers, message graphs from social networking sites, and mention-activity graphs formed by Twitter users mentioning one another in their tweets. Dynamic interaction graphs are very different from traditional social graphs, such as the friendship graphs from social networking sites, in which the social relationships evolve gradually. Real-life social interactions, such as phone calls and tweets, happen much more rapidly. For example, as of January 2014, 58 million tweets were generated daily on Twitter. In essence, a dynamic interaction graph can be viewed as a data stream of interactions.

Enterprises are analyzing streams of interactions for insights relevant to real-time decision making. This poses a significant challenge to algorithm design, because the overwhelming majority of graph algorithms assume static graph structures. As a result, most existing systems designed for graph stream analysis [27, 78, 80] successively process a sequence of static views, or “snapshots”, of a dynamic graph, where a snapshot comprises all interactions seen so far. As time advances, the result is updated incrementally, if possible, or else by re-

running the algorithm from scratch. We call this simple model the “snapshot model”.

A key drawback of this approach is the ever-increasing size of the snapshots. Graph analysis is usually much more complex than maintenance of simple aggregates over a stream of data, and the memory usage of virtually all available graph algorithms increases with increasing graph size. As a result, computation and memory resources quickly run out as interactions are added to the dynamic graph. Another drawback of the snapshot model is the *recency* problem: as time progresses, the proportion of stale data in the snapshot becomes ever larger and analysis results increasingly reflect out-of-date characteristics of the dynamic graph.

One simple approach to reducing the size of the snapshots and enforcing recency requirements is to use a “sliding-window” model, where only recent interactions that happen within a small fixed-size time window are considered in the analysis. This simplistic cut-off approach completely forgets historical interactions and thus loses the *continuity* of the analytic results with time. Historical interactions may be less relevant to today’s decision making, but do not completely lack value, especially in the aggregate. The following example demonstrates the drawbacks of the snapshot and sliding-window models.

**Example 1** (Influence Analysis). *An advertising company is analyzing the mention-activity graph from Twitter to identify key influencers with respect to skiing equipment. A key influencer is someone who has many interactions with other users on skiing-related topics. Consider the following three users:*

- *Alice joined Twitter five years ago and has been regularly and frequently interacting with other users on skiing-related topics since then. She has been inactive for the*

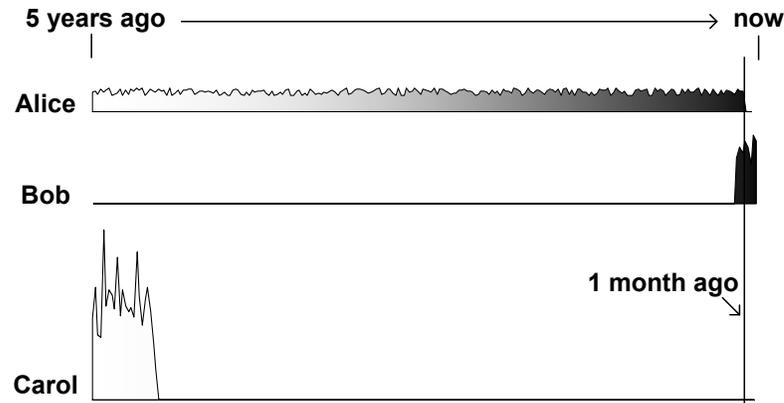


Figure 3.1: Influence analysis example

*last three weeks because she is on a skiing trip in Europe.*

- *Bob joined Twitter two months ago, and since then has had many interactions on skiing related topics.*

- *Carol also joined Twitter five years ago. She was extremely active on skiing topics for the first six months, but then lost interest and has never tweeted about skiing again.*

*Figure 3.1 illustrates the frequency of interactions for the three users over time. Intuitively, Alice is a steady influencer who is temporarily dormant. Bob can be viewed as a rising star. Although it is unknown whether Bob can maintain his influence in the future (instead of becoming another Carol), as of now, he should be considered a target for ads. Obviously, Carol is not an influencer at present.*

*Under the snapshot model, there is no distinction between recent interactions and old ones. Alice, correctly, is an influencer. Bob, incorrectly, is not an influencer because the number of his interactions is relatively small compared to the cumulative interaction counts of the old-timers. On the other hand, the fact that Carol has had a huge number of interactions incorrectly makes her an influencer, even though all of these interactions are in the remote past (but perhaps she should receive an ad just in case). By comparison, if we use a sliding-window model with a one-month window length, Bob will be an*

*influencer, but Alice will not be considered as an influencer at all. Carol will never receive an ad.*

To address the above issue, we take an approach inspired by the literature on sampling from data streams. Specifically, we consider *temporally biased sampling*, as was proposed for ordinary (non-graph) data streams in [6]. The general idea is to sample data items according to a probability that decreases over time, so that the sample contains a relatively high proportion of recent data points. In our example, the gray levels in Figure 3.1 illustrate temporally biased sampling rates (darker shades correspond to higher inclusion probabilities). Carol’s historical interactions will be significantly downgraded in the influence analysis (but not completely ignored). Bob’s recent interactions will be valued more. Although Alice is not active right now, her consistent interactions throughout time help her maintain influence.

Temporally biased sampling is especially appealing for analyzing dynamic interaction graphs. First, sampling deals gracefully with the increasing size of a dynamic graph. Second, temporal biasing creates samples with more recent interactions (recency) yet still keeps some old interactions to provide the necessary context for the analysis (continuity). Finally, users can apply any existing algorithm for static graphs as-is, avoiding the need to design new, even more complex algorithms that attempt to satisfy recency and continuity requirements.

Although the idea of temporally biased sampling is not new, we are the first to apply it to the analysis of dynamic graphs. In particular, as discussed in what follows, we refine the generic temporally biased sampling approach by exploiting graph-specific properties—especially the overlapping of edges between graphs—to achieve space and time efficiencies. We also describe challenges and

solutions when building a distributed system to support this important new functionality.

We formalize temporally biased sampling for dynamic graphs via a *probabilistic edge decay* (PED) model. Under this model, we sample interactions from the current snapshot. Each interaction has an independent probability of appearing in the resulting sample graph, and this probability is non-increasing with the age of the interaction. The PED model subsumes both snapshot and sliding-window models; see Section 3.3. To control sampling variability, we generate multiple independent sample graphs, execute the analytic algorithm on each one, and then average (or otherwise aggregate) the results. The practical advantages of this approach can be significant. In an empirical study on a real Twitter dataset (see Section 3.7.2 below), we found that a significant fraction of the top influencers found via the PED approach were either steady but temporarily dormant influencers like Alice or rising stars like Bob; these important sets of influencers would be overlooked under a snapshot or sliding-window approach, respectively.

We have developed an end-to-end system for dynamic graph analysis, called TIDE, that embodies the above ideas. TIDE is implemented on top of the Spark distributed processing system [118], leveraging its native streaming [119] and graph processing [43] support. TIDE allows users to analyze dynamic graphs using existing algorithms for static graphs; moreover, analyses can be specified using existing APIs for batch graph processing systems. Empirical studies on a real Twitter dataset demonstrate the effectiveness and efficiency of the system. TIDE is the first distributed system to systematically support probabilistic edge decay for analyzing dynamic graphs.

The contributions of this chapter are as follows:

- We formalize a general PED model for dynamic graphs that implements temporally biased sampling and subsumes existing models.
- We develop incremental sample-maintenance methods for PED models with exponential decay functions.
- We exploit overlap between sample graphs to store the sample set in a space-efficient manner.
- We provide a bulk graph execution model to efficiently analyze multiple samples of dynamic interaction graphs simultaneously.
- We exploit overlap between the realizations of a sample graph at successive time points to allow efficient incremental graph analysis.
- We show how to efficiently implement the TIDE system using Spark (with some modifications).
- We provide experiments on real-world data to assess our new techniques.

## 3.2 Dynamic Interaction Graphs: Existing Models

In this section we formalize both snapshot and sliding window models for dynamic interaction graphs. Given a time domain  $T$ , a *dynamic interaction graph* (or *dynamic graph* for short) is defined as  $G = (V, E)$ , where  $V$  is a set of vertices and  $E \subseteq V \times V \times T$  is a set of time-stamped edges. The presence of an edge  $e = (u, v, t) \in E$  indicates that vertex  $u$  interacts with vertex  $v$  at time  $t$ . We denote by  $t(e)$  the time stamp associated with edge  $e$ . Note that there can be multiple

edges from  $u$  to  $v$  but with different timestamps. In addition, there may be other attributes associated with the vertices and edges of  $G$ .

In Twitter, for example, Alice *mentions* Bob in a tweet if the tweet includes the string “@Bob”, and this *mention interaction* indicates a certain level of attention paid by Alice to Bob [92]. Such mention interactions in Twitter can be modeled as a dynamic graph. The vertices are Twitter users, and an edge from Alice to Bob with timestamp  $t$  means Alice mentioned Bob in a tweet at time  $t$ . The actual tweet can be modeled as an attribute associated with this edge, and user profiles for Alice and Bob can be captured as vertex attributes. Note that arriving edges sometimes introduce new vertices into a dynamic graph; for simplicity, we consider such vertices to already exist in the dynamic graph, but with no prior adjacent edges.

A *snapshot* of a dynamic graph  $G$  at time  $t$  is defined as  $G_t = (V, E_t)$ , where  $E_t = \{e \mid e \in E \wedge t(e) \leq t\}$ . Similarly, a *window* of  $G$  from time  $t$  to  $t'$  is defined as  $G_{t,t'} = (V, E_{t,t'})$ , where  $E_{t,t'} = \{e \mid e \in E \wedge t \leq t(e) \leq t'\}$ . In the *snapshot model*, an analytic function  $F$  applied to a dynamic graph  $G$  at time  $t$  is actually applied to the snapshot  $G_t$ , with the result  $F(G_t)$ . As time advances to  $t'$ , the result is updated to  $F(G_{t'})$  either by computing it from scratch on  $G_{t'}$  or by incrementally updating the result from  $F(G_t)$  to  $F(G_{t'})$ . In the *sliding-window model*, the function  $F$  is applied to  $G_{t-w,t}$ , where  $w$  is a fixed *window size*, i.e., the analysis only considers interactions that happened within the last  $w$  time units.

Observe that both models embody a binary view of an edge’s role in an analysis; it is either included for analysis or not. An included edge has the same importance as any other edge, regardless of how outdated it is. As mentioned previously, this simplistic view makes it impossible to satisfy both recency and con-

tinuity requirements simultaneously. In contrast, temporally biased sampling of the dynamic graph provides a probabilistic view of an edge’s role: edges from past to present all have chance to be considered (continuity) but outdated edges are less likely to be used (recency) in an analysis, so that the influence of an edge decays over time. In the following section, we describe the probabilistic edge decay (PED) model for temporally biased sampling.

### 3.3 The PED Model

When applying a function to a dynamic graph at time  $t$  under the PED model, an edge  $e$  with a timestamp  $t(e) \leq t$  has an independent probability  $P^f(e)$  of being included in the analysis, where  $P^f(e) = f(t - t(e))$  for a non-increasing *decay function*  $f : \mathfrak{R}_+ \mapsto [0, 1]$ . As time advances,  $e$ ’s age  $t - t(e)$  increases and the inclusion probability  $P^f(e)$  either decreases or remains unchanged. Note that the snapshot model and the sliding-window model are two special cases of the PED model with  $f \equiv 1$  and  $f(a) = I(a \leq w)$  respectively, where  $I(X)$  denotes the indicator of event  $X$ . In general, we can require that  $f$  be positive and strictly decreasing. Then, at any time  $t$ , every edge  $e$  with  $t(e) \leq t$  has a non-zero chance of being included in the analysis (continuity) but an edge becomes increasingly unimportant in the analysis over time, so that newer edges are more likely to participate in the analysis (recency).

Formally, let  $G = (V, E)$  be a dynamic graph and  $f$  a decay function. For  $t \geq 0$ , denote by  $\mathcal{G}_t = \{(V, E') : E' \subseteq E_t\}$  the set of  $2^{|E_t|}$  possible graphs at time  $t$ . (Here  $E_t$  is defined as in Section 3.2 and  $|E_t|$  denotes the number of edges in  $E_t$ ; we suppress the underlying dynamic graph  $G$  in the notation.) Define the

possible-graph distribution  $\mathbb{P}_{f,t}$  over  $\mathcal{G}_t$  by setting

$$\mathbb{P}_{f,t}(G') = \prod_{e \in E'} f(t - t(e)) \prod_{e \in E_t - E'} [1 - f(t - t(e))] \quad (3.1)$$

for  $G' = (V, E') \in \mathcal{G}_t$ . A *sample graph* at time  $t$  (with respect to  $f$ ) is defined as a graph drawn from the distribution  $\mathbb{P}_{f,t}$ . In the *PED model*, an analytic function  $F$  applied to  $G$  at time  $t$  is actually applied to  $N \geq 1$  independent and identically distributed (i.i.d.) sample graphs  $G_t^{f,1}, G_t^{f,2}, \dots, G_t^{f,N}$  to yield i.i.d. results  $F(G_t^{f,1}), F(G_t^{f,2}), \dots, F(G_t^{f,N})$ . These results can be used to control the variability introduced by the sampling process. In the simplest cases, the results can be averaged together. For example, if  $F$  returns the influence score for each person in an interaction graph, then one might want to compute the average per-person influence score at time  $t$ . In general, analysts can decide whether and how they want to aggregate the results into one result; see Section 3.7 for further discussion.

In what follows, we focus on the important class of *exponential* decay functions of the form  $f(a) = p^a$  for some  $0 < p < 1$ . We call  $p$  the *decay probability*. In general, the exponential decay of edges captures most application scenarios and has been widely adopted in practice [94, 117, 122]. Moreover, exponential edge decay guarantees that the space requirement for storing the dynamic graph is bounded with high probability; see Section 3.4.2.

For simplicity, we adopt a discretized time approach that has been widely used in existing work [119, 88]. Specifically, the continuous time domain is partitioned into intervals of length  $\Delta$ , and the dynamic graph is observed only at times  $\{k\Delta : k \in \mathcal{N}\}$ , where  $\mathcal{N} = \{0, 1, 2, \dots\}$ . Moreover, all edges that arrive in an interval  $[k\Delta, (k+1)\Delta)$  are treated as if they arrived at time  $k\Delta$ , i.e., at the start of the interval. Thus we can take  $T = \mathcal{N}$  for the time domain,  $k \in \mathcal{N}$  to represent

the age of an edge, and  $f(k) = p^k$  to represent the exponential decay function. Moreover, updates to a dynamic graph can be viewed as arriving in a stream of batches  $B_0, B_1, B_2, \dots$ , where all incoming edges in batch  $B_i$  have time stamp  $i$ .

### 3.4 Maintaining Sample Graphs

In this section we describe how to efficiently maintain the set of  $N$  sample graphs over time. The key ideas are to incrementally update the sample graphs and to exploit overlaps between the sample graphs at a given time point by storing the graphs in an aggregated form. We first describe our general approach to incremental maintenance of the sample graphs and then describe how these graphs are stored in a space-efficient “aggregate graph”. We then combine these techniques to obtain specific algorithms for eager and lazy updating of the set of sample graphs.

#### 3.4.1 Incremental Updating: General Approach

As time advances from  $t$  to  $t + 1$ , a naive way to update the results is to materialize  $N$  independent sample graphs from scratch and then analyze them. However, generating  $N$  samples from the ever larger snapshot graph is prohibitively expensive. An incremental approach for computing sample graphs rests on the following theorem, the proof of which is straightforward.

**Theorem 1.** *For  $f(k) = p^k$ , let  $G_t^{f,i} = (V, E_t^{f,i})$  be the  $i$ th sample graph at time  $t$ , so that  $G_t^{f,i}$  has probability distribution  $\mathbb{P}_{f,t}$  given in (3.1), and let  $B_{t+1}$  be the batch of incoming edges at time  $t + 1$ . Let  $G' = (V, S_p(E_t^{f,i}) \cup B_{t+1})$ , where  $S_p(E_t^{f,i})$  is a Bernoulli sample*

of  $E_t^{f,i}$  with sampling probability  $p$ . Then  $G'$  has distribution  $\mathbb{P}_{f,t+1}$ , that is,  $G'$  can be viewed as a sample graph at time  $t + 1$ .

This result provides an efficient way of constructing  $G_{t+1}^{f,i}$  from  $G_t^{f,i}$ . Instead of generating  $G_{t+1}^{f,i}$  from scratch, we only need to subsample the edge set of  $G_t^{f,i}$  and combine the subsample with the edges in the arriving batch  $B_{t+1}$ . It follows immediately from the theorem that the  $N$  sample graphs at  $t+1$  generated by this incremental updating scheme are the desired  $N$  independent sample graphs, provided that the  $N$  sample graphs at  $t$  are independent and each subsampling process is executed independently. The instantiations of the  $i$ th sample graph at times  $t$  and  $t + 1$  overlap significantly. Indeed, it is not hard to see that, in expectation,  $G_t^{f,i}$  shares a fraction  $p$  of its edges with  $G_{t+1}^{f,i}$  under incremental updating.

