Big Graph Analytics Platforms

Big Graph Analytics Platforms

Da Yan
The University of Alabama at Birmingham
yanda@uab.edu

Yingyi Bu
Couchbase, Inc.
yingyi@couchbase.com

Yuanyuan Tian
IBM Almaden Research Center, USA
ytian@us.ibm.com

Amol Deshpande
University of Maryland
amol@cs.umd.edu

now
the essence of knowledge
Boston — Delft

Foundations and Trends® in Databases

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

D. Yan, Y. Bu, Y. Tian, and A. Deshpande. *Big Graph Analytics Platforms.* Foundations and Trends® in Databases, vol. 7, no. 1-2, pp. 1–195, 2014.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-242-6

Foundations and Trends® in Databases

Volume 7, Issue 1-2, 2014

Editorial Board

Editor-in-Chief

Editors

Editorial Scope

Topics

Foundations and Trends® in Databases covers a breadth of topics relating to the management of large volumes of data. The journal targets the full scope of issues in data management, from theoretical foundations, to languages and modeling, to algorithms, system architecture, and applications. The list of topics below illustrates some of the intended coverage, though it is by no means exhaustive:

- Data models and query languages
- Query processing and optimization
- Storage, access methods, and indexing
- Transaction management, concurrency control, and recovery
- Deductive databases
- Parallel and distributed database systems
- Database design and tuning
- Metadata management
- Object management
- Trigger processing and active databases
- Data mining and OLAP
- Approximate and interactive query processing
- Data warehousing
- Adaptive query processing
- Data stream management
- Search and query integration
- XML and semi-structured data
- Web services and middleware
- Data integration and exchange
- Private and secure data management
- Peer-to-peer, sensornet, and mobile data management
- Scientific and spatial data management
- Data brokering and publish/subscribe
- Data cleaning and information extraction
- Probabilistic data management

Information for Librarians

Foundations and Trends® in Databases, 2014, Volume 7, 4 issues. ISSN paper version 1931-7883. ISSN online version 1931-7891. Also available as a combined paper and online subscription.

Foundations and Trends® in Databases
Vol. 7, No. 1-2 (2014) 1–195

DOI: 10.1561/1900000056

Big Graph Analytics Platforms

Da Yan
The University of Alabama at Birmingham
yanda@uab.edu

Yingyi Bu
Couchbase, Inc.
yingyi@couchbase.com

Yuanyuan Tian
IBM Almaden Research Center, USA
ytian@us.ibm.com

Amol Deshpande
University of Maryland
amol@cs.umd.edu

Contents

Abstract

Due to the growing need to process large graph and network datasets created by modern applications, recent years have witnessed a surging interest in developing big graph platforms. Tens of such big graph systems have already been developed, but there lacks a systematic categorization and comparison of these systems. This article provides a timely and comprehensive survey of existing big graph systems, and summarizes their key ideas and technical contributions from various aspects. In addition to the popular vertex-centric systems which espouse a think-like-a-vertex paradigm for developing parallel graph applications, this survey also covers other programming and computation models, contrasts those against each other, and provides a vision for the future research on big graph analytics platforms. This survey aims to help readers get a systematic picture of the landscape of recent big graph systems, focusing not just on the systems themselves, but also on the key innovations and design philosophies underlying them.

D. Yan, Y. Bu, Y. Tian, and A. Deshpande. *Big Graph Analytics Platforms*.
Foundations and Trends® in Databases, vol. 7, no. 1-2, pp. 1–195, 2014.

DOI: 10.1561/1900000056.