### 3.4.2 The Aggregate Graph

Besides the overlap between instantiations of a sample graph at two consecutive time points, there is also overlap between different sample graphs at the same time point. Suppose, for example, that each update batch is of size  $M$ . Denote by  $S_{p,M,t}$  the number of edges in a sample graph at time  $t$  with decay function  $f(k) = p^k$ , and assume throughout that  $t$  is large. We then have  $E[S_{p,M,t}] \approx M \sum_{k=0}^{\infty} p^k = \frac{M}{1-p}$ . Moreover,  $S_{p,M,t}$  has a Poisson-Binomial distribution, so that, specializing the high-accuracy “refined normal approximation” in [108], we have for large  $t$  and  $j = 0, 1, \dots$  that  $P(S_{p,M,t} \leq j) \approx \Phi(y) + \gamma(1 - y^2)\phi(y)$ , where  $\Phi$  and  $\phi$  are the cumulative distribution function and probability density function of a standard (mean 0, variance 1) normal distribution,  $y = (j + 0.5 - \mu)/\sigma$ ,  $\gamma = (\mu/\sigma^3)(p^3 - p^2 +$

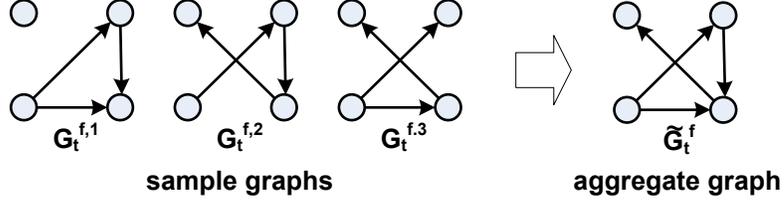


Figure 3.2: Example aggregate graph

$p)/(1 + p - p^3 - p^4)$ ,  $\mu = M/(1 - p)$ , and  $\sigma^2 = Mp/(1 - p^2)$ . For the moderate values of  $p$  and large values of  $M$  encountered in practice—e.g.,  $p = 0.8$  and  $M = 13.9$  million in our experiments—the distribution of  $S_{p,M,t}$  is sharply concentrated around its mean. With the above values of  $p$  and  $M$ , for example,  $S_{p,M,t}$  lies within roughly  $\pm 1\%$  of its mean with a probability exceeding 99.99%. Denoting by  $S'_{p,M,t}$  the number of edges shared by one sample graph with another, we have  $E[S'_{p,M,t}] \approx M \sum_{k=0}^{\infty} p^{2k} = \frac{M}{1-p^2}$ , because an edge with age  $k$  has a probability  $p^k \cdot p^k = p^{2k}$  of appearing in both sample graphs at the same time. Again, there is sharp concentration about the mean, and so the expected fraction of shared edges is  $E[S'_{p,M,t}/S_{p,M,t} \mid S_{p,M,t} > 0] \approx \frac{M}{1-p^2} / \frac{M}{1-p} = \frac{1}{1+p} > \frac{1}{2}$ , and similarly  $S'_{p,M,t}/S_{p,M,t} > 1/2$  with high probability.

Given the significant overlap between different sample graphs at a time point, we see that naively maintaining  $N$  sample graphs  $G_t^{f,1}, G_t^{f,2}, \dots, G_t^{f,N}$  separately incurs much redundancy. Instead, we can store the  $N$  sample graphs as a single *aggregate graph*  $\tilde{G}_t^f = (V, \cup_{i=1}^N E_t^{f,i})$ , where the edge sets of the sample graphs are simply unioned. Figure 3.2 shows an example aggregate graph comprising three sample graphs. The attributes for an edge that appears in multiple sample graphs need only be stored once in the aggregate graph. For each aggregate edge, we keep track of the sample graph(s) to which the edge belongs.

In contrast to the continually increasing memory requirement in the snap-

shot model, the PED model has a bounded memory requirement as new edges are added over time, provided that the update batch at each time stamp is bounded. Denoting by  $M$  the maximum size of an update batch, we see from our earlier analysis that the size of each sample graph is bounded by  $\frac{M}{1-p}$  with very high probability. It follows that even the naive approach of storing  $N$  sample graphs separately has a sharp probabilistic upper bound of  $\frac{MN}{1-p}$  edges.

To analyze the expected space requirement for the aggregate graph, first observe that, under incremental updating, an edge  $e$  that does not appear in the aggregate graph at time  $t$  will not appear in the aggregate graph for  $t' > t$ . As a result, we can establish a memory bound that is significantly smaller than that of the naive approach.

**Theorem 2.** *Let  $M$  be the maximum size of an update batch, and  $f(k) = p^k$  be an exponential decay function. Then the expected number of edges in the aggregate graph of  $N$  sample graphs at any time is bounded by  $M\lceil \log_{\frac{1}{p}}(N) \rceil + \frac{M}{1-p}$ .*

*Proof.* Based on the definition of the exponential decay function, an edge whose age is  $k$  just prior to a given update of a sample graph will be removed from the graph with probability  $1 - p^k$ . Thus the edge has probability  $1 - (1 - p^k)^N$  of appearing in at least one of  $N$  sample graphs after an update. The expected total number of edges in the aggregate graph is therefore bounded by  $\sum_{k=0}^{\infty} M(1 - (1 -$

$p^k)^N$ ). Setting  $K = \lceil \log_p \frac{1}{N} \rceil = \lceil \log_{\frac{1}{p}} N \rceil$ , we have

$$\begin{aligned}
& \sum_{k=0}^{\infty} M(1 - (1 - p^k)^N) \\
&= M \sum_{k=0}^{K-1} 1 - (1 - p^k)^N + M \sum_{k=K}^{\infty} 1 - (1 - p^k)^N \\
&\leq MK + M \sum_{k=K}^{\infty} Np^k = MK + \frac{MNp^K}{1 - p} \\
&\leq M\lceil \log_{\frac{1}{p}}(N) \rceil + \frac{M}{1 - p},
\end{aligned}$$

where  $(1 - p^k)^N \geq 1 - Np^k$  by Bernoulli's inequality.  $\square$

The above theorem provides an upper bound (for all time points) on the expected memory consumption when using the aggregate graph to maintain  $N$  sample graphs. Observe that the expected number of edges that need to be stored is reduced from  $O(MN)$  for the naive approach to  $O(M \log N)$ . For example, when  $p = 0.8$ ,  $N = 96$ , and  $M = 10$  million, the expected storage requirement would be 4.8 billion edges for the naive approach but only about 250 million edges using the aggregate graph. Arguments as before show that, typically, the above expected storage complexities for the two approaches also yield high probability upper bounds. In the aggregate graph, we can use a bit array for each edge  $e$  to indicate the sample graphs in which  $e$  appears; this additional storage is worthwhile because of the savings from not storing redundant edges and their attributes. In fact, as we show in Section 3.4.4 below, we can even avoid storing the bit array for an edge and simply materialize it whenever it is needed.

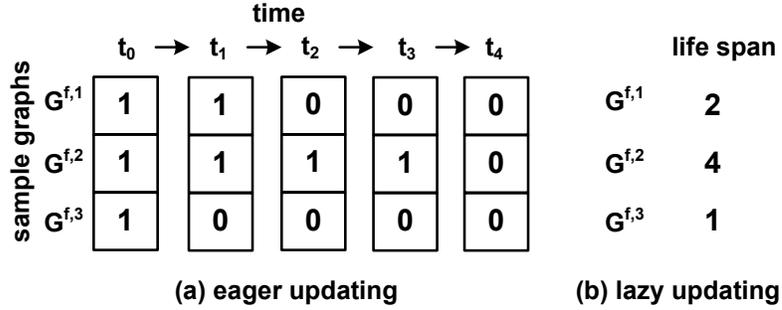


Figure 3.3: Incremental updating for one edge

### 3.4.3 Eager Incremental Updating

We can now combine incremental updating techniques with the aggregate graph to obtain specific algorithms for maintaining a set of  $N$  sample graphs. Our first approach is called the *eager* incremental updating method and is a straightforward implementation of the process described in Theorem 1.

We store the  $N$  sample graphs together in the aggregate graph, and attach a bit array of size  $N$ , denoted as  $\beta$ , to each edge  $e$  in the aggregate graph, to indicate the sample graphs to which this edge belongs. Specifically,  $e.\beta[i] = 1$  means that  $e$  appears in the  $i$ th sample graph and  $e.\beta[i] = 0$  otherwise. As shown in Figure 3.3(a), whenever a new edge  $e$  is first added to the dynamic graph,  $e.\beta[i] = 1$  for all  $i$ , because the edge appears in all sample graphs. As time goes by,  $e$  gradually disappears from some sample graphs. At each batch arrival time, we apply a Bernoulli trial with probability  $p$  on  $e$  for each sample graph where  $e$  still appears. Thus, at each update, we scan through the bit array and, for each bit that equals 1, we set it to 0 with probability  $1 - p$ . Once  $\beta$  contains all 0s, we remove the edge from the aggregate graph.

This eager incremental updating method is simple and straightforward, but

it requires a bit array of size  $N$  for each edge in the aggregate graph. This motivates our second approach, the *lazy* incremental updating method.

### 3.4.4 Lazy Incremental Updating

The lazy incremental updating method avoids materializing the bit arrays based on the observation that the *life span*  $L_e^i$  of edge  $e$  in the  $i$ th sample graph follows a geometric distribution; the life span is the time from when the edge arrives until it is permanently removed from the aggregate graph via a Bernoulli subsampling step. That is,  $P(L_e^i = l) = p^{l-1}(1 - p)$  for  $l \in \{1, 2, \dots\}$ . Note that  $L_e^i \geq 1$  because  $e$  always appears in all of the sample graphs when it first arrives. Figure 3.3(b) shows the life spans in different sample graphs of the example edge in Figure 3.3(a).

Based on this observation, we can simplify the incremental updating process. For an edge  $e$  that has just been added to the  $i$ th sample graph, we directly sample the lifetime  $L_e^i$ . Then, based on the edge's time stamp  $t(e)$  and the life span  $L_e^i$ , we know exactly when it will disappear from the  $i$ th sample graph. Observe, however, that we need to keep track of the life span for each edge in each sample graph. A naive approach would use  $N$  integers per edge, which is an even worse storage requirement than for the  $N$  bits per edge in the eager incremental updating method.

We avoid the storage problem by using a lightweight method to deterministically materialize the  $N$  integers whenever they are needed, while maintaining their mutual statistical independence. Specifically, we exploit a 64-bit version of the MurmurHash3 random hash function [44]. Given the unique combination

of an edge ID and a sample graph ID, MurmurHash3 can deterministically and efficiently generate a 64-bit integer. Moreover, the integers generated for different (edge ID, graph ID) combinations appear random enough to pass the highly rigorous TestU01 [73] test suite for pseudorandom number generators. We use standard techniques to transform the pseudorandom 64-bit integers produced by MurmurHash3 into pseudorandom samples from the geometric distribution; see, for example, [72, p. 469].

### 3.4.5 General Decay Functions

The foregoing discussion can be generalized to decay functions other than the exponential function  $f(k) = p^k$ . Indeed, for an arbitrary decay function given by  $f(k) = \theta_k$  with  $1 = \theta_0 \geq \theta_1 \geq \theta_2 \geq \dots$ , we can use an eager incremental updating scheme as before, but with a Bernoulli sampling rate  $p_k = \theta_k / \theta_{k-1}$  when processing batch  $B_k$  for  $k \geq 1$ . If  $\sum_k \theta_k < \infty$ , then arguments almost identical to those given before show that the number of edges in the aggregate graph is bounded in expectation and with high probability.

## 3.5 Bulk Analysis of Sample Graphs

The previous section discussed how to efficiently maintain a set of  $N$  sample graphs. In this section, we focus on how to efficiently execute analysis algorithms on these graphs. An important design goal of our system is to provide, for dynamic graphs, the same familiar analytics interfaces used in systems for managing static graphs. We therefore adopt the popular vertex-centric iterative

computation model used in static graph processing systems such as Pregel [77], GraphLab [76], Trinity [98] and GRACE [110]. Under this computation model, a user-defined *compute()* function is invoked on each vertex  $v$  to change the state of  $v$  and of  $v$ 's adjacent edges; changes to other vertices are propagated through either message passing (e.g., in Pregel) or scheduling of updates (e.g., in GraphLab). This computation is carried out iteratively until there is no status change for any vertex. Given this computation model, we describe techniques both for bulk execution of analytics and for incremental updating of analytical results as time progresses.

### 3.5.1 Bulk Graph Execution Model

The most straightforward way to analyze  $N$  sample graphs is to materialize each sample graph from the current aggregate graph and apply the analytic function of interest to each individual sample graph. However, this naive approach ignores the significant overlap between the sample graphs, as discussed in Section 3.4.1. The key observation is that similar topologies lead to similar vertex and edge states among the different sample graphs during the iterative computation.

To take advantage of the similarities among sample graphs, we propose a bulk execution model on multiple sample graphs. We first partition the  $N$  sample graphs into one or more *bulk sets* comprising  $s$  ( $\leq N$ ) sample graphs. For each bulk set, we combine the  $s$  sample graphs into a *partial aggregate* graph, and process the partial aggregate graph as a whole instead of processing the  $s$  sample graphs individually. The state of a vertex or an edge in the partial ag-

gregate graph is an array of the states of the corresponding vertex or edge in the  $s$  sample graphs. If an edge does not appear in a sample graph, then the associated array element is *null*.

Computation at a vertex  $v$  in the partial aggregate graph proceeds by looping through the  $s$  sample graphs, reconstructing the set of  $v$ 's adjacent edges in each sample graph and applying the *compute()* function. The resulting updates to other vertices are then grouped by the destination vertex ID and the combined updates are propagated via message passing or scheduling of updates. Consider, for example, a message passing setting, and suppose that the bulk computation on a vertex  $v$  results in two messages to destination vertex  $u$ :  $\langle u, m_1 \rangle$  for the  $i$ th sample graph and  $\langle u, m_2 \rangle$  for the  $j$ th sample graph. Then a combined message  $\langle u, \{(m_1, i), (m_2, j)\} \rangle$  is sent to  $u$ . Algorithm 5 demonstrates the bulk execution model when message passing is used for update propagation. After one bulk set is complete, we proceed to the next bulk set until all of the  $N$  sample graphs are processed.

The benefit of the bulk graph execution model is multifold. First, extracting and loading sample graphs from the full aggregate graph in groups of size  $s$  amortizes the nontrivial overheads of this pre-analysis step. Second, because graph traversal requires many random memory accesses, bulk execution of computations on the same vertex across different sample graphs results in local computations that yield improved caching behavior. Finally, the similar message values in a combined message from one vertex to another create opportunities for compression during communication over the network. Likewise, compression can be applied when persisting similar values in the state array for a vertex or edge on disk for purposes of checkpointing.

---

**Algorithm 3.1:** Bulk Graph Execution Model

---

**input** : A vertex  $v$  in a partial aggregate graph of  $s$  sample graphs, its adjacent edges  $E_v$ , and its incoming messages  $inMsgs$

- 1 initialize  $msgs = \emptyset$ ; // each msg is a tuple (DestVtxID, MsgData, SampleGraphID)
- 2 **for**  $i=1$  **to**  $s$  **do**
- 3     construct a new vertex  $v_i$  where  $v_i.state = v.state[i]$ ;
- 4      $inMsgs_i = inMsgs.getMsgsForGraph(i)$ ;
- 5     initialize  $v_i$ 's adjacent edges  $E_{v_i} = \emptyset$ ;
- 6     **foreach**  $e \in E_v$  **do**
- 7         **if**  $e$  is in the  $i$ th sample graph **then**
- 8             construct  $e_i$  where  $e_i.state = e.state[i]$ ;
- 9              $E_{v_i}.add(e_i)$ ;
- 10         **end**
- 11     **end**
- 12      $orgMsgs = compute(v_i, E_{v_i}, inMsgs_i)$ ; // call the UDF
- 13      $msgs.add(attachGraphID(orgMsgs, i))$ ;
- 14      $v.state[i] = v_i.state$ ;
- 15     **foreach**  $e_i \in E_{v_i}$  **do**
- 16          $e$  is the corresponding edge in  $E_v$ ;
- 17          $e.state[i] = e_i.state$ ;
- 18     **end**
- 19 **end**

**output:** The combined messages grouped by dest vertex id:

$grpMsgs = msgs.groupByDest()$

---

The number  $s$  of sample graphs in a bulk set is a tunable system parameter that trades off space minimization and computational efficiency. A larger value of  $s$  enables shared computation among more sample graphs and hence more benefit from compression of vertex/edge states and combined updates, but also leads to higher memory requirements for each bulk execution. We discuss how to choose  $s$  in Section 3.7.

### 3.5.2 Incremental Graph Analysis

Now that we have an efficient way to analyze the  $N$  sample graphs at time  $t$ , can we exploit the results at  $t$  to more efficiently generate results at  $t + 1$ ? We have shown that the instantiations of a given sample graph at two consecutive time points share a large number of common edges, which leads to similar vertex and edge states during the graph computation. For iterative graph algorithms such as Katz centrality [64] and PageRank [23] computation, this provides an opportunity to use the ending vertex and edge states at time  $t$  as the starting states for the iterative computation at time  $t + 1$ . These improved starting states can lead to faster convergence. One caveat is that some algorithms do not work correctly under this incremental scheme [35], so that recomputation from scratch is required. Existing dynamic graph processing systems [27, 78, 36] encounter the same issue.

## 3.6 Implementation on Spark

In this section, we provide a brief overview of Spark, and then describe several important Spark-specific optimizations that we incorporated when implementing TIDE.

### 3.6.1 Spark Overview

Spark is a general-purpose distributed processing framework based on a functional programming paradigm. Spark provides a distributed memory abstraction called a Resilient Distributed Dataset (RDD) to support fault-tolerant computation across a cluster of machines. RDDs can either reside in the aggregate main memory of the cluster, or in efficiently serialized disk blocks. An RDD is immutable and cannot be modified, but a new RDD can be constructed by transforming an existing RDD. Spark utilizes both lineage tracking and checkpointing for fault tolerance.

### 3.6.2 Implementation and Optimization

Figure 3.4 demonstrates the end-to-end data pipeline of the TIDE system as implemented on Spark. First, TIDE leverages Spark Streaming to ingest batches of arriving edges, thereby supporting input sources such as HDFS, Kafka, Flume, and so on. The new edges are fed into the incremental updating component that maintains the sample graphs, the result of which is a compact in-memory RDD representing the aggregate graph of  $N$  samples. TIDE then extracts  $s$  samples

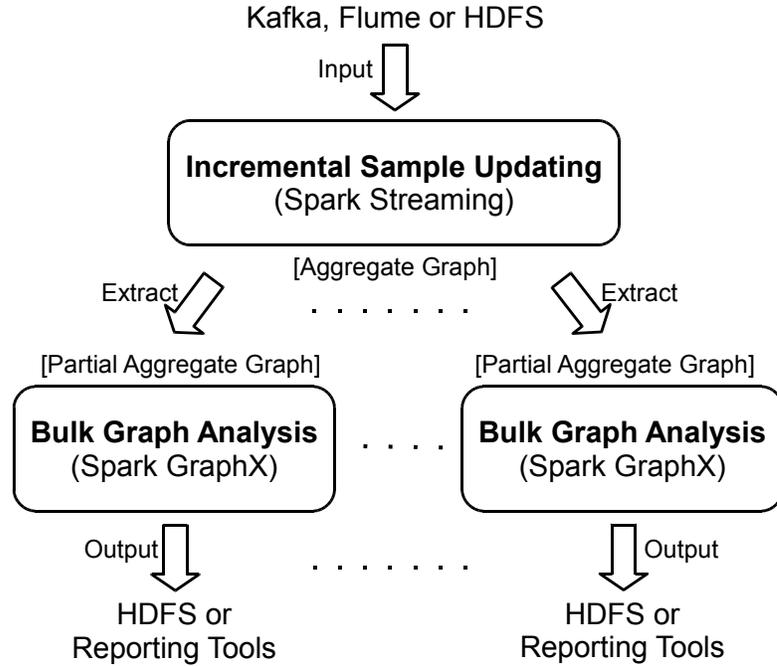


Figure 3.4: Overall Data Pipeline of TIDE System

(where  $s$  is the size of a bulk set) and transforms them into a partial aggregate graph in the Spark GraphX distributed graph RDD representation. The iterative bulk graph analysis algorithm is then executed on this representation of the partial aggregate graph. TIDE repeats the above process for the successive bulk sets of  $s$  sample graphs until all of the  $N$  sample graphs are analyzed. The result of each bulk graph analysis is an RDD that can be stored on HDFS or fed into various reporting tools.

Inside the incremental sample updating component, each batch of new edges is stored in an RDD  $b_t$ , where  $t$  is the time stamp, and the current aggregate graph is stored in an RDD  $g_t$ . Initially, the aggregate graph is just the first batch of edges, i.e.  $g_0 = b_0$ . At  $t = 1$ , a new RDD  $g'_0$  is created from  $g_0$  by applying a set of transformations that implement the one-step edge decay process (i.e., the Bernoulli subsampling step). Next,  $g'_0$  is unioned with  $b_1$  to

produce the updated RDD  $g_1$ . This process continues as time advances. For eager incremental updating, the decay transformations include a map operation to update the bit array of each edge and a filter operation to discard edges that have become nonexistent in all sample graphs. For lazy incremental updating, the decay transformation comprises only a filter operation to check whether an edge has become nonexistent.

The bulk graph analysis component of TIDE is built on top of GraphX [43] which is an implementation of the Pregel and GraphLab processing frameworks on top of Spark. In particular, we implement a bulk execution wrapper for GraphX that performs the vertex-centric computation on the partial aggregate graph as in Algorithm 5.

Finally, we use the lineage and checkpointing mechanisms in Spark to support fault tolerance in TIDE. In what follows, we highlight some implementation optimizations that are specific to Spark.

### **In-Place Update**

Because of their efficiency, the TIDE implementation uses memory-resident RDDs extensively. Memory management is a challenge, however. Because RDDs are immutable, TIDE must continuously create new RDDs as new edges arrive; indiscriminate creation of a large number of objects can quickly saturate memory. This is especially problematic for eager incremental updating, because of its higher memory requirement for storing the aggregate graph. Therefore, TIDE avoids creating new objects by applying in-place updates whenever possible. That is, new RDDs are still created, but they refer to existing objects in

old RDDs. To keep the lineage of RDDs intact, TIDE must also notify Spark that the old in-memory RDDs have been changed. Thus, if an old RDD needs to be reprocessed (e.g., in case of a failure), it must first be regenerated from the latest checkpoints rather than being read directly from memory. In Section 3.7.1, we explore the effectiveness of in-place update for eager and lazy incremental updating methods.

### **Location-Aware Balancing Coalesce**

In Spark, an RDD is divided into a set of partitions that are distributed among the workers in a cluster; each worker can have multiple partitions. Spark tracks the lineage of each partition, i.e., its parent partitions and the operations required to obtain the partition from its parents. For map and filter operations, the resulting RDD has exactly the same number of partitions as the parent RDD, even though some of the partitions can become empty after the filter operation. The union of two RDDs having  $k_1$  and  $k_2$  partitions is an RDD having  $k_1 + k_2$  partitions; the partitions are simply unioned together.

In the incremental updating process, we need to repeatedly thin the aggregate graph through filter operations (and also map operations, in the case of eager updating) and union it with arriving edges. This procedure creates a potential problem. If the RDD for each batch contains  $k$  partitions, then, after ingesting  $n$  batches, the aggregate graph RDD would comprise  $nk$  partitions with highly skewed sizes. Indeed, a partition with older edges is likely to be quite small, or even empty. Since the partition serves as the basic scheduling unit in Spark, the presence of many small and empty partitions incurs a lot of unnecessary scheduling overhead.

Spark provides a *coalesce* operation to reduce the number of partitions in an RDD. If *shuffling*-based coalesce is used, then data in the RDD are reshuffled to generate fewer balanced partitions; otherwise, local partitions are simply merged together. Neither approach is directly applicable to our setting. Because coalesce needs to be applied frequently, shuffling is too expensive. On the other hand, arbitrary merging of local partitions yields highly imbalanced partition sizes. To avoid shuffling data while generating balanced partitions, we extend Spark with a *location-aware balancing coalesce* operation. This new coalesce operation combines local partitions (and thus avoids shuffling), but carefully chooses the candidate partitions based on their sizes by applying the Longest Processing Time (LPT) heuristic [45].

### **Distributed Monte Carlo Simulation**

The eager incremental updating approach requires independent Bernoulli trials on each edge in each sample graph. To ensure that there is no correlation between the pseudorandom numbers generated for different Spark workers, we use the technique discussed in [52] for generating multiple streams of uniform numbers that are provably disjoint. In addition, we track the starting seed for each Spark partition, so that an updating operation on a given partition always produces exactly the same result if executed again (e.g., during failure recovery).

### 3.7 Experimental Evaluation

In this section, we first describe some experiments designed to test the performance of our techniques for maintaining a set of sample graphs. We then evaluate the quality and performance of the PED approach when the sample graphs are used for influence analysis and community analysis.

**Cluster Setup.** All experiments were conducted on a cluster of 17 IBM System x iDataPlex dx340 servers. Each has two quad-core Intel Xeon E5540 2.8GHz processors and 32GB RAM; servers are interconnected using a 1Gbit Ethernet. Each server runs Ubuntu Linux and Java 1.6. One server is dedicated to run the Spark coordinator and each of the remaining 16 servers is configured to run a Spark worker. We set `SPARK_WORKER_CORES=8`, `SPARK_WORKER_MEMORY=28G`, and default values for the other Spark parameters, based on standard practice.

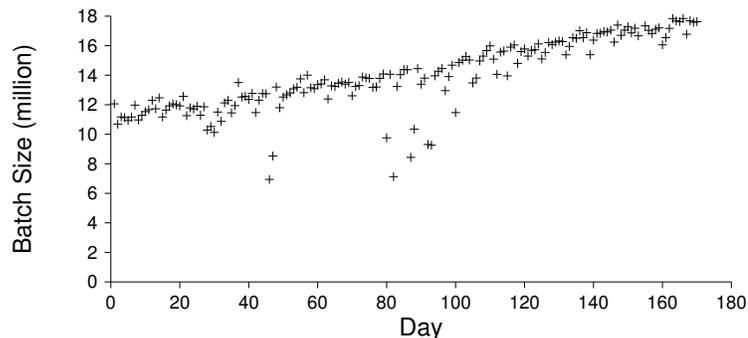


Figure 3.5: Per day batch size of the Twitter dataset

**Dataset.** We used a real Twitter dataset for our experiments. It was obtained via the GNIP service and comprises 10% of the tweets generated between Sep 9, 2011 and Feb 29, 2012. We extracted the mention interactions out of this Twitter dataset and formed dynamic graphs. Figure 3.5 shows the per-day batch

size (number of new edges) of this dataset. On average, 13.9 million new interactions were added per day. We experimented on daily batches, 2-day batches and 3-day batches in our empirical studies. The reason for such a coarse-grained discretization is to ensure that the data is of a large enough scale to test the system, since our dataset is only a small sample of the Twitter stream. In real settings, interactions are generated much more frequently, and thus a fine-grained discretization such as hourly batches would be adopted. In our experiments, the largest running aggregate graph contains around 65 million vertices and 1 billion edges.

**Parameters.** There are three important parameters that need to be specified in TIDE: the decay rate  $p$ , the total number of sample graphs  $N$  to incrementally maintain, and the number of sample graphs  $s$  in each bulk set for graph execution.

The decay rate  $p$  is completely application specific, and controls the proportion of historical interactions that an application considers in the analysis. For example, by setting  $p = 0.8$ , around 0.1% of the interactions from 30 periods ago are included in the current analysis. As another example, suppose that we want to ensure that, with probability  $q = 0.01$ , an influencer who had  $n = 1000$  interactions  $k = 60$  periods ago is still represented in the current network, where “represented” means having at least one adjacent edge remaining. Then we would set  $p = [1 - (1 - q)^{1/n}]^{1/k} \approx 0.825$ .

The number  $N$  of sample graphs controls the precision of the results. A variety of statistical methodologies are available for determining a good value of  $N$ . A comprehensive discussion of this topic is beyond the scope of this chapter, so we content ourselves with a few examples. In the simplest setting, the

goal of the analysis is to compute an expected value  $\mu$  of an analytic graph function  $F$  with respect to the possible-graph distribution  $\mathbb{P}_{f,t}$  defined in (3.1). That is,  $\mu = \sum_{G' \in \mathcal{G}_t} F(G') \mathbb{P}_{f,t}(G')$  or, equivalently,  $\mu = E[F(G')]$ , where  $G'$  is a sample graph at time  $t$ . As an example,  $F$  might return the average influence score of the top 100 influencers. Given an initial value of  $N$ , we compute i.i.d. result samples  $X_1, X_2, \dots, X_N$ , where  $X_i = F(G_i^{f,i})$  and  $G_i^{f,i}$  is the  $i$ th sample graph. Then  $\hat{\mu}_N = N^{-1} \sum_{i=1}^N X_i$  is an unbiased and strongly consistent estimator of  $\mu$ . Assuming that  $N$  is sufficiently large (say,  $N \geq 20$ ), one can compute a standard  $100(1 - \delta)\%$  approximate confidence interval as  $\hat{\mu} \pm z_\delta s_N / \sqrt{N}$ , where  $z_\delta$  is the  $(1 - 0.5\delta)$ -quantile of the standard normal distribution and  $s_N$  is the sample standard deviation of  $X_1, X_2, \dots, X_N$ . If the confidence interval is too wide and the desired accuracy is  $\pm 100\varepsilon\%$ , then, going forward,  $N$  can be increased to  $N^* = z_\delta^2 s_N^2 / (\varepsilon \hat{\mu}_N)^2$  to try and achieve the desired accuracy. In general, we can monitor the confidence interval of the results as time progresses and increase  $N$  on the fly when the estimated accuracy falls below a threshold.<sup>1</sup> If  $F$  takes values in  $\mathfrak{R}^d$  for some  $d > 1$ , then the above methodology can be applied, but using, e.g., an appropriate hyper-rectangular confidence region of specified maximum edge length on the  $d$  quantities of interest [79]. In more complex situations where, e.g.,  $F$  returns a list of top- $k$  influencers or an iceberg-query result of all persons with influence score above a threshold, simple averaging of the results from the different sample graphs may not suffice—see, e.g., [56]. The procedures for aggregating the results might then become complex, so that simple formulas for estimating error may not be available. In this case, bootstrapping techniques or other methods for assessing uncertainty may be needed; see [5] for a recent discussion.

As discussed in Section 3.5.1, increasing the bulk set size  $s$  helps compression

---

<sup>1</sup>If  $N$  needs to be increased in TIDE, we have to compute the set of sample graphs from scratch. However, this happens infrequently.

but also leads to higher memory requirements. Suppose that we have an upper bound on the expected amount of the memory occupied by a partial aggregate graph of  $s$  samples plus the memory consumed by the messages passed while processing this graph. A safe choice is to set  $s$  as large as possible such that the upper bound does not exceed the aggregate worker memory size. We have derived such a bound for our TIDE prototype, but omit the details because the bound is highly specific to our particular Spark implementation.

For the experiments in this section, we found that  $p = 0.8$  is a reasonable decay rate for our example graph applications. Using the processes for choosing  $N$  and  $s$  as described above, we found that  $N = 96$  provides accurate enough results for all experiments and  $s = 16$  is a safe choice for the 3-day batch dataset and graph algorithms used in our experiments. However, in order to demonstrate the effect of the three parameters on the performance of TIDE, we also experiment with different settings of  $p$ ,  $N$  and  $s$ .

In our experiments, we load the streaming input data as a sequence of HDFS files and produce an output sequence of HDFS files that represent the final analytic results at successive time points. We focus on evaluating the performance of the incremental updating and bulk graph analysis components of the TIDE system pipeline shown in Figure 3.4, because the time of the remaining operations (reading input, extracting partial aggregate graphs, and outputting analysis results) is negligible by comparison. Indeed, for the iterative graph algorithms we consistently observed that these remaining operations comprised less than 1.5% of the total execution time.

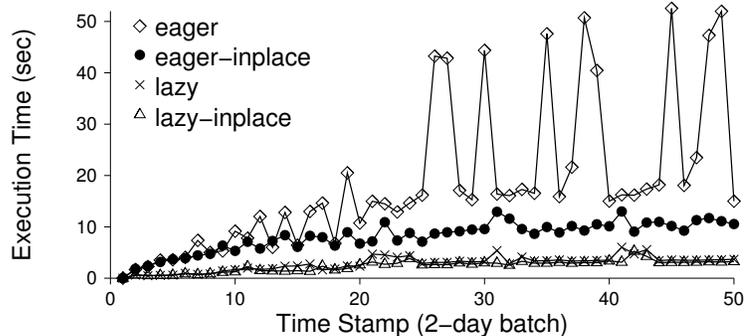


Figure 3.6: Per-batch time for incremental updating

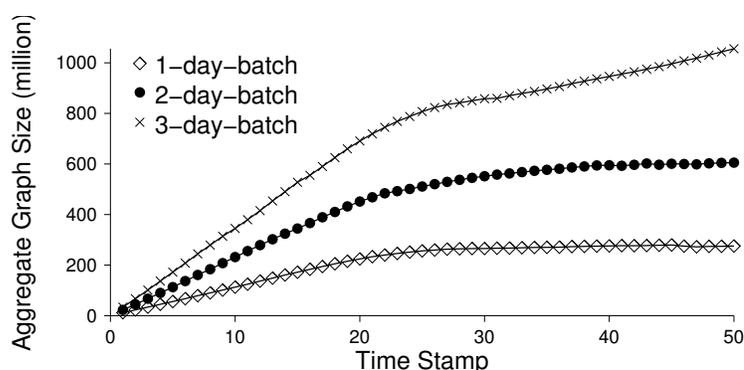


Figure 3.7: Running aggregate graph size

### 3.7.1 Incremental Updating Methods

In this section we empirically study the performance of incremental updating methods. The coalesce and checkpointing operations were carried out for every 10 batches. For the daily batch update, on average every checkpointing takes 47sec for eager updating but only 24sec for lazy updating, because the lazy updating method stores less data. We focus on per-batch comparisons between eager and lazy updating by excluding the times required for checkpointing and coalescing from the execution times reported below.

**Comparison of Updating Methods.** Figure 3.6 depicts the per-batch execution

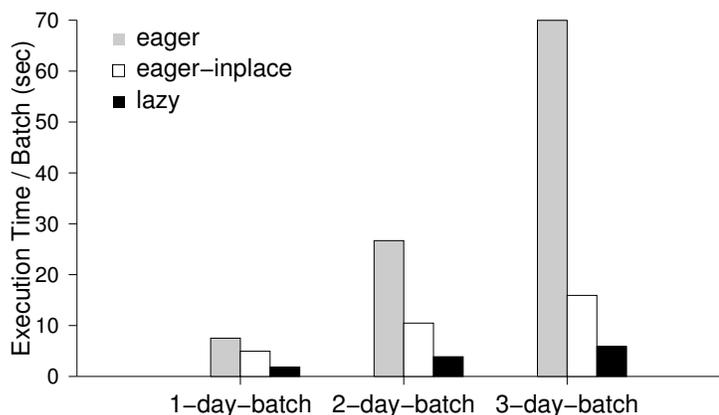


Figure 3.8: Avg per-batch time for incremental updating (after 30th batch)

times for eager and lazy updating, both with and without in-place update, for the first 50 time stamps (batches), using the 2-day batch data as a representative. As shown, the in-place update has a huge effect on the eager updating method. This is because eager updating has a high memory requirement for storing the per-edge bit arrays. Without in-place updates, new bit arrays are continually created as decay transformations are applied. Indeed, a given edge can be re-created multiple times. When memory becomes saturated, garbage collection is invoked to reclaim obsolete objects, causing spikes on the curve for eager updating without in-place updates. In comparison, the lazy updating method benefits little from in-place update because it does not materialize bit arrays; we therefore omit the numbers for lazy updating with in-place updates in the remaining experiments. The running time for the naive sampling method is not reported in Figure 3.6 because it is extremely slow—it has to read and iterate over all the data at each batch arrival. For, e.g., the 50th batch, merely loading the data takes about 78 seconds, and extracting a single sample graph takes about 12 seconds, which means roughly 20 minutes are required for the naive method to obtain all of the sample graphs for this single batch.

It can be seen from Figure 3.6 that, for all four incremental updating methods, execution times initially grow as new edges are added, but gradually stabilize. This is because the aggregate graph size initially increases quickly, but the rate of increase tapers off by around the 30th batch, as depicted in Figure 3.7. We observe the same phenomenon for 1-day batches. These observations reflect the probabilistic upper bounds discussed in Section 3.4.2. The size of the aggregate graph for 3-day batches is still increasing after 30 batches, because the number of edges in later batches are considerably larger than in earlier batches.

Figure 3.8 displays, for various batch sizes, the average execution times per batch after the first 30 batches (i.e. after the execution times stabilize). In-place update shows increasing benefit—from 1.5x to 4.4x speedup—for eager updating as the batch size increases. Lazy updating exhibits a consistent speedup of approximately 2.7x over the in-place eager approach for the various batch sizes.