1

Introduction

The growing need to deal with massive graphs in real-life applications has led to a surge in the development of big graph analytics platforms. Tens of big graph systems have already been developed, and more are expected to emerge in the near future. Researchers new to this young field can easily get overwhelmed and lost by the large amount of literature. Although several experimental studies have been conducted in recent years that compare the performance of several big graph systems [Lu et al., 2014, Han et al., 2014a, Satish et al., 2014, Guo et al., 2014], there lacks a comprehensive survey that clearly summarizes the key features and techniques developed in existing big graph systems. A recent survey [McCune et al., 2015] attempts to cover the landscape as well, but primarily focuses on vertex-centric systems; it omits most of the work on other programming models and also several crucial optimization and programmability issues with vertex-centric systems. In addition to describing the various systems, this survey puts more emphasis on the innovations and technical contributions of existing systems, in order to help readers quickly obtain a systemic view of the key ideas and concepts. We hope this will help big graph system researchers avoid reinventing the wheel, apply useful existing techniques to their own systems, and come up with new innovations.

In the rest of this chapter, we first review the history of research on Big Graph systems, and then overview some important features of existing Big Graph systems. Finally, we present the organization of this survey. Many contents of this survey are covered by our tutorial in SIGMOD 2016 [Yan et al., 2016a], the slides of which are available online[1] and contain animations to illustrate the various techniques used by existing systems.

1.1 History of Big Graph Systems Research

Although graph analytics has always been an important research topic throughout the history of computation, the research on *big* graph processing only flourished in recent years as part of the big data movement, which has seen increased use of advanced analytics on large volumes of unstructured or semi-structured data. A hallmark of this movement has been the MapReduce distributed data processing framework, introduced by Google [Dean and Ghemawat, 2004], and the companion Google File System (GFS) [Ghemawat et al., 2003]. Subsequently, the Apache Hadoop project[2] implemented the open-source counterparts, the Hadoop Distributed File System (HDFS) and the Hadoop MapReduce framework in 2006. Since then, a huge body of research has focused on designing novel MapReduce algorithms as well as on improving the framework for particular workloads. A large body of work in that space focused on big graph analytics, and many tailor-made MapReduce algorithms were proposed for solving specific graph problems [Lin and Schatz, 2010]. As an early MapReduce-based framework designed for general-purpose graph processing, PEGASUS [Kang et al., 2009] models graph computation by a generalization of matrix-vector multiplication. However, the reliance on the disk-based Hadoop MapReduce runtime, which requires repeated reads and writes of large files from HDFS, fundamentally limits its performance.

[1] `http://www.cse.cuhk.edu.hk/systems/gsys_tutorial/`

[2] `https://hadoop.apache.org/`

Later, Malewicz et al. [2010] proposed the Pregel framework specially designed for large-scale big graph processing. Since many graph algorithms are iterative, Pregel keeps the graph data in the main memory and adopts an iterative, message-passing computation model (inspired by the well-known Bulk Synchronous Parallel model for parallel computation), and is thus much more efficient than MapReduce. Pregel also adopts a "think-like-a-vertex" programming model which is more intuitive and user-friendly for average programmers and a natural fit for a range of graph analysis tasks. The vertex-centric programming model of Pregel is also very expressive since a vertex can communicate with any other vertex by passing messages. Since the introduction of Pregel, it has sparked a large number of research works on extending the basic Pregel framework in different aspects to improve the graph processing performance [Tian et al., 2013, Yan et al., 2014a, Zhang et al., 2014, Han and Daudjee, 2015, Yan et al., 2016b].

Independent of Pregel, Low et al. [2010] developed a multi-core, shared-memory graph-based computation model, called GraphLab. Then, Low et al. [2012] extended it to work in a distributed environment, while keeping the shared memory programming abstraction, in which a vertex can directly access the states of its adjacent vertices and edges. Later, GraphLab switched to the GAS (Gatter-Apply-Scatter) computation model to further improve the system performance [Gonzalez et al., 2012]. Although the GAS model covers a large number of graph algorithms, it is less expressive than the Pregel model, since a vertex can only access the data of its adjacent vertices and edges; we call this a *neighborhood-based shared memory* abstraction. This programming abstraction is especially popular among recent big graph systems designed to run on a single machine, such as GraphChi [Kyrola et al., 2012].