We also study the effects on system performance of the decay factor  $p$  and the number of samples  $N$ . We only show results for the lazy updating method on 2-day batches, as experiments on other batch sizes exhibit the same trends. Table 3.1 displays the average execution time per batch under several different parameter settings. The running times increase in accordance with the number of edges in the aggregate graph, but the increase is not necessarily proportional to the number of edges. This is because the incremental updating methods run very fast, so that Spark’s job launching and task scheduling times become non-negligible.

**Location-Aware Balancing Coalesce.** We also study the impact of the location-aware balancing coalesce operation described in Section 3.6.2 relative to the two existing shuffle-based and non-shuffle coalesce operations in Spark. We define

Table 3.1: Per-batch time for lazy updating (after 30th batch)

parameter	avg time	# edges in 50 <sup>th</sup> batch
$p = 0.5, N = 96$	2.04 sec	201 million
$p = 0.8, N = 96$	3.87 sec	605 million
$p = 0.8, N = 192$	5.47 sec	683 million

Table 3.2: Coalesce operations for lazy incremental updating

	shuffle-based	non-shuffle	location-aware
skewness	1.01	8.64	1.08
time (sec)	120.62	0.84	1.84

the *skewness* of partitions as the ratio of the maximum partition size divided by the minimum partition size. The skewness is 1 for balanced partitions.

Table 3.2 compares the three coalesce operations when performed at the 40th time stamp of the lazy updating method using the 2-day batch dataset. Shuffle-based coalesce generates balanced partitions, but requires orders of magnitude more running time than the other methods. Non-shuffle coalesce is fast, but produces unacceptably skewed partition sizes. Our location-aware balancing coalesce produces good balanced partitions reasonably quickly. For eager updating, the skewnesses of the three coalesce operations are similar to those for lazy updating. The execution times for non-shuffle and location-aware balancing shuffle stay the same, since these two algorithms merely combine local partitions without touching the data underneath. However, the shuffle-based coalesce takes more time (350 sec) for eager updating.

### 3.7.2 Dynamic Graph Analysis

We choose three representative graph algorithms to demonstrate how our PED model can be used in two example graph applications. We then discuss the performance impact of the bulk graph execution technique. All experiments in this section were conducted on the 3-day batch datasets. To avoid dealing with the initial transient phase where graph size increases dramatically, we report qualitative results for the 40th batch and performance results from the 40th batch onward. The aggregate graph contains around 65 million vertices and 1 billion edges from the 40th batch onward.

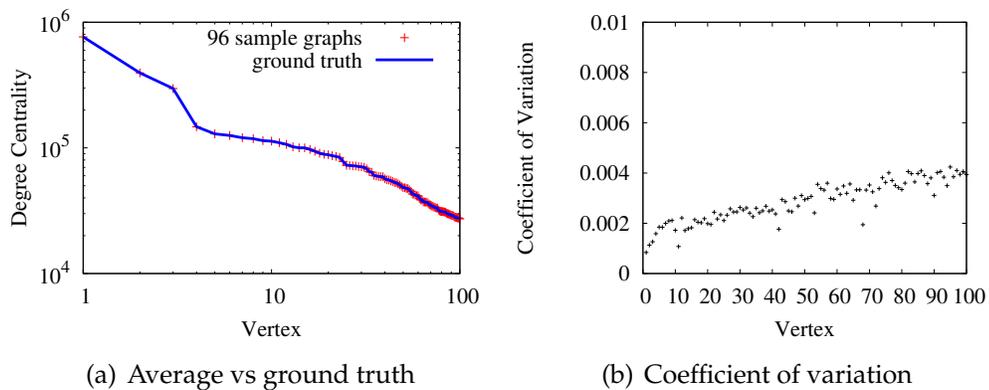


Figure 3.9: Quality of results for degree centrality

#### Influence Analysis

Influence analysis is one of the most important types of analysis for social graphs. Centrality measures of vertices are widely used in practice for this application [22]. We chose the following two representative centrality measures:

**Degree centrality.** Degree centrality is the simplest way to measure the relative importance of a vertex in a graph. The degree-centrality score of a vertex  $v$  is

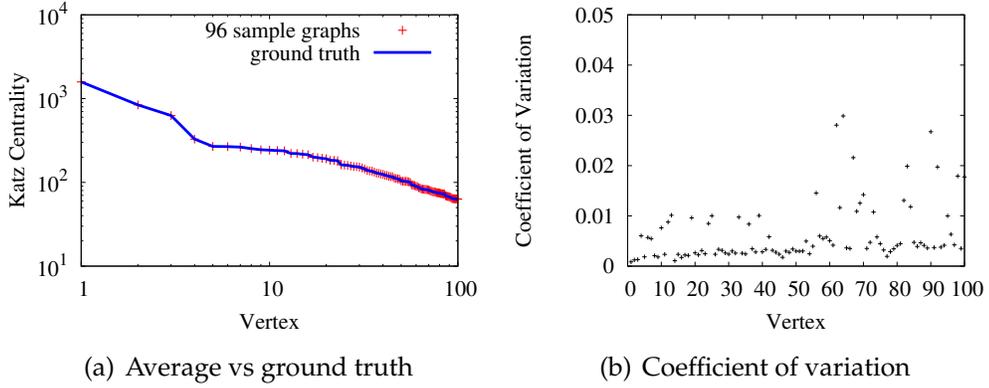


Figure 3.10: Quality of results for Katz centrality

defined as the number of edges incident to  $v$ :  $C_{\text{deg}}(v) = \sum_u |E_{(u,v)}|$ , where  $E_{(u,v)}$  is the set of edges from  $u$  to  $v$ .

**Katz centrality.** Katz centrality is a more complex measure of the importance of a vertex. The Katz centrality of a vertex  $v$  measures the number of paths that end at  $v$ , penalized by the path length:  $C_{\text{Katz}}(v) = \sum_u \sum_{x \in \text{Path}_{(u,v)}} \alpha^{l(x)}$ , where  $\text{Path}_{(u,v)}$  is the set of paths connecting  $u$  to  $v$ ,  $l(x)$  is the length of path  $x$ , and  $\alpha$  is an attenuation factor. In our experiments, we set  $\alpha = 0.002$ .

We consider analytic functions  $F$  that return the centrality score for each vertex in a graph, and our goal is to estimate  $\mu$ , the expected value of  $F$  with respect to the possible-graph distribution  $\mathbb{P}_{f,t}$ . As discussed previously, we estimate  $\mu$  unbiasedly by  $\hat{\mu}_N = N^{-1} \sum_{i=1}^N F(G_t^{f,i})$ ; i.e., for each vertex, we compute the average centrality score over the  $N$  sample graphs at time  $t$ .

To evaluate the quality of the results, we compare the estimated influence scores in  $\hat{\mu}_N$  to the ground-truth influence scores in  $\mu$ . The ground-truth vector  $\mu$  can be computed exactly by incorporating the decay probabilities in the centrality calculation. For degree centrality, the expected number of incident

edges is computed as  $C_{\text{deg}}(v) = \sum_u \sum_{e \in E_{(u,v)}} P^f(e)$ , where  $P^f(e) = f(t - t(e))$  is the decay probability of  $e$ . For Katz centrality, the expected number of penalized paths connected to a vertex is computed as  $C_{\text{Katz}}(v) = \sum_u \sum_{x \in \text{Path}_{(u,v)}} \alpha^{l(x)} P(x)$ . Here  $P(x)$  is the probability of a path  $x$ , which can be calculated as the product of the probabilities of the edges in  $x$ . Note that computing  $\mu$  for either algorithm is prohibitively expensive for real-world applications, because it requires computation over all edges from the past. The size of the dynamic graph quickly grows beyond the capacity of any graph processing system.

Figures 3.9(a) and 3.10(a) compare, for each of the top 100 vertices (the 100 vertices with the highest true expected centrality scores), the average (over the 96 sample graphs) degree-centrality and Katz-centrality scores to the ground truth expected scores. As can be seen, the differences are almost indistinguishable.

Figures 3.9(b) and 3.10(b) display the coefficient of variation, over the 96 sample graphs, of centrality scores for the top 100 vertices. For degree centrality, all vertices have variations less than 0.5%, and for Katz centrality, although the variations are slightly higher, but they are all less than 3%. Clearly, the multiple samples yield good estimates of expected degree and Katz centralities. All of the above results show that choosing  $N = 96$  achieves sufficient accuracy for both algorithms in our setting.

**PED vs Snapshot and Sliding-Window Models.** To demonstrate the potential practical benefits of the PED approach, we empirically compare the set of influencers (as measured by degree centrality) found from our real Twitter dataset when using a PED, snapshot, and sliding-window model (with a window size of three days). Among the top 100 influencers found by the PED model, about

24% of them were, like Alice in Example 1 of Section 3.1, temporarily dormant influencers missed by the sliding-window model and 25% were, like Bob, rising star influencers missed by the snapshot model. Similarly, of the top 1000 influencers found by PED, 17% were like Alice and 26% were like Bob. In summary, a significant portion of potentially important influencers would be totally missed using either the snapshot model or the sliding-window model instead of PED.

### **Connectivity and Community Analysis**

Connectivity and community analysis explores the community structure in social graphs. Existing studies [116] have shown that a social network usually contains a *giant* connected component that consists a constant fraction of the entire graph's vertices. In this example application, we study the characteristics of this giant component under the PED model.

We ran the connected-component algorithm on each sample graph to identify the giant component. The average size of the 96 giant components is 32.3 million vertices with a small standard deviation of only 2207. We observed that about 19 million vertices belong to the giant components of all sample graphs and form the high-probability "backbone" of the giant component. On the other hand, there are about 11.1 million vertices that appear in less than 10% of the sample graphs. Such vertices are connected to the network via edges that are infrequent and/or old. Our PED model can help us understand these two different types of vertices.

## Performance of Bulk Graph Execution

In this subsection, we evaluate the performance of the bulk graph execution technique when analyzing degree centrality, Katz centrality, and connected-component structure. The three analysis algorithms span a range of graph analysis complexities. Determination of degree centrality does not require any iterative computation. The computation of Katz centrality is iterative, similar to that of PageRank. Moreover, it can use the incremental graph analysis scheme discussed in Section 3.5.2 to incrementally update the centrality scores from time  $t$  to time  $t+1$ . Connected-component computation is iterative but cannot leverage the incremental graph analysis scheme. This is because the label-propagation-based algorithm [104] cannot correctly handle incremental deletions of edges, so that re-computation from scratch is necessary at each time point.

In this experiment, we measure average bulk graph processing results from the 40th update onward, i.e., after stabilization. We use LZ4 compression for shuffling in Spark. Empirically, we observed that the use of LZ4 reduced run times by up to 46% for the bulk graph execution model and up to 12% for the naive execution model (processing one sample graph at a time).

Figure 3.11 compares the bulk graph execution model to the naive approach for the above three algorithms and for various values of the bulk-set size  $s$ . At any time point during the iterative Katz-centrality or connected-component algorithms, convergence occurs at roughly the same speed for all  $N$  sample graphs, due to their similar topologies and computation states. When processing the 40th batch, for example, the connected-component algorithm converges in 13 to 15 iterations for most sample graphs. Because the Katz-centrality computation takes roughly the same amount of time for each iteration, we report

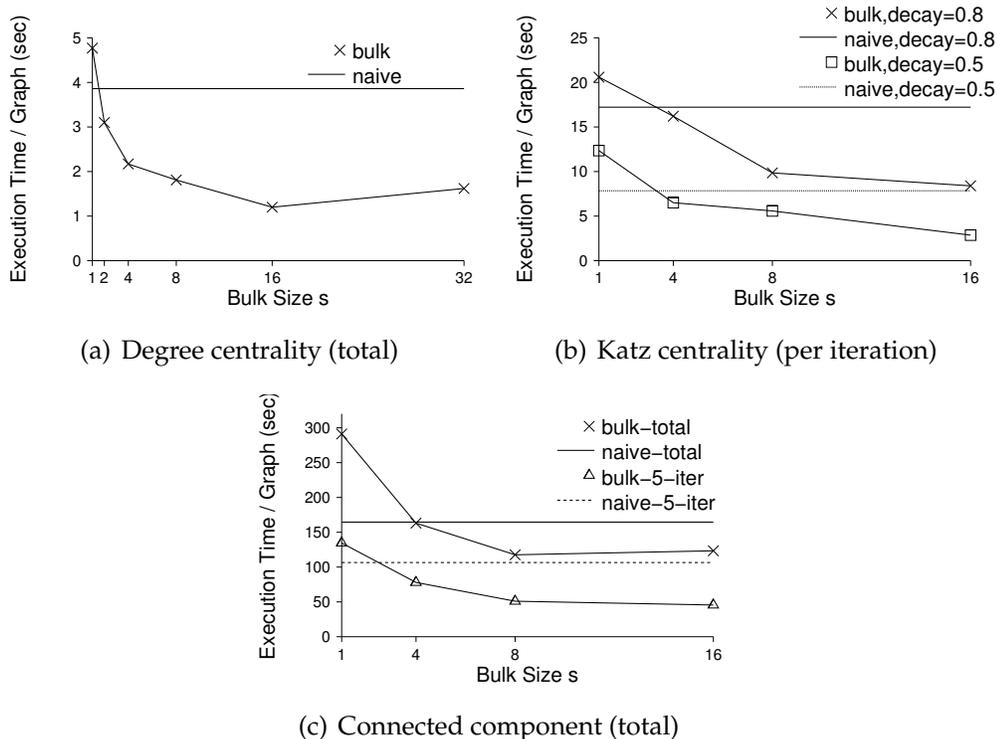


Figure 3.11: Bulk graph execution vs the naive approach

the per-iteration execution time, whereas we report the total execution time for the other two algorithms.

The bulk graph execution model essentially degenerates to the naive approach when  $s = 1$ , but the execution times are all slower than that of the naive approach, due to the overhead of the bulk-execution wrapper. As  $s$  increases, this overhead is quickly amortized and the per-sample-graph performance gradually surpasses that of the naive approach. However, at some point, the advantage starts to decrease due to the higher memory burden of storing a larger partial aggregate graph. Figure 3.11(b) also shows the running time for Katz centrality for different decay factors  $p$ . The running time under  $p = 0.5$  is significantly less than  $p = 0.8$  because the aggregation graph contains only

about one third of the edges. Still, bulk execution significantly reduces the average running time per sample graph in both cases.

Bulk graph execution benefits the simple degree centrality algorithm much more than the two iterative algorithms, because a non-trivial overhead must be paid per iteration in GraphX. Consider, for example, the execution times for the first five iterations of the connected-component algorithm; see Figure 3.11(c). It is known [104] that most of the computation in this algorithm occurs in the first few iterations (five iterations in our case). Even though there is very little to do in remaining iterations, we still must pay the per-iteration overhead in GraphX. As can be seen, the time for each remaining iteration is more or less the same for different bulk-set sizes.

As discussed before, the iterative Katz-centrality algorithm is able to leverage the incremental graph analysis scheme by using the end states at a time point as the starting states for the next time point. Empirically, we observed substantial performance improvements when using this incremental scheme. As an example, for a randomly chosen sample graph at the 40th batch, the Katz-centrality algorithm requires 28 iterations to converge if computing from scratch. In contrast, by reusing the result of the 39th batch, only four iterations are needed.

### **3.8 Related Work**

In recent years, a number of distributed graph processing systems [77, 111, 42, 98, 110, 114, 104] have been proposed for static graphs. For distributed processing of dynamic graphs, existing systems include Kineograph [27], as well as

environments designed for incremental iterative data flows, such as Naiad [78] and the system described in [36]. All three of these systems, however, are based on the snapshot model.

Several recent works have investigated, from a graph-database perspective, the problems of storing and retrieving large-scale evolving graphs [80, 91, 65], but they do not consider complex graph analytics such as influence-analysis and community-detection algorithms. More specifically, a hybrid adaptive replication method was proposed in [80] to manage dynamic graphs in distributed memory, which handles graph updates efficiently. A distributed hierarchical index was proposed in [65] to efficiently retrieve historical snapshots of a graph at specified time points. It utilizes indexing to exploit commonalities among multiple graphs in order to avoid redundant computation of certain simple vertex properties, such as degree counting. It is unclear whether this technique can be generalized to compute arbitrarily complex vertex properties, such as Katz centrality, and whether it will provide an overall performance benefit when analyzing dynamically changing graphs (since any indexes must be incrementally maintained). Chenghui et al studied efficient query processing over a sequence of snapshot graphs by precomputing the results on a small set of representative graphs [91].

In [48], a modified definition of Katz centrality is proposed to capture both time-dependency and recency of random walks in a dynamic graph. In comparison, TIDE is not designed for a specific graph algorithm, but rather is a general platform that allows direct application of a wide range of static graph algorithms to dynamic graphs.

The general idea of temporally biased sampling in (non-graph) data streams

was introduced in [6] to reduce staleness in the sample in order to obtain analytic results more relevant to the present. Data-decay methods were studied for data streams [28], but the focus was on relatively simple aggregation queries. The use of probabilistic edge decay for analysis of dynamic graphs has not been formally studied before.

## CHAPTER 4

# INTERACTIVE EDGE-WEIGHTED PERSONALIZED PAGERANK VIA MODEL REDUCTION

Personalized PageRank is a standard tool for finding vertices in a graph that are most relevant to a query or user. To personalize PageRank, one adjusts node weights or edge weights that determine teleport probabilities and transition probabilities in a random surfer model. There are many fast methods to approximate PageRank when the node weights are personalized; however, personalization based on edge weights has been an open problem since the dawn of personalized PageRank over a decade ago. In this chapter, we describe the first fast algorithm for computing PageRank on general graphs when the edge weights are personalized. Our method, which is based on model reduction, outperforms existing methods by nearly *five orders of magnitude*. This huge performance gain over previous work allows us — for the very first time — to solve learning-to-rank problems for edge weight personalization *at interactive speeds*, a goal that had not previously been achievable for this class of problems.

### 4.1 Introduction

PageRank was first proposed to rank web pages [23], but the method is now used in a wide variety of applications such as object databases, social networks, and recommendation systems [14, 12, 37, 39]. In the PageRank model, a random walker moves through the nodes in a graph, at each step moving to a new node by transitioning along an edge (with probability  $\alpha$ ) or by “teleporting” to a position independent of the previous location (with probability  $(1 - \alpha)$ ). That is, the vector  $x^{(t)}$  representing the distribution of walker locations at time  $t$  satisfies

the discrete-time Markov process

$$x^{(t+1)} = \alpha P x^{(t)} + (1 - \alpha)v \quad (4.1)$$

where the column-stochastic matrix  $P$  represents edge transition probabilities and the vector  $v$  represents teleportation probabilities. The PageRank vector  $x$  is the stationary vector for this process, i.e. the solution to

$$Mx = b, \text{ where } M = (I - \alpha P), \quad b = (1 - \alpha)v. \quad (4.2)$$

The entries of  $x$  represent the importance of nodes.

Standard PageRank uses just graph topology, but many graphs come with weights on either nodes or edges. For example, we may weight nodes in a web graph by how relevant we think they are to a topic [53], or we might assign different weights to different types of relationships between objects in a database [14]. Weighted PageRank methods use node weights to bias the teleport vector  $v$  [58, 53, 13] or edge weights to bias the transition matrix  $P$  [115, 59, 14, 112]. Through these edge and node weights, PageRank can be *personalized* to particular users or queries. Concretely, in *node-weighted personalized PageRank*, we solve

$$Mx(w) = (1 - \alpha)v(w), \quad w \in \mathbb{R}^d$$

where  $w$  is a vector of personalization parameters. In *edge-weighted personalized PageRank*, we solve

$$M(w)x(w) = (1 - \alpha)v, \quad w \in \mathbb{R}^d.$$

For example, the personalization vector can specify the topic preference for the query, so the random walker will teleport to nodes associated with preferred topics (node-weighted personalized PageRank) or move through edges associated with preferred topics more frequently (edge-weighted personalized

PageRank). The parameters are determined by users, queries, or domain experts [53, 14, 112, 107], or tuned by machine learning methods [82, 4, 12, 37, 38].

Despite the importance of both edge-weighted and node-weighted personalized PageRank, the literature shows an unfortunate dichotomy between the two problems. There exists a plethora of papers and efficient methods for node weight personalization (as we will discuss in Section 4.2), but not for the much harder problem of edge weight personalization! Yet many real applications are in dire need of edge-weighted personalized PageRank. For example, together with machine learning methods, edge-weighted personalized PageRank improves ranking results by incorporating user feedback [82, 12, 37, 38]; but training with existing methods takes hours. In TwitterRank [112], weighting edges by the similarity of user’s topical interests gives better results than just weighting the nodes according to those same interests. However, because of their inability to compute edge-weighted personalized PageRank online, the TwitterRank authors recommend simply taking a linear combination of topic-specific PageRank vectors instead of solving the “correct” problem of edge-weighted personalized PageRank.

In early work on the ObjectRank system, the authors mention the flexibility of personalizing edge weights as an advantage of their approach, but they do not describe a method for doing this fast. In subsequent work, the computation was accelerated by heuristics [106, 107], but it was still too slow. The later ScaleRank work [55] tried to address this issue, but only applies to a limited type of graph. Thus, despite its importance and serious attempts to address performance concerns, efficiently computing edge-weighted personalized PageRank remains an open problem.

In this chapter, we introduce the first truly fast method to compute  $x(w)$  in the edge-weighted personalized PageRank case. Our method is based on a novel *model reduction* strategy, where a reduced model is constructed offline, but it allows fast evaluation online. The speed of our new approach enables exciting interactive applications that have previously been impossible; for example, in one application described in Section 4.5, the reduced model is nearly *five orders of magnitude* faster than state-of-the-art algorithms. This is very important for standard algorithms with cost proportional to the graph size, but reduced models also speed up sublinear approximation algorithms for localized PageRank problems. Thus a variety of applications can benefit from the techniques that we describe.

We discuss some preliminaries in Section 4.2, then turn to the three main contributions of the chapter in the following sections. In Section 4.3, we describe the first scalable method for edge-weighted personalized PageRank by applying the idea of *model reduction*. We also show how to use common, simple parameterizations to make our methods even faster, and we reason about cost/accuracy tradeoffs in model reduction for PageRank. In Section 4.4, we show how fast reduced models enable new applications, such as interactive learning to rank, that were not previously possible. And in a thorough experimental evaluation in Section 4.5, we show that our method outperforms existing work by nearly *five orders of magnitude*. Finally, we discuss related work in Section 4.6.

## 4.2 Preliminaries

In standard (weighted) PageRank, we consider a random walk on a graph with weighted adjacency matrix  $A \in \mathbb{R}^{n \times n}$ , in which a nonzero entry  $A_{ji} > 0$  represents the weight on an edge from node  $j$  to node  $i$ . The weighted out-degree of node  $j$  is  $d_j = \sum_i A_{ji}$ , and we define the degree matrix  $D$  to be a diagonal matrix with diagonal entries  $d_j$ . Assuming no sinks (nodes with zero outgoing weight), the transition matrix for a random walk on this weighted graph is  $P = AD^{-1}$ . If the graph has sinks, one must define an alternative for what happens after a random walker enters a sink. For example, the walker might either stay put or teleport to some node in the graph chosen from an appropriate distribution.

When  $v$  is a dense vector (the *global* PageRank problem), the standard PageRank algorithm is the iteration (4.1), which can be interpreted as a random walk, a power iteration for an eigenvalue problem, or a Richardson or Jacobi iteration for (4.2) [39]. One can use more sophisticated iterative methods [71, 18], but (4.1) converges fast enough for many offline applications [39]. All these repeatedly multiply the matrix  $P$  by vectors, and therefore have arithmetic cost (and memory traffic) proportional to the number of edges in the graph. Hence, these methods are ill-suited to interactive applications for large graphs [13].

There are two standard methods to approximate PageRank online in time sublinear in the number of edges. The first case is when the teleportation vector  $v$  can be well approximated as a linear combination of a fixed set of reference distributions, i.e.  $v \approx Vw$  where  $V \in \mathbb{R}^{n \times d}$  and  $w \in \mathbb{R}^d$ ; see [58, 53]. For example, in web ranking, the  $j$ th column of  $V$  may indicate the importance of different nodes as authorities to a reference topic  $j$ , and the weight vector  $w$  indicates how

interesting each topic is to a particular ranking task. In this case, the solution to the PageRank problem is

$$x(w) = (1 - \alpha)M^{-1}Vw. \quad (4.3)$$

If  $M^{-1}V$  is precomputed offline by solving  $d$  ordinary PageRank problems, then any element of  $x(w)$  can be reconstructed in  $O(d)$  time, while the entire PageRank vector can be computed in  $O(nd)$  time. Because some nodes are unimportant to almost any topic, it may be unnecessary to compute every element of the PageRank vector.

We also can estimate PageRank in sublinear time when the restart vector  $v$  is sparse. In a common case that we refer to as *localized PageRank*, the vector  $v$  is one for an entry corresponding to a particular vertex and zero in all other components. Since the random surfer often returns to this vertex, the localized PageRank vector reveals properties of the region around this vertex. The Monte Carlo method [13] and Bookmark-Coloring Algorithm (BCA) [19, 7] estimate the PageRank values for the important nodes close to the target vertex, which are often the most important components of the PageRank vector in recommendation systems. Though these PageRank values can be computed with sub-second latency for a specific set of parameters, in an application requiring frequent recomputation for different users and queries, the total latency may still be significant.

Unfortunately, none of these methods apply to fast computation of edge-weighted personalized PageRank. We will discuss how to quickly approximate edge-weighted personalized PageRank via model reduction in the following section.

## 4.3 Model Reduction

Model reduction is a general approach to approximating solutions of large linear or nonlinear systems. Applied to PageRank, the key steps in model reduction are:

1. Observe that  $x(w)$  often lies close to a low-dimensional space. We build a basis for a  $k$ -dimensional reduced space in a data-driven way using the POD/PCA method on PageRank vectors for a sample of parameters, as we describe in Section 4.3.1
2. To pick an approximation in a  $k$ -dimensional reduced space, we need  $k$  equations. We describe how to choose these equations (offline) in Section 4.3.2, and how to form the reduced system of equations quickly (online) in Section 4.3.3–4.3.4.
3. After we solve the reduced system, we reconstruct the PageRank vector from the coordinates in the reduced space. We often do not need the full PageRank vector, and can compute a part of the vector at lower cost.

In the experiments in Section 4.5, we achieved good accuracy with reduced spaces of dimension  $k \approx 100$ .

### 4.3.1 Reduced Space Construction

The basic assumption in our methods is that for each parameter vector  $w \in \mathbb{R}^d$ , the PageRank vector  $x(w)$  is well approximated by some element in a low-dimensional linear space (the trial space). We construct the trial space in two

steps. First, we compute PageRank vectors  $\{x^{(j)}\}_{j=1}^r$  for a sample set of parameters  $\{w^{(j)}\}_{j=1}^r$ . Then we find a basis for a  $k$ -dimensional reduced space  $\mathcal{U}$  (typically  $k < r$ ) containing good approximations of each sample  $x^{(j)}$ .

In our work, we choose the sample parameters  $w^{(j)}$  uniformly at random from the parameter space. This works well for many examples, though a low-discrepancy sequence [81] might be preferable if the sample size  $r$  is small. Sparse grids [24] or adaptive sampling [86] might yield even better results, and we leave these to future work.

Once we have computed the  $\{x^{(j)}\}$ , we construct the reduced space  $\mathcal{U}$  by the proper orthogonal decomposition (POD) method, also known as principal components analysis (PCA) or a truncated Karhunen-Loève (K-L) expansion [90, 87]. We form the data matrix  $X = [x^{(1)}, x^{(2)}, \dots, x^{(r)}] \in \mathbb{R}^{n \times r}$  and compute the “economy” SVD

$$X = U^X \Sigma (V^X)^T$$

where  $U^X \in \mathbb{R}^{n \times r}$  and  $V^X \in \mathbb{R}^{r \times r}$  are matrices with orthonormal columns and  $\Sigma$  is a diagonal matrix with entries  $\sigma_1 \geq \dots \geq \sigma_r \geq 0$ . The best  $k$ -dimensional space for approximating  $X$  under the 2-norm error is the range of  $U = \begin{bmatrix} u_1^X & \dots & u_k^X \end{bmatrix}$ . The singular value  $\sigma_{k+1}$  bounds  $\min_y \|Uy - x^{(j)}\|_2$  for each sample  $x^{(j)}$ . If  $\sigma_{k+1}$  is small, the space is probably adequate; if  $\sigma_{k+1}$  is large, the trial space may not yield good approximations.

### 4.3.2 Extracting Approximations

Given a trial space and an online query parameter  $w$ , we want an element of the trial space that approximates  $x(w)$ . That is, if  $U = [u_1, u_2, \dots, u_k] \in \mathbb{R}^{n \times k}$ ,

we want an approximate PageRank vector of the form  $\tilde{x}(w) = Uy(w)$  for some  $y(w) \in \mathbb{R}^k$ . All of the methods we describe can be expressed in the framework of Petrov-Galerkin methods; that is, we choose  $y(w)$  to satisfy

$$W^T [M(w)Uy(w) - b] = 0$$

for some set of test functions  $W$ . We can also choose  $y(w)$  to force the entries of  $\tilde{x}(w)$  to sum to one, i.e.

$$\text{minimize } \|W^T [M(w)Uy(w) - b]\|^2 \text{ s.t. } \sum_j (Uy(w))_j = 1;$$

we solve this constrained linear least squares problem by standard methods [40, Chapter 6.2]. We consider two methods of choosing an approximate solution: *Bubnov-Galerkin* and the *Discrete Empirical Interpolation Method (DEIM)*.

In the Bubnov-Galerkin approach, we choose  $W = U$ ; that is, the trial space and the test space are the same. Because  $U$  is a fixed matrix, we can precompute the system matrices that arise from the Bubnov-Galerkin approach for simple edge weight parameterizations. However, this approach is expensive for more complicated problems.

In the DEIM approach, we enforce a subset of the equations. That is, we choose  $y$  to minimize the norm of a projected residual error:

$$\|\Pi [M(w)Uy(w) - b]\|^2,$$

where  $\Pi$  is a diagonal matrix that selects entries from some index set  $\mathcal{I}$ ; that is,

$$\Pi_{ii} = \begin{cases} 1, & i \in \mathcal{I} \\ 0, & \text{otherwise.} \end{cases}$$

Equivalently, we write the objective in the minimization problem as

$$\|M_{\mathcal{I},:}(w)Uy(w) - b_{\mathcal{I}}\|^2$$

where  $M_{\mathcal{I},:}(w)$  is the submatrix of  $M(w)$  involving only those rows in the index set  $\mathcal{I}$ , and similarly with  $b_{\mathcal{I}}$ . If  $|\mathcal{I}| = k$ , we enforce  $k$  of the equations in the PageRank system; otherwise, for  $|\mathcal{I}| > k$ , we enforce equations in a least squares sense.

Minimizing this objective is equivalent to Petrov-Galerkin projection with  $W = \Pi M(w)U$ . Because  $W$  has few nonzero rows, only a few rows of  $M(w)$  need ever be materialized, even when the parameterization is complicated. The key to this approach is a proper choice of the index set  $\mathcal{I}$  of equations to be enforced.

Once the reduced system is solved and the coordinates in the reduced space  $y(w)$  are located, we can reconstruct the approximate PageRank vector in the original space by  $\tilde{x}(w) = Uy(w)$ . This can be slow when the graph has millions of nodes, as it requires  $O(kn)$  work. In practice, we usually do not require the whole PageRank vector, but only the PageRank values for a subset of the vertices, such as the nodes with high PageRank values, or the nodes associated with some keywords. Assuming the subset of vertices we are interested in is  $\mathcal{S}$ , we can compute the PageRank values for the vertices in  $\mathcal{S}$  by  $\tilde{x}_{\mathcal{S}}(w) = U_{\mathcal{S},:} \cdot y(w)$ . The computation can be done with only  $O(k|\mathcal{S}|)$  work.

### 4.3.3 Parameterization Forms

How  $P(w)$  depends on  $w$  matters to how fast we can make different model reduction methods. For example, suppose  $P(w) = A(w)D(w)^{-1}$ , where the weighted adjacency matrix  $A(w)$  has edge weights that are general nonlinear functions in  $w$  and  $D(w)$  is the diagonal matrix of out-degree weights, as before. The DEIM approach would require that we compute the weight for all edges in

$E' = \{(i, j) \in E : j \in \mathcal{I}\}$ ; and also all the edges  $E'' = \{(i, j') \in E : \exists(i, j) \in E'\}$  in order to normalize. The expense of the Bubnov-Galerkin method in this case would be even worse: to form  $U^T M(w)U$  we would need to multiply  $P(w)$  by each of the  $k$  columns of  $U$ , at a cost comparable to running the PageRank iteration to convergence. We therefore consider two simple parameterizations for which the Bubnov-Galerkin and DEIM methods can be made more efficient.

In a *linear* parameterization, we assume

$$P(w) = \sum_{s=1}^m w_s P^{(s)}$$

where each  $P^{(i)}$  is column stochastic and the weights  $w_i$  are non-negative and sum to one. For a linear parameterization, the Bubnov-Galerkin approach is fast, since

$$U^T M(w)U = U^T U - \alpha \sum_{s=1}^m w_s U^T P^{(s)} U,$$

and the (small) matrices  $U^T U$  and  $U^T P^{(s)} U$  can be precomputed. The linear parameterization has been used several times in settings in which each edge has one of  $m$  different types [14, 107, 82]. In such settings, the parameterization corresponds to a generalized random walker model in which the walker first decides what type of edge to follow with probabilities given by the  $w_s$ , then decides between edges of that type. However, the model is limited in that it does not allow the probability of picking a particular edge type to vary across nodes.

In a *scaled linear* parameterization,  $P(w) = A(w)D(w)^{-1}$  where the weighted adjacency  $A(w)$  is

$$A(w) = \sum_{s=1}^m w_s A^{(s)}$$

and  $D(w)$  is the diagonal matrix of outgoing weights

$$d_i(w) = \sum_{s=1}^m w_s d_i^{(s)}, \quad d_i^{(s)} = \sum_{j=1}^n A_{ji}^{(s)}.$$

The scaled linear parameterization corresponds to choosing each edge weight  $A_{ji}(w)$  as a linear combination of edge features  $A_{ji}^{(s)}$ . For example, in a social network, the edge weights might depend on features like communication frequency, profile similarity, and the time since last communication [12]. Post topic similarity between users has also been adopted as an edge feature for sites such as Twitter [112]. Because of the normalization,  $M(w)$  in the scaled linear case is not a linear function of  $w$ , and so the Bubnov-Galerkin system cannot be written as a linear combination of pre-computed matrices. For DEIM in the scaled linear case, however, we can materialize only the subset of rows of  $A(w)$  with indices in  $\mathcal{I}$ , since the out-degree weight vector needed for normalization is a linear combination of the weight vectors  $d^{(i)}$ , which we can precompute.

#### 4.3.4 Choosing interpolation equations

As shown in Appendix 6.1, if the columns of  $U$  are normalized to unit 1-norm, the key quantity in the error bound for DEIM is  $\|(M_{\mathcal{I},:}U)^\dagger\|_1$ . We do not want interpolation nodes that are always unimportant, since in this case  $M_{\mathcal{I},:}U$  will have small elements, and this might suggest that we should choose “important” nodes according to the average of some randomly sampled PageRank vectors. However, this heuristic sometimes fails, as  $M_{\mathcal{I},:}U$  can be nearly singular even if the elements of  $M_{\mathcal{I},:}U$  are not nearly zero.