While Pregel and GraphLab are designed specially for graph processing, a number of systems, such as GraphX [Gonzalez et al., 2014] and Pregelix [Bu et al., 2014], rely on a general-purpose data processing engine for execution, at the same time providing graph-specific programming interfaces similar to those in Pregel and GraphLab.

Vertex-centric systems are ideally suited for graph analysis tasks like PageRank computation where the overall computation can be broken down into individual tasks, each involving a specific vertex (i.e., its local state, and the states of its adjacent edges). Many machine learning tasks (e.g., belief propagation, matrix factorization, stochastic gradient descent) are also a natural fit for those systems. However, many complex graph analysis tasks cannot be easily decomposed in such fashion. For example, a class of graph problems termed "ego-centric analysis" [Quamar et al., 2016] require analyzing the neighborhoods of the vertices in their entirety. Also, graph problems such as graph matching or graph mining may have intermediate or output results with size superlinear or even exponential in the input graph size. Complex graph algorithms, e.g., the Hungarian algorithm for maximum bipartite matching, even require random access to the entire graph. Solving these problems using vertex-centric processing leads to substantial communication and memory overheads, since each vertex needs to collect the relevant neighborhood subgraph (if not the entire graph) to its local state before processing the subgraph.

This has led to the development of many alternative programming frameworks, examples of which include Socialite [Seo et al., 2013b], Arabesque [Teixeira et al., 2015], NScale [Quamar et al., 2016], among others. In addition, several systems including Ligra [Shun and Blelloch, 2013], Galois [Nguyen et al., 2013], Green-Marl DSL [Hong et al., 2012], etc., provide low-level graph programming frameworks that can handle nearly arbitrary graph computations. These frameworks often focus on specific classes of graph problems, and make a range of different assumptions about the computing environment, making them incomparable in many cases. Arabesque tackles problems like graph matching and graph mining where the intermediate result can be very large, while assuming that the entire graph can be held in a single machine memory. NScale is a strict generalization of the vertex-centric framework, and can handle tasks that require access to multi-hop neighborhoods of vertices; but it does not support the other classes of problems discussed above. Socialite uses a Datalog-inspired programming model which is most suitable for graph problems that can be expressed as recursive

Datalog queries. Ligra, Galois, and other similar systems require random access to the graph and focus on large-memory multi-core environments. Thus, developing a sufficiently expressive, yet easy-to-use and easy-to-parallelize graph programming model, remains a critical and open challenge in this field.

The majority of existing big graph systems are designed for processing static graphs (or with small topology mutations). However, real-world graphs often evolve over time, with vertices and edges continually being added or deleted, and their attributes being frequently updated. A new class of big graph systems, such as KineoGraph [Cheng et al., 2012], TIDE [Xie et al., 2015b], DeltaGraph [Khurana and Deshpande, 2013], and Chronos [Han et al., 2014b], have emerged to process and analyze temporal and streaming graph data. This area is however still in its infancy and there are many open problems that need to be addressed to effectively handle continuous and/or temporal analytics on big graphs.

There is also a large body of work on executing queries related to a specific vertex (or a small subset of vertices) against large volumes of graph data, which has developed a range of specialized indexes and search algorithms. This survey does not cover that body of work.

1.2 Features of Big Graph Systems

We can categorize the big graph platforms along various dimensions. Since an important feature of the modern big graph systems is user-friendliness in programming parallel graph algorithms, we first summarize the programming abstractions (languages and models) of existing systems. While most systems adopt existing programming languages that are familiar to users (e.g., C/C++ and Java), some systems require users to learn a new domain-specific language dedicated to programming parallel graph algorithms (e.g., Green-Marl [Hong et al., 2012] and Trinity Specification Language [Shao et al., 2013]).