In the case  $|\mathcal{I}| = k$ , we want rows  $M_{\mathcal{I},:}U$  that are maximally linearly independent. While an optimal choice is hard in general, this type of *subset selection* problem is standard in linear algebra, and a standard approach to selecting the most important rows is to apply the pivoted QR algorithm and use the pivot

---

**Algorithm 4.1:** Row subset selection via QR

---

**In:**  $Z : n \times m$  matrix

**Out:**  $I$ : list of  $m$  indices

**def** FIND-I( $Z$ )

Initialize  $I \leftarrow \emptyset$ ,  $Q \leftarrow 0^{n \times m}$ ,  $r \leftarrow 0^m$

norm2  $\leftarrow$  squared norms of rows of  $Z$

**for**  $k = 1$  to  $m$  **do**

$s \leftarrow \operatorname{argmax}_{1 \leq i \leq n} \operatorname{norm}(i)$

$I \leftarrow I \cup \{s\}$ ,  $Q_{k,:} \leftarrow Z_{n,:}$ ,  $r_k \leftarrow \sqrt{\operatorname{norm2}(s)}$

$Q_{k,:} \leftarrow (Q_{k,:} - \sum_{l=1}^{k-1} \langle Q_{l,:}, Z_{n,:} \rangle Q_{l,:}) / r_k$

**for**  $i = 1$  to  $n$  **do**

norm2( $i$ )  $\leftarrow$  norm2( $i$ ) -  $\langle Z_{i,:}, Q_{k,:} \rangle^2$

**end for**

**end for**

**return**  $I$

---

order as a ranking [40, Chapter 5]. However, if we were to apply pivoted QR to the rows of  $M(w)U$  in order to choose the interpolation set, we would first have to compute  $M(w)U$ , which is again comparable in cost to computing PageRank directly. As an alternative, we evaluate  $M(\tilde{w}^{(j)})U$  for one or more test parameters  $\tilde{w}^{(j)}$ , then apply pivoted QR to the rows of  $Z = \begin{bmatrix} M(\tilde{w}^{(1)})U & \dots & M(\tilde{w}^{(q)})U \end{bmatrix}$ . By using more than one test parameter in the selection process, not only do we ensure that we are taking into account the behavior of  $M$  at more than one point, but we can choose as many as  $kq$  independent rows of the matrix  $Z$ .

For scaled linear and nonlinear parameterizations, the cost of forming the Bubnov-Galerkin matrix  $U^T M(w)U$  is linear in the size of the graph, and may

even exceed the cost of an ordinary PageRank computation. In contrast, for DEIM methods, the cost of forming the reduced system is proportional to the number of incoming edges for the nodes with indices in  $\mathcal{I}$  (in the scaled linear case) or those nodes plus all outgoing edges from those nodes (in the fully non-linear case). Hence, there is a performance advantage to choosing low-degree nodes for the interpolation set  $\mathcal{I}$ . At the same time, choosing nodes with high in-degree for the set  $\mathcal{I}$  may improve the accuracy of the reconstruction.

Note that the pivoted QR algorithm defines “utility” for the nodes, and it chooses the nodes with the highest utilities. The utility for vertex  $i$  is  $\sqrt{\text{norm}2(i)}$  described in Algorithm 4.1. We write the utility of vertex  $i$  as  $\text{util}(i)$  and the incoming degree as  $\text{cost}(i)$ . We propose two methods to manage the trade-off between  $\text{util}(i)$  and  $\text{cost}(i)$ : the cost-bounded pivot method and the threshold pivot method.

In the cost-bounded pivot method, a parameter  $C$  indicates the average cost we are willing to pay for each node. A straightforward heuristic would be choosing the nodes with highest  $\text{util}(i)$  with  $\text{cost}(i) \leq C$ . However, this heuristic ignores all the nodes that exceed the cost budget. We can soften the budget constraint by choosing the nodes with highest  $\text{util}(i) / \max(\text{cost}(i), C)$ . Alternatively, in the threshold pivot method, we require the node selected in each step to have utility higher than  $\max_i \text{util}(i) / F$ . Among these nodes, the one with smallest cost is selected. The parameter  $F$  indicates the maximum ratio allowed between the highest utility and the utility of the selected node in each step. We will discuss the trade-offs of the two methods in the experiment section.

## 4.4 Learning to Rank

We now describe an application enabled by fast PageRank approximations: learning to rank. In learning to rank, the goal is to learn the best values of the parameters that determine edge weights, based on training data such as user feedback or historic activities, as discussed in [12, 4]. Using surrogates, learning to rank becomes not only an interesting offline computation, but an online computation that could be done on a per-query basis.

As a concrete example, consider training data  $T = \{(i_q, j_q)\}$ , where each pair  $(i, j) \in T$  indicates that  $i$  should be ranked higher than  $j$ . The optimization problem is

$$\min_w L(w) = \sum_{(i,j) \in T} l(x_i(w) - x_j(w)) + \lambda \|w\|^2, \quad (4.4)$$

where  $x_i(w)$  and  $x_j(w)$  indicate the PageRank of nodes  $i$  and  $j$  for a parameter vector  $w$ . The loss  $l(\cdot)$  penalizes violations of the ranking preference; popular choices include squared loss and Huber loss [12]. To minimize  $L$  by gradient-based methods, one needs both  $L$  and the derivatives

$$\frac{\partial L}{\partial w_s} = \sum_{(i,j) \in T} l'(x_i - x_j) \frac{\partial(x_i - x_j)}{\partial w_s} + 2\lambda w_s, \quad (4.5)$$

To compute the derivatives of  $x$ , one uses the relation

$$M \frac{\partial x}{\partial w_s} = -\frac{\partial M}{\partial w_s} x; \quad (4.6)$$

that is, each partial derivative requires solving a linear system involving the PageRank matrix  $M(w)$ .

We replace the PageRank vector  $x(w)$  with an approximation  $\hat{x}(w) = Uy(w)$ , and seek to minimize the modified objective

$$\min_w \hat{L}(w) = \sum_{(i,j) \in T} l(\hat{x}_i(w) - \hat{x}_j(w)) + \lambda \|w\|^2. \quad (4.7)$$

We write the component differences  $\hat{x}_i(w) - \hat{x}_j(w)$  as  $z_{ij}y(w)$  where  $z_{ij} = U_{i,:} - U_{j,:}$  is the difference between two rows of  $U$ . The derivatives of  $\hat{L}$  are then

$$\frac{\partial \hat{L}}{\partial w_s} = \sum_{(i,j) \in T} l'(z_{ij}y(w)) z_{ij} \frac{\partial y(w)}{\partial w_s} + 2\lambda w_s. \quad (4.8)$$

Depending on the size of the training set, one might form the  $z_{ij}$  vectors at the start of the optimization.

Once  $y(w)$  has been evaluated for a given parameter vector  $w$ , the subsequent derivative computations can re-use factorizations, and so require less computation. For example, in the case of a linear parameterization and the Bubnov-Galerkin method, the derivatives of  $y$  satisfy

$$\tilde{M} \frac{\partial y}{\partial w_s} = -\frac{\partial \tilde{M}}{\partial w_s} y = \alpha \tilde{P}^{(s)} y, \quad (4.9)$$

where  $\tilde{M}(w) = U^T M(w) U$  is the same matrix that appears in the linear system for  $y(w)$ , while  $\tilde{P}^{(s)} = U^T P^{(s)} U$  is one of the matrices that was precomputed in the offline phase. Though computing  $y(w)$  requires  $O(k^3)$  work to factor  $\tilde{M}$  after it has been formed, the derivative computations can re-use this work, and involve only  $O(k^2)$  additional work to form the right hand side and solve the resulting system.

For a scaled linear parameterization or nonlinear parameterization with the DEIM method,  $y$  satisfies the normal equations

$$\tilde{M}^T \tilde{M} y = \tilde{M}^T b_I \quad (4.10)$$

where  $\tilde{M} = M_{I,:} U$ . Differentiating (4.10) gives the formula

$$\tilde{M}^T \tilde{M} \frac{\partial y}{\partial w_s} = -\tilde{M}^T \left( \frac{\partial \tilde{M}}{\partial w_s} \right) y - \left( \frac{\partial \tilde{M}}{\partial w_s} \right)^T \tilde{r} \quad (4.11)$$

where  $\tilde{r} = \tilde{M} y - b_I$  is the residual in the reduced least squares problem. Once we have computed  $\tilde{M}$  and its derivatives, computing  $y$  takes  $O(k^2 |I|)$  time, while ad-

ditional solves to evaluate partial derivatives take  $O(k|I|)$  time. In many graphs, though, the dominant cost is the time to form  $\tilde{M}$  and its derivatives.

## 4.5 Experiments

Our experimental evaluation has three goals: First, we want to show that *global edge-weighted personalized PageRank can be computed interactively with model reduction*. As reported in Section 4.5.3, our model reduction methods can answer such queries in half a second, while the standard power method takes 20 seconds (DBLP graph) to 1 minute (Weibo graph). Second, we want to verify that *model reduction improves the performance of localized PageRank*. As reported in Section 4.5.4, our methods are usually 1-2 orders of magnitudes faster than BCA. Third, we want to demonstrate that model reduction enables *learning-to-rank at interactive speeds*. As we report in Section 4.5.5, our model reduction methods are up to nearly five orders of magnitudes faster than the full computation on a learning-to-rank application. We discuss our experimental setup and preprocessing in Sections 4.5.1 and 4.5.2.

### 4.5.1 Setup

**Environment.** We run all experiments on a computer with two Intel Xeon X5650 2.67GHz processors and 48GB RAM. We have implemented most of our methods in C++ using Armadillo [95] for matrix computations. For the LP solver required by the ScaleRank algorithm, we follow the original implementation and use the GLPK package [1]. A few routines are currently prototyped with

SciPy [3].

**Datasets.** We run experiments on two graphs: the DBLP citation graph and the Weibo social graph. Table 4.1 shows their basic statistics. The DBLP graph, provided by ArnetMiner [103], has four types of vertices (paper, author, conference and venue) and seven edge types. We adopt the linear parameterization used in ObjectRank to assign edge weights [14, 55]. This graph is used for global PageRank and parameter learning.

The Weibo graph, released as part of the KDD Cup 2012 competition,<sup>1</sup> is a microblog dataset with subscription edges between users and text. In prior work [112], personalized PageRank with edge weights derived from topic analysis over user posts was used to rank topical authorities. In our experiments, we apply LDA to analyze the topics represented in user posts, following the approach in [21], with the number of topics set to 5, 10, and 20. We use both scaled-linear and linear parameterizations based on the user topic distributions. For the scaled-linear parameterization, we adopt a weighted cosine as the edge weight:

$$A_{ji}(w) = \sum_{s=1}^m w_s \cdot (\varphi_i)_s \cdot (\varphi_j)_s = \sum_{s=1}^m w_s A_{ji}^{(s)},$$

where  $\varphi_i$  denotes the topic distribution for user  $i$  and  $A_{ji}^{(s)} \equiv (\varphi_i)_s \cdot (\varphi_j)_s$ . We can also normalize the weighted adjacency matrix  $A^{(s)}$  for a linear parameterization:

$$P^{(s)} = A^{(s)}(D^{(s)})^{-1}.$$

We use the Weibo graph in experiments for both global and localized PageRank. For localized PageRank, we run the experiments on 1000 randomly selected nodes, as different nodes can have slightly different running time and accuracy for both our methods and the baseline.

<sup>1</sup><https://www.kddcup2012.org/c/kddcup2012-track1>

Table 4.1: Basic Statistics of Datasets

Dataset	# Vertices	# Edges	# Params
DBLP	3,494,258	18,515,718	7
Weibo	1,944,589	50,655,130	5, 10, 20

**Baselines.** We compare our methods to ScaleRank [55] for global PageRank and the Bookmark Coloring Algorithm (BCA) [19, 7] for localized PageRank. We discuss these methods in more detail in Section 4.6. For global PageRank, ScaleRank [55] is the only previous work we know to efficiently compute edge-weighted personalized PageRank. Like our approach, ScaleRank has an offline phase that samples the parameter space, and an online phase in which queries are processed interactively. We only report the performance of ScaleRank on the DBLP graph, as it is ill-suited for general graphs. We set the parameters according to the original paper. For localized PageRank, prior methods approximate the PageRank of the top- $k$  nodes in sub-second time without preprocessing. We compare our model reduction methods with a variant of BCA proposed in [7]. For the BCA termination threshold  $\epsilon$ , which controls the trade-off between running time and accuracy, we use  $\epsilon = 10^{-8}$ ; this leads to Kendall  $\tau$  values of around 0.01.

**Metrics.** We evaluate both how well our approximate PageRank values match the true PageRank values and the distance between the induced rankings. We denote the exact and approximate PageRank vectors as  $x$  and  $\tilde{x}$ , respectively. We measure accuracy of the approximate PageRank values by the normalized

L1 distance. The normalized L1 distance on the evaluation set  $\mathcal{S}$  is defined as

$$NL_1(x, \tilde{x}, \mathcal{S}) = \frac{\|x_{\mathcal{S}} - \tilde{x}_{\mathcal{S}}\|_1}{\|x_{\mathcal{S}}\|_1}.$$

For global PageRank,  $\mathcal{S}$  contains all the nodes, while for localized PageRank,  $\mathcal{S}$  is the exact top-100 set. To evaluate ranking, we adopt Kendall’s  $\tau$ , the *de facto* standard for ranking measurement [69]. Denote the number of concordant pairs and discordant pairs as  $n_C$  and  $n_D$ , respectively; the normalized Kendall distance is defined as the percentage of pairs that are out of order, i.e.

$$\tau' = \frac{n_D}{n_C + n_D},$$

More specifically, we compute  $\tau'$  over the union of the exact and the approximated top-100 sets.

For both metrics, we report average results over 100 random test parameters.

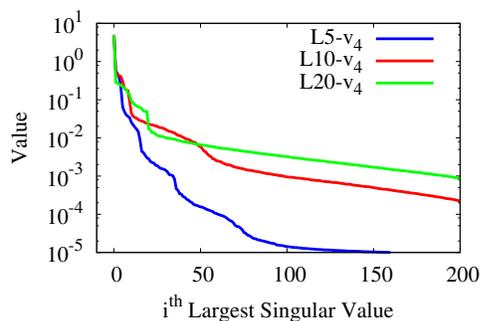
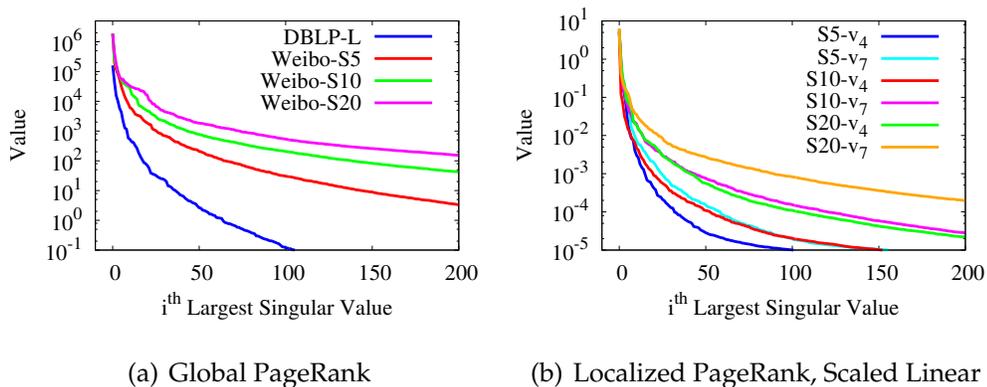
## 4.5.2 Preprocessing

**Reduced Space Construction.** For all experiments, we construct the reduced space from PageRank vectors computed at 1000 uniformly sampled parameters.

We report the singular values of the global PageRank data matrix (see Section 4.3) for both DBLP and the Weibo graph in Figure 4.1(a). We examine the singular values of the localized PageRank data matrix for 10 randomly selected nodes and report them in Figures 4.1(b) and 4.1(c). For the scaled-linear parameterization, we found that for a given number of parameters, the singular values for most of our sample nodes decay at a similar rate, though one or two nodes show singular values with slower decay. We show the singular values for

two representative nodes, denoted as  $v_4$  and  $v_7$ , in Figure 4.1(b). For the linear parameterization, the singular values for all the nodes decay at a similar speed; thus we pick one node to report in Figure 4.1(c). Unsurprisingly, the singular values decay more slowly when there are more parameters on the Weibo graph. With the same number of parameters, the singular values decay more rapidly for the scaled-linear than for the linear parameterization.

For most experiments, we set the reduced dimension to  $k = 100$ , as the singular values either decay slowly after this point, or they are already small enough. The only exception is localized PageRank with the scaled-linear parameterization, where we set  $k = 50$  for 5 and 10 parameters, as the singular values are already quite small there.



(c) Localized PageRank, Linear

Figure 4.1: Singular Values

**Interpolation Set Selection.** We use the pivoted QR method discussed in Section 4.3 in our experiments. For graphs with skewed incoming degree distributions, we need to restrict the total number of incoming edges for the nodes in  $\mathcal{I}$  for performance. For both cost-bounded pivot and threshold pivot methods, we compute the interpolation set with different values for the parameter  $C$  or  $F$ , and select the  $\mathcal{I}$  that works best on a small validation set of personalized PageRank parameters<sup>2</sup>. As the incoming degree distribution for DBLP is not heavy-tailed, we use the unconstrained pivoted QR method. For the Weibo graph, we use cost-bounded pivot with  $C = 100$  and threshold pivot with  $F = 10$  for localized PageRank, and cost-bounded pivot with  $C = 1000$  for global PageRank.

With  $|\mathcal{I}| = k$ , the reduced system is nearly singular for some parameters. We can avoid this issue by slightly increasing the size of  $|\mathcal{I}|$  (i.e.  $1.2k$ ). This suggests that there is no single best interpolation set with size  $k$  for all parameters, and the more robust choice for DEIM is to choose  $|\mathcal{I}| > k$ . Further increasing the size of the interpolation set in DEIM produces more accurate results, with some increase in running time. We set  $|\mathcal{I}| = 2k$  for all the experiments; the extra cost over choosing  $k$  nodes is modest, and we usually see little improvement after this point.

**Preprocessing Time.** We show preprocessing times in Table 4.2. For global PageRank, we compute sample PageRank vectors using the standard power method with stopping criterion  $\|x^{(t+1)} - x^{(t)}\|_1 < 10^{-10}$ . For localized PageRank, we use BCA with  $\epsilon = 10^{-9}$ ; in this case, the preprocessing time slightly varies with different numbers of parameters and parameterization forms. In general, global

---

<sup>2</sup> We try  $C = 10^0, 10^1, \dots, 10^5$  and  $F = 5, 10, 20, 40, 80$ .

Table 4.2: Preprocessing Time on Different Datasets

Procedure	DBLP	Weibo-Global	Weibo-Local
Prepare Samples	6 hours	17 hours	3-7 min
Get Basis $U$	0.8 hours	0.4 hours	1-2 min
Prepare $U^T P^{(l)} U$ <sup>3</sup>	2 min	N/A	< 2 min
Choose $\mathcal{I}$ <sup>4</sup>	11 min	12-18min	< 1 min

PageRank takes hours of CPU time for preparation while localized PageRank takes minutes. In either case, the computation can be easily parallelized across samples.

### 4.5.3 Global PageRank

In this section, we discuss the results for global PageRank. The standard power iteration is ill-suited for interactive global PageRank – it takes about 20 seconds on the DBLP graph and a minute on the Weibo graph.

**DBLP Graph.** We conduct experiments on the DBLP graph to demonstrate the performance of model reduction methods for linearly parameterized problems. Figure 4.2 and Figure 4.3 show the running time and accuracy of different methods. The running time has two parts: finding the coefficients in the reduced space, and constructing the PageRank vector as a linear combination of reduced basis vectors. Both model reduction approaches are accurate. For DEIM with  $|\mathcal{I}| = 200$ , the Kendall distance for the top 100 results is around  $3 \times 10^{-5}$ , while

<sup>3</sup> Used by Bubnov-Galerkin.

<sup>4</sup> Used by DEIM.

the normalized L1 distance is around 0.0005. The Bubnov-Galerkin method has even lower error. In contrast, ScaleRank has both Kendall distance and normalized L1 distance around 0.07.

All the methods produce results with interactive latency, and the model reduction methods run slightly faster than ScaleRank. Most of the time for model reduction is spent on PageRank vector construction. As discussed in Section 4.3.2, this time can be largely reduced by materializing the PageRank values only for a subset of the nodes. In contrast, the ScaleRank method spends much more time in obtaining the coefficients, and this cost cannot be easily reduced. This is because ScaleRank requires solving several linear programs and is much slower than solving a linear system.

**Weibo Graph.** We run experiments on the Weibo graph to see how model reduction methods work for social graphs with scaled-linear parameterization. We use DEIM in this case, since the Bubnov-Galerkin method is slow for scaled-linear parameterization. Because the DEIM approximation vector usually does not sum to one for this problem, we add the constraint discussed in Section 4.3. In Figure 4.5(a) and Figure 4.5(b), we report the running time and accuracy for different numbers of parameters. The reduced model can answer queries with interactive latency, and the running time would be much less if we only reconstructed the PageRank values for a small subset of vertices. The error increases with the number of parameters, as increasing the number of parameters increases the error intrinsic in approximating solutions from a low-dimensional space. The Kendall distance is about 0.005 with 10 parameters, but it increases to about 0.013 when there are 20 parameters.

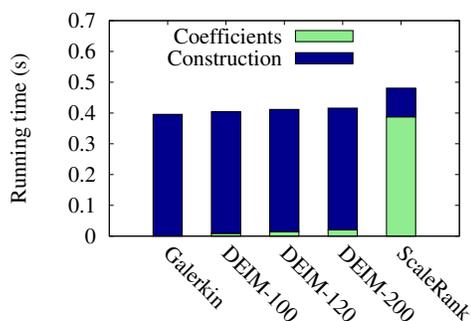


Figure 4.2: Running Time on DBLP Graph

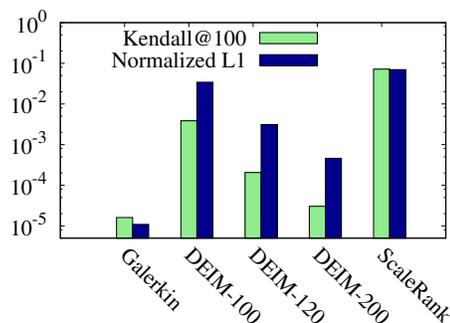


Figure 4.3: Accuracy on DBLP Graph

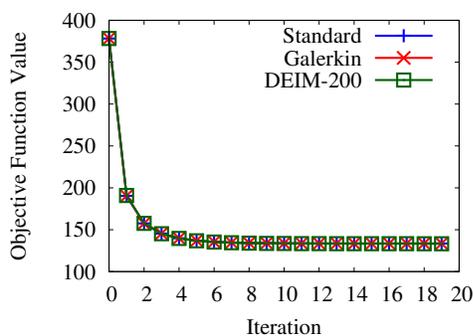


Figure 4.4: Objective Function Value for Parameter Learning

#### 4.5.4 Localized PageRank

In this section, we discuss localized PageRank on the Weibo graph. We report the Kendall distance for accuracy, as the approximations computed by model reduction always have much lower L1 distance than BCA. With fewer parameters, we see better performance for our methods.

In most applications of localized PageRank, we are only interested in a relatively small number of top-ranked nodes. To find the top 100 nodes, it generally suffices to compute PageRank values for the most promising 10000 nodes as indicated by average PageRank values over the sampled parameters. To construct

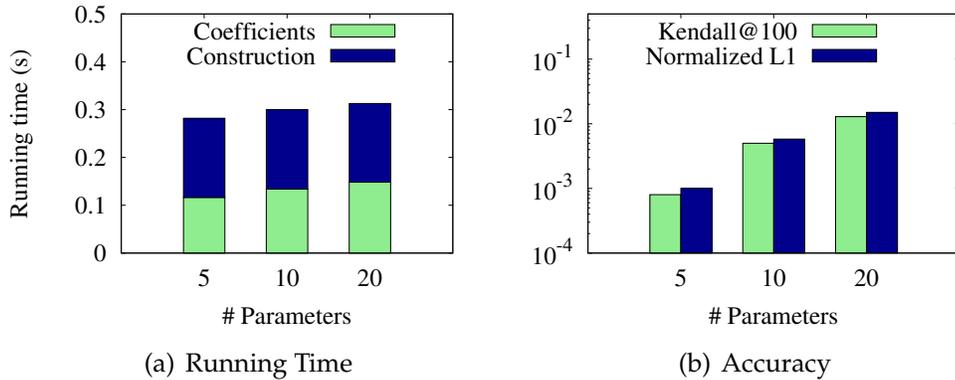


Figure 4.5: Global PageRank on Weibo Graph

the part of the PageRank vector associated with these 10000 nodes requires less than a millisecond. The running time for model reduction method reported in this section assumes constructing the PageRank values only for top vertices.

**Linear Parameterization.** For a linear parameterization, the only work in forming the Bubnov-Galerkin system is adding  $m$  matrices of size  $k \times k$ . In our experiments, with  $k = 100$  and  $m = 20$ , we are able to form and solve the reduced system within a millisecond.

For the problem to retrieve top 100 nodes, the Bubnov-Galerkin approach is nearly two orders of magnitudes faster than BCA, with better accuracy. We report the running time and accuracy in Figure 4.6. The Bubnov-Galerkin method has much better accuracy with  $m = 5, 10$  than for  $m = 20$ . For  $m = 20$ , the error increases as the 100-dimensional trial space is too small to contain highly accurate approximations to the PageRank vector. Still, the Bubnov-Galerkin method produces more accurate results compared with BCA.

**Scaled-Linear Parameterization.** For the scaled-linear parameterization, the Bubnov-Galerkin approach is expensive, and we turn instead to DEIM. The run-

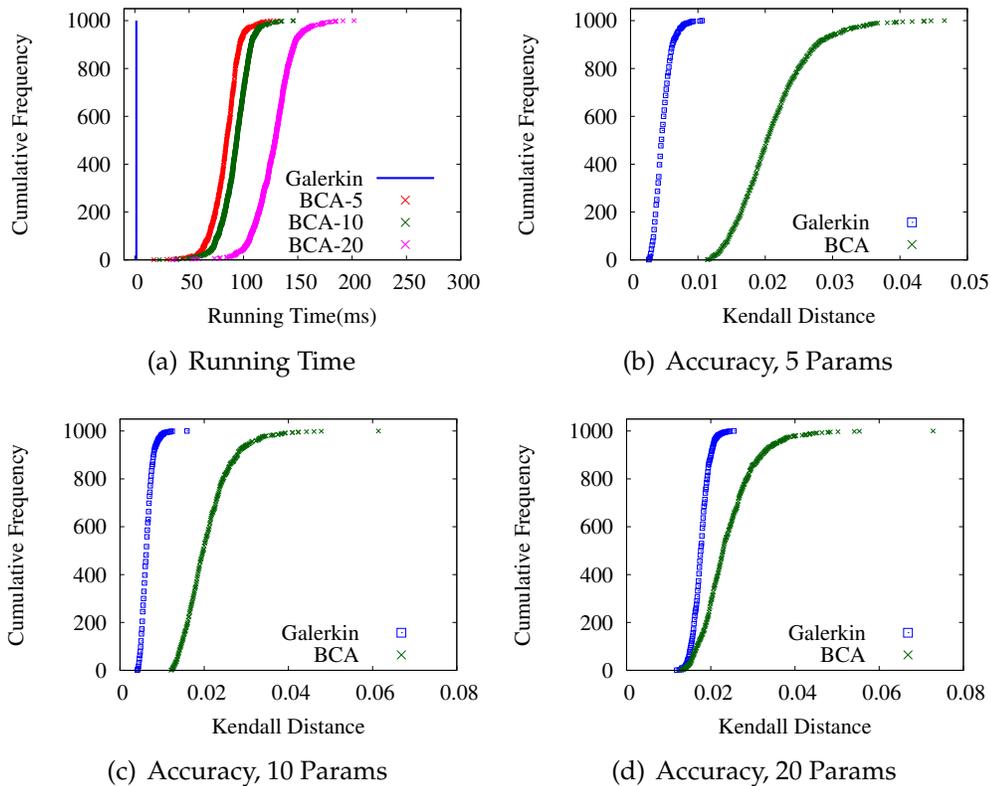


Figure 4.6: Localized PageRank with Linear Parameterization

ning time and accuracy are reported in Figure 4.7. For  $m = 5, 10$ , DEIM answers the query nearly one order of magnitude faster than BCA with better accuracy. For  $m = 20$ , DEIM is much slower than for  $m = 5, 10$  for two reasons. First, because the edge weight computation is proportional to the number of parameters, it takes more time to form  $M_{T,:}(w)$ . Second, we need a larger reduced space ( $k = 100$ ) to achieve acceptable accuracy.

In all cases, while the interpolation sets selected by the cost-bounded pivot method result in shorter running time, it produces less accurate results for a tiny fraction of outlier nodes. The threshold pivot method selects the interpolation sets in a more robust way, but it also requires longer time for each PageRank computation.

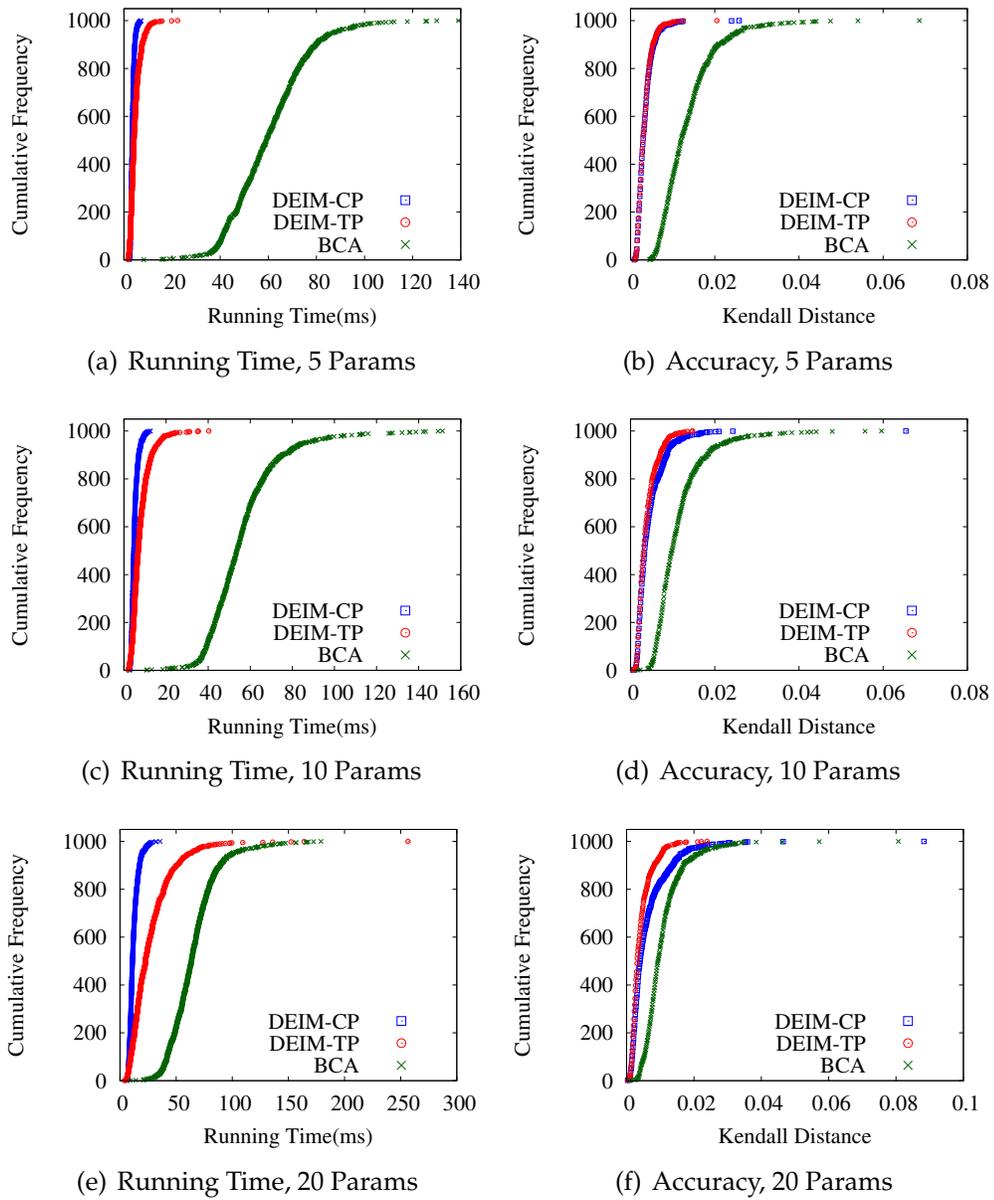


Figure 4.7: Localized PageRank with Scaled-Linear Parameterization

### 4.5.5 Parameter Learning

In this section, we demonstrate that parameter learning for edge-weighted personalized PageRank can be done at interactive rates via model reduction. For our experiments, we considered the DBLP graph with linear parameterization

over the edge types, similar to the setting of PopRank [82]. A partial ranking list with eight papers was used in our experiments to represent the feedback received from the user. Although the original PopRank model is based on a non-differentiable objective function, we use a differentiable objective suggested in later work [4, 12] as it achieves the same goal with much less training time. Following the parameter learning framework described in Section 4.4, we choose the squared loss with margin  $b = 0.2$  and set  $\lambda = 1000$  for the regularizer. That is, we minimize the loss function

$$\sum_{(i,j) \in T} \max(x_j(w) - x_i(w) + 0.2, 0)^2 + 1000 \|w - w_0\|_2^2.$$

Here  $T$  is the set of pairwise ranking preferences based on the partial ranking list, and  $w_0$  is the default parameter  $(0.34, 0.33, 1.0, 0.33, 0.75, 0.25, 1.0)$ .

Figure 4.4 shows the value of the objective function after each iteration with different methods, and all the methods result in almost the same objective function value. As expected, the parameters after each iteration with different methods are also very similar.

The average running time for each iteration is reported in Table 4.3. For applications involving interactive learning, user feedback usually just slightly adjusts the parameters, and 5-10 optimization iterations should be enough to produce the adjusted parameters to reflect the user's personal preference. In fact, as shown in Figure 4.4, the objective function value does not change much after eight iterations. Assuming ten iterations are required to produce the new ranking result after taking user feedback, for linear parameterization, we are able to finish the learning procedure in 0.02 seconds by the Bubnov-Galerkin method. Otherwise, DEIM is required to give the adjusted ranking within a second. In comparison, the original method takes minutes for each optimization

iteration, as a separate PageRank computation is required for the objective and for each component of the gradient.

Table 4.3: Avg. Running Time per Opt. Iteration

Method	Standard	Bubnov-Galerkin	DEIM-200
Time(sec)	159.3	0.002	0.033

## 4.6 Related Work

**Personalized PageRank Computation.** There has been a lot of work on fast personalized PageRank computation. A recent review can be found in [13]. However, as discussed in Section 4.2, most previous work focuses on node-weighted personalized PageRank and does not apply to edge-weighted personalized PageRank.

To the best of our knowledge, ScaleRank [55] is the only other work that answers edge-weighted personalized PageRank queries interactively. Like our approach, this method computes sample PageRank vectors  $\{x^{(j)}\}_{j=1}^r$  in the offline phase, and the approximation to the online query is a linear combination of these samples. However, extracting the linear combination in ScaleRank is slower, as a series of linear programming problems must be solved. It is also unclear how to support learning-to-rank application in this framework. Furthermore, it can only find approximate solutions efficiently on typed graphs, as general graphs would introduce too many constraints in the linear programs.

For localized PageRank, the Monte Carlo method [13] and the Bookmark Coloring Algorithm (BCA) [7] have been proposed to efficiently find the approx-

imate PageRank vector without preprocessing. These methods only explore a localized region close to the target vertex, as the nodes far from the target node usually have negligible PageRank value. The Monte Carlo method simply simulates random walks starting from the target node. BCA maintains a running estimate of the PageRank vector and the residual vector, and repeatedly picks a node and “pushes” its residual to the outgoing nodes until all the residuals are smaller than a predefined threshold.

**Surrogate Model.** Another approach is to approximate  $x(w) = M(w)^{-1}b$  by a fast *surrogate model*  $\hat{x}(w) \approx x(w)$ . One could build surrogates by interpolation or regression, whether via polynomials, radial basis functions, Gaussian processes, or neural networks. In these methods, PageRank is a black box: the methods sample the PageRank vector during model construction, but do not use the structure of the underlying linear system during model evaluation. An example of this approach is [29, 30, 31, 32], in which high-degree polynomial interpolation is used to approximate the dependence of the PageRank vector on the teleportation parameter  $\alpha$ . In contrast, we pursue a *model reduction* strategy that uses the PageRank equations.

**Model Reduction in Simulation.** In physical simulations, one uses model reduction for tasks that involve repeated model evaluation, such as design, optimization, control, and uncertainty quantification. Our work is inspired by the work of Patera and co-workers on the use of reduced basis methods for elliptic PDEs with uncertain (or stochastic) coefficients [46], and by work on model reduction of dynamical systems in the time or frequency domain [10, 96, 26, 11].