1.2.1 Programming Model

Most big graph systems adopt the vertex-centric model where a programmer only needs to specify the behavior of one vertex. The vertex-centric model can be further divided into two types: (1) message passing (e.g., in Pregel), where vertices communicate with each other by sending messages; and (2) shared-memory abstraction (e.g., in GraphLab), where vertices directly access the states of other vertices and edges.

Message passing is a natural model in a distributed environment, since users can explicitly dictate message passing behavior in their programs. In contrast, the shared-memory abstraction allows programmers to directly access data as if operating on a single machine, and most single-machine vertex-centric systems adopt this model. However, distributed GraphLab adopts the shared-memory abstraction and there are also single-machine systems that adopt message passing (e.g., FlashGraph [Zheng et al., 2015]).

The vertex-centric framework can be further extended with a block-centric model (e.g., Giraph++ [Tian et al., 2013] and Blogel [Yan et al., 2014a]), which partitions the vertices into multiple disjoint subgraphs, so that value propagation within each subgraph could bypass network communication. The block-centric model often improves the performance of graph computation by orders of magnitude.

Besides the vertex-centric systems, some big graph systems adopt a matrix-based programming model; these include PEGASUS [Kang et al., 2009], GBASE [Kang et al., 2011], and SystemML [Ghoting et al., 2011]. These systems represent a graph algorithm by a sequence of generalized matrix-vector multiplications, which can be efficiently processed since sparse matrix algebra has been studied for decades in the High Performance Computing (HPC) field. However, users who are not familiar with matrix algebra might prefer vertex-centric programming to matrix-based programming. Recently, Sundaram et al. [2015] helped bridge the gap for these users: their GraphMat system translates a vertex-centric program to high performance sparse matrix operations to be run on the backend.

Another important class of programming models is subgraph-centric models, where users write programs to process a subgraph instead of a single vertex. These models target graph problems whose output size can be exponential to the graph size (e.g., graph matching and finding motifs), or problems that require analyzing entire neighborhoods in a holistic manner. Since vertices in a subgraph can be randomly accessed by a user program, a critical issue for a subgraph-centric model is how to efficiently construct the relevant subgraphs. Arabesque [Teixeira et al., 2015] and NScale [Quamar et al., 2016] are two systems that use a subgraph-centric model, although there are significant differences in the models they adopt.

There are also systems that require users to write graph algorithms using a domain specific language (DSL), e.g., Green-Marl [Hong et al., 2012, 2014], Galois [Nguyen et al., 2013], and Ligra [Shun and Blelloch, 2013]. The language constructs of those DSLs expose opportunities for parallelism, which can be utilized by the system for efficient parallel execution. Of course, users have to learn a new language or programming paradigm in order to use such a system.

Finally, several recent systems have been built to bring in declarative query languages for big graph analytics. First, since many graph algorithms can be expressed as recursive Datalog [Bancilhon and Ramakrishnan, 1986] queries, a number of research projects are inventing new-generation Datalog systems for scalable big graph analytics. Second, often times, a graph analytics job is only one part of a gigantic, end-to-end SQL[3]-dominated data analysis pipeline which includes constructing graphs dynamically from tabular data sources and converting graph computation results back into tabular reports; therefore, several systems have integrated vertex-centric programming models into declarative query languages to make those end-to-end data analysis tasks easier [Simmen et al., 2014, Gonzalez et al., 2014].

1.2.2 Expressiveness

Most big graph systems aim at solving a broad class of graph problems using a unified programming framework. Therefore, it is meaningless to study big graph systems without studying the algorithms and applications that can be implemented in these systems. However, many

[3] SQL. `https://en.wikipedia.org/wiki/SQL`

papers just introduce API simplicity and performance advantages of their systems in order to promote their work, but these benefits may come at a cost of additional assumptions and narrower expressiveness that were understated, which should be made clear to avoid blind or even wrong system choice. We now discuss the expressiveness of the various programming models described before, and provide some advice on how to choose an appropriate framework for an application at hand.

Many graph algorithms only require each vertex to communicate with its neighbors, such as PageRank and other more complicated random walk algorithms (e.g., [Zhang et al., 2016]). In these algorithms, intermediate data are only exchanged along edges, and so the volume of intermediate data is comparable to the data size. We say that these algorithms require *edge-based communication.* In some of these algorithms, a vertex only needs the aggregated value of the received values, which provides opportunities for further optimization. For example, MOCgraph [Zhou et al., 2014], GraphD [Yan et al., 2016d], and the superstep-splitting technique of Giraph [Ching et al., 2015] all propose aggregating messages earlier instead of buffering them for later processing, in order to save memory space; while PowerGraph [Gonzalez et al., 2012], GraphChi [Kyrola et al., 2012] and X-Stream [Roy et al., 2013] assume that data values are aggregated at each vertex from its incoming edges, in their model design. We, however, would like to indicate that not all algorithms with edge-based communication allow its vertices to aggregate received values, such as the attribute broadcast algorithm of Yan et al. [2015].

Edge-based communication implies that any information can be propagated for just one hop at a time, which leads to poor performance if a vertex u needs to transmit a value to another vertex v far away from u in a large-diameter graph. Pointer jumping (aka path doubling), a technique from PRAM algorithm design, solves this problem by doubling the propagation length from u to v, until v is reached. This requires a vertex to be able to send data to any other vertex, not just its neighbors. We say that these algorithms require *ID-based communication*, where a vertex u can send messages to another vertex w as long as w's ID is known. Pregel [Malewicz et al., 2010] adopts ID-based communication and thus can implement pointer-jumping algorithms such as those to be described in Chapter 3.2, while GraphLab [Gonzalez et al., 2012] only allows each vertex to access its neighbors' data, and thus cannot support these algorithms. In fact, Pregel has probably the most expressive programming model in theory, and it is known how to write a large number of graph algorithms efficiently in that model. The Bulk Synchronous Parallel (BSP) model, on which Pregel is based, has been very well-studied, but as a synchronous model, the number of iterations must be kept low in a distributed setting, which can be achieved with the help of pointer jumping.

Another solution to avoid slow value propagation is to use a block-centric model, where nearby vertices are grouped into a block for processing together each time. In a distributed environment, since a block is assigned to a unique machine, only blocks need to communicate with each other, and computation over vertices inside a block does not generate communication. In a single-machine environment, each block usually fits in a CPU cache, and thus block-based processing improves cache locality in its execution. In addition to faster value propagation (i.e., block-wise), the block-centric model also significantly reduces the communication workload. Representative block-centric system include Giraph++ [Tian et al., 2013] and Blogel [Yan et al., 2014a].

Some graph algorithms (e.g., k-core finding [Salihoglu and Widom, 2014]) need to mutate the graph topology during computation, and thus, support for deletion and addition of edges and vertices is also an important aspect of system expressiveness. For example, VENUS [Cheng et al., 2015] streams immutable graph structure and thus does not support algorithms that require graph mutations.

The models discussed so far are mostly vertex-centric. However, many graph mining problems define constraints on subgraphs, e.g., graph matching and motif mining. Subgraph-based models are proposed to solve these problems by writing user-friendly programs, where computation is directly performed on subgraphs. Such systems include NScale [Quamar et al., 2016] and Arabesque [Teixeira et al., 2015], which we discuss in more detail in Chapter 7.

We remark that there are other models that could be more appropriate for a specific application at hand. For example, if one is viewing a graph as a matrix, and solving a machine learning problem that uses matrix operations, then matrix-based systems like SystemML [Ghoting et al., 2011] could be a better choice. Also, if graph processing is just part of a dataflow program, then dataflow-based systems could provide more flexibility, e.g., GraphX [Gonzalez et al., 2014] can interoperate with other dataflow operators in Spark [Zaharia et al., 2012] to avoid data import/export.

1.2.3 Execution Mode

Most big graph systems target iterative graph