## CHAPTER 5

### CONCLUSIONS

Iterative graph computation is a key component in a variety of applications. The big data era gives rise to new challenges related to the volume, velocity and variety of data. In this dissertation, we have addressed challenges posed by these three aspects of big data.

First, we have presented a new block-oriented computation model that is compatible with vertex-centric programming abstraction but executes a block of highly connected vertices at a time instead of one vertex at a time. Our block-aware graph execution engine can achieve better cache performance and enables more flexible block-level and vertex-level scheduling to further accelerate convergence. In particular, it can provide near-interactive runtime and better multicore speedup than current per-vertex computation models for a large class of graph processing applications. We believe that this work is only a first step towards more confluence between the HPC and the database communities and a major step towards enabling iterative graph processing with interactive response times, a fascinating topic for future research. We are aware that other framework providers are actively improving the performance of their graph processing engines. However, our block-oriented execution model provides an orthogonal perspective on optimizing computationally light graph applications, and we believe that it is applicable to other frameworks as well.

Second, we have described TIDE, a distributed system for analyzing dynamic graphs. TIDE employs a model based on probabilistic edge decay to implement a temporally biased sampling scheme that allows controlled trade-offs between emphasizing recent interactions and providing continuity with

respect to past interactions; the PED model generalizes existing snapshot and sliding-window models. To facilitate maintenance of a set of sample graphs, we have provided both provably compact “aggregate” representations and efficient incremental updating methods. We have also introduced a bulk execution model to simultaneously process these graphs using the same programming APIs as are found in existing static graph processing systems. Through experiments on a Twitter dataset, we have demonstrated the effectiveness and efficiency of our proposed methods for tasks such as identifying key influencers and exploring community structure. Future work includes investigating decay functions beyond the exponential function and leveraging results from the probabilistic database community to obtain a more comprehensive set of analysis techniques for sample graphs.

Third, we have presented the first general scalable method for edge-weighted personalized PageRank based on model reduction. We discuss optimizations for common parameterizations and cost/accuracy tradeoffs when applying model reduction to power-law graphs. For applications such as learning to rank, our model reduction methods are nearly *five orders of magnitudes* faster than the standard approach. In future work, we plan to investigate how to choose the best index set for the DEIM method under cost constraints. Another interesting direction is to investigate whether the PageRank computations can be pushed to the client side by sending the reduced model. We would also like to investigate how to update the reduced model efficiently as the graph evolves over time.

CHAPTER 6  
APPENDIX FOR CHAPTER 5

## 6.1 Quasi-optimality

Petrov-Galerkin methods are *quasi-optimal*: the error in the Petrov-Galerkin approximation is within some factor of the best error possible in the space. We summarize with the following theorem.

**Theorem 1.** *Suppose  $\|e_*\| = \min_{y_*} \|Uy_* - x\|$  is the error in the best approximation from  $\mathcal{R}(U)$  in some norm. The error  $\|e\| = \|Uy - x\|$  for the Petrov-Galerkin approximation is bounded by*

$$\|e\| \leq (1 + \kappa)\|e_*\|$$

for  $\kappa = \|U\| \|(W^T M U)^{-1}\| \|W^T M\|$ . For the DEIM method, the error is bounded by

$$\|e\| \leq (1 + \kappa_{\text{DEIM}})\|e_*\|$$

for  $\kappa_{\text{DEIM}} = \|U\| \|(M_{I,:} U)^\dagger\| \|M_{I,:}\|$  where  $(M_{I,:} U)^\dagger$  indicates the Moore-Penrose pseudoinverse, i.e. the least-squares solution operator.

*Proof.* Suppose  $Uy_*$  is the best approximation in the space to  $x = M^{-1}b$  under some norm. Let  $e = x - Uy$  and  $e_* = x - Uy_*$  be the error in the chosen approximation and the best approximation, respectively. Substituting  $b = M(Uy_* + e_*)$  into the Petrov-Galerkin ansatz yields

$$W^T M U(y - y_*) = W^T M e_*,$$

and therefore

$$e = e_* - U(y - y_*) = \left[ I - U(W^T M U)^{-1} W^T M \right] e_*.$$

Taking norms, we have

$$\|e\| \leq (1 + \|U\| \|(W^T M U)^{-1}\| \|W^T M\|) \|e_*\|.$$

In the DEIM case, a similar argument yields

$$e = [I - U(M_{I,:} U)^\dagger M_{I,:}] e_*$$

and taking norms as before yields the final result.  $\square$

The most significant term in this bound is the norm of the inverse of the projected system, i.e.  $\|(W^T M U)^{-1}\|$  or  $\|(M_{I,:} U)^{-1}\|$ . In particular, if the columns of  $U$  are normalized to have absolute sums equal to zero, then  $\|U\|_1 = 1$  and  $\|M_{I,:}\|_1 \leq 1 + \alpha$ , so that for the one-norm the quasi-optimality constant is bounded by

$$\kappa_{\text{DEIM}} \leq (1 + \alpha) \|(M_{I,:} U)^\dagger\|_1.$$

That is, the key quantity in this case is  $\|(M_{I,:} U)^\dagger\|_1$ , which measures how close the projected system is to being singular. When evaluating the reduced model, we can bound this quantity with little extra cost via Hager's algorithm or variants [51, 54].

## 6.2 Model Reduction with Constraints

Under the constraint that the entries of the approximate PageRank vectors must sum to 1, extracting the approximation becomes a constrained optimization problem

$$\begin{aligned} & \underset{y(w)}{\text{minimize}} && \|W^T(M(w)Uy(w) - b)\|_2 \\ & \text{subject to} && e^T U y(w) = 1, \end{aligned}$$

where  $e$  is the vector of all ones.

Such constrained least square problem can be solved via an augmented system through the method of Lagrange multiplier [40, Chapter 6.2].

$$\begin{bmatrix} \tilde{M}^T \tilde{M} & u \\ u^T & 0 \end{bmatrix} \begin{bmatrix} y \\ \lambda \end{bmatrix} = \begin{bmatrix} \tilde{M}^T \tilde{b} \\ 1 \end{bmatrix}$$

where  $\tilde{M} = W^T M(w)U$ ,  $\tilde{b} = W^T b$ ,  $u = U^T e$ .

## BIBLIOGRAPHY

- [1] GNU Linear Programming Kit. <http://www.gnu.org/software/glpk/>.
- [2] Scaling apache giraph to a trillion edges. <http://on.fb.me/1czMarU>.
- [3] SciPy. <http://www.scipy.org/index.html>.
- [4] Alekh Agarwal, Soumen Chakrabarti, and Sunny Aggarwal. Learning to rank networked entities. In *KDD*, 2006.
- [5] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael I. Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *SIGMOD*, pages 481–492, 2014.
- [6] Charu C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *VLDB*, 2006.
- [7] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using PageRank vectors. In *FOCS*, 2006.
- [8] Reid Andersen, David F. Gleich, and Vahab S. Mirrokni. Overlapping clusters for distributed computation. In *WSDM*, 2012.
- [9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [10] Athanasios C Antoulas. *Approximation of Large-Scale Dynamical Systems*. SIAM, 2005.
- [11] Athanasios C Antoulas, Christopher A Beattie, and Serkan Gugercin. Interpolatory model reduction of large-scale dynamical systems. In *Efficient modeling and control of large-scale systems*, pages 3–58. Springer, 2010.
- [12] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, 2011.

- [13] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.
- [14] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.
- [15] Stephen T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *SC*, page 27, 1995.
- [16] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [17] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O’Neil, and Patrick E. O’Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.
- [18] Pavel Berkhin. A survey on PageRank computing. *Internet Mathematics*, 2(1):73–120, 2005.
- [19] Pavel Berkhin. Bookmark-Coloring Algorithm for Personalized PageRank Computing. *Internet Mathematics*, 3:41–62, 2006.
- [20] D.P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1995.
- [21] Bin Bi, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Junghoo Cho. Scalable topic-specific influence analysis on microblogs. In *WSDM*, 2014.
- [22] Stephen P. Borgatti and Martin G. Everett. A graph-theoretic perspective on centrality. *Social Networks*, 28(4):466–484, 2006.
- [23] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [24] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004.
- [25] Adam Chacon and Alexander Vladimirovsky. Fast two-scale methods for eikonal equations. *SIAM J. Scientific Computing*, 34(2), 2012.

- [26] Saifon Chaturantabut and Danny C Sorensen. Nonlinear model reduction via discrete empirical interpolation. *SIAM J. Sci. Comput.*, 32(5):2737–2764, 2010.
- [27] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kinograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, 2012.
- [28] Edith Cohen and Martin J. Strauss. Maintaining time-decaying stream aggregates. *J. Algorithms*, 59(1):19–36, 2006.
- [29] Paul G Constantine and David F Gleich. Using polynomial chaos to compute the influence of multiple random surfers in the pagerank model. In *Algorithms and Models for the Web-Graph*, pages 82–95. Springer, 2007.
- [30] Paul G Constantine and David F Gleich. Random alpha PageRank. *Internet Mathematics*, 6(2):189–236, 2009.
- [31] Paul G Constantine, David F Gleich, and Gianluca Iaccarino. Spectral methods for parameterized matrix equations. *SIAM J. Matrix Anal. Appl.*, 31(5):2681–2699, 2010.
- [32] Paul G Constantine, David F Gleich, and Gianluca Iaccarino. A factorization of the spectral Galerkin system for parameterized matrix equations: Derivation and applications. *SIAM J. Sci. Comput.*, 33(5):2995–3009, 2011.
- [33] David Crandall, Andrew Owens, Noah Snavely, and Daniel P. Huttenlocher. Discrete-Continuous optimization for large-scale structure from motion. In *CVPR*, pages 3001–3008, 2011.
- [34] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
- [35] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Dynamic Graph Algorithms*. CRC Press, 1999.
- [36] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *PVLDB*, 5(11), 2012.
- [37] Wei Feng and Jianyong Wang. Incorporating heterogeneous information for personalized tag recommendation in social tagging systems. In *KDD*, 2012.

- [38] Bin Gao, Tie-Yan Liu, Wei Wei, Taifeng Wang, and Hang Li. Semi-supervised ranking on very large graphs with rich metadata. In *KDD*, 2011.
- [39] David F. Gleich. PageRank beyond the web. *arXiv*, cs.SI:1407.5107, 2014.
- [40] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, fourth edition, 2012.
- [41] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3. ed.)*. Johns Hopkins University Press, 1996.
- [42] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
- [43] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
- [44] Murmurhash. [sites.google.com/site/murmurhash](http://sites.google.com/site/murmurhash).
- [45] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.
- [46] Martin A Grepl, Yvon Maday, Ngoc C Nguyen, and Anthony T Patera. Efficient reduced-basis treatment of nonaffine and nonlinear partial differential equations. *ESAIM: Mathematical Modelling and Numerical Analysis*, 41(03):575–605, 2007.
- [47] Michael Griebel and Peter Oswald. Greedy and randomized versions of the multiplicative Schwarz method. *Linear Algebra Appl.*, 437(7):1596–1610, 2012.
- [48] Peter Grindrod and Desmond J. Higham. A matrix iteration for dynamic network summaries. *SIAM Rev.*, 55(1):118–128, 2013.
- [49] William D. Gropp and David E. Keyes. Domain decomposition on parallel computers. *Impact Comput. Sci. Eng.*, 1:421–439, 1989.
- [50] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.

- [51] William Hager. Condition estimates. *SIAM J. Sci. Stat. Comput.*, 5(2):311–316, 1984.
- [52] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, Francois Paneton, and Pierre L'Ecuyer. Efficient Jump Ahead for 2-Linear Random Number Generators. *INFORMS Journal on Computing*, 20(3):385–390, 2008.
- [53] Taher H Haveliwala. Topic-sensitive PageRank. In *WWW*, 2002.
- [54] Nicholas J. Higham and Franoise Tisseur. A block algorithm for matrix 1-norm estimation with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21:1185–1201, 2000.
- [55] Vagelis Hristidis, Yao Wu, and Louiqa Raschid. Efficient ranking on entity graphs with personalized relationships. *IEEE Trans. Knowl. Data Eng.*, 26(4):850–863, 2014.
- [56] Ihab F. Ilyas and Mohamed A. Soliman. *Probabilistic Ranking Techniques in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
- [57] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: An optimizing framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18:135–158, 2004.
- [58] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *WWW*, 2003.
- [59] Bin Jiang. Ranking spaces for predicting human movement in an urban environment. *International Journal of Geographical Information Science*, 23(7):823–837, 2009.
- [60] Karthik Kambatla, Naresh Rapolu, Suresh Jagannathan, and Ananth Grama. Asynchronous algorithms in MapReduce. In *CLUSTER*, 2010.
- [61] U. Kang, Duen Horng Chau, and Christos Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *ICDE*, 2011.
- [62] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, 2009.

- [63] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *DAC*, 1997.
- [64] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
- [65] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.
- [66] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.
- [67] Yehuda Koren. Collaborative filtering with temporal dynamics. *Commun. ACM*, 53(4):89–97, 2010.
- [68] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.
- [69] Ravi Kumar and Sergei Vassilvitskii. Generalized distances between rankings. In *WWW*, 2010.
- [70] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.
- [71] Amy N Langville and Carl D Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.
- [72] A. M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, fifth edition, 2014.
- [73] Pierre L’Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), 2007.
- [74] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and Homer H. Chen. Hardware-efficient belief propagation. In *CVPR*, 2009.
- [75] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos

- Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.
- [76] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
- [77] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
- [78] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.
- [79] R. G. Miller. *Simultaneous Statistical Inference*. Springer, second edition, 1981.
- [80] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, 2012.
- [81] William J. Morokoff and Russel E. Caflisch. Quasi-random sequences and their discrepancies. *SIAM J. Sci. Comput.*, 15(6):1251–1279, 1994.
- [82] Zaiqing Nie, Yuanzhi Zhang, Ji-Rong Wen, and Wei-Ying Ma. Object-level ranking: bringing order to web objects. In *WWW*, 2005.
- [83] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communications, and Computing*, 18:297–311, 2007.
- [84] J-S. Park, M. Penner, and V. Prasanna. Optimizing graph algorithms for improved cache performance. In *IPDPS*, 2002.
- [85] D. Patterson and J. Hennessey. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.
- [86] A. Paul-Dubois-Taine and David Amsallem. An adaptive and efficient greedy procedure for the optimal training of parametric reduced-order models. *Int. J. Numer. Meth. Engng.*, 2014.
- [87] René Pinnau. Model reduction via proper orthogonal decomposition. In

- Model Order Reduction: Theory, Research Aspects and Applications*, pages 95–109. Springer, 2008.
- [88] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: reliable stream computation in the cloud. In *EuroSys*, 2013.
- [89] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106, 2002.
- [90] Muruhan Rathinam and Linda R. Petzold. A new look at proper orthogonal decomposition. *SIAM J. Num. Anal.*, 41(5):1893–1925, 2003.
- [91] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11), 2011.
- [92] Daniel M. Romero, Brendan Meeder, and Jon Kleinberg. Differences in the mechanics of information diffusion across topics: Idioms, political hashtags, and complex contagion on twitter. In *WWW*, 2011.
- [93] Ryan A. Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. Modeling dynamic behavior in large evolving graphs. In *WSDM*, 2013.
- [94] Maayan Roth, Assaf Ben-David, David Deutscher, Guy Flysher, Ilan Horn, Ari Leichtberg, Naty Leiser, Yossi Matias, and Ron Merom. Suggesting friends using the implicit social graph. In *KDD*, 2010.
- [95] Conrad Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, Australia, October 2010.
- [96] W. Schilders. *Model Order Reduction: Theory, Research Aspects and Applications*, volume 13 of *Mathematics in Industry*. Springer, Berlin, 2008.
- [97] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.
- [98] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

- [99] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multi-level Methods for Elliptic Partial Difference Equations*. Cambridge University Press, 1996.
- [100] Daniel Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *SIAM J. Comput.*, 42(1):1–26, 2013.
- [101] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.
- [102] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/Collect: Graph algorithms for the (semantic) web. In *ISWC*, 2010.
- [103] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Ar-netMiner: extraction and mining of academic social networks. In *KDD*, 2008.
- [104] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From “think like a vertex” to “think like a graph”. *PVLDB*, 7(3), 2013.
- [105] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.
- [106] Ramakrishna Varadarajan, Vagelis Hristidis, and Louiqa Raschid. Explaining and reformulating authority flow queries. In *ICDE*, 2008.
- [107] Ramakrishna Varadarajan, Vagelis Hristidis, Louiqa Raschid, Maria-Esther Vidal, Luis Daniel Ibáñez, and Héctor Rodríguez-Drumond. Flexible and efficient querying and ranking on hyperlinked data sources. In *EDBT*, 2009.
- [108] A.Y. Volkova. A refinement of the central limit theorem for sums of independent random indicators. *Theory Probab. Appl.*, 40(4):791–794, 1996.
- [109] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, 2005.
- [110] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

- [111] Apache Giraph. `giraph.apache.org`.
- [112] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. TwitterRank: finding topic-sensitive influential twitterers. In *WSDM*, 2010.
- [113] David Wingate, Nathaniel Powell, Quinn Snell, and Kevin Seppi. Prioritized multiplicative Schwarz procedures for solving linear systems. In *IPDPS*, 2005.
- [114] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14), 2013.
- [115] Wenpu Xing and Ali Ghorbani. Weighted PageRank algorithm. In *CNSR*, 2004.
- [116] Philip S. Yu, Jiawei Han, and Christos Faloutsos, editors. *Link Mining: Models, Algorithms, and Applications*. Springer, 2010.
- [117] Philip S. Yu, Xin Li, and Bing Liu. On the temporal dimension of search. In *WWW Alt*, 2004.
- [118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
- [119] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.
- [120] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. PrIter: A distributed framework for prioritized iterative computations. In *SOCC*, 2011.
- [121] Hongkai Zhao. Parallel implementation of fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007.
- [122] Li Zheng, Chao Shen, Liang Tang, Tao Li, Steve Luis, and Shu-Ching Chen. Applying data mining techniques to address disaster information management challenges on mobile devices. In *KDD*, 2011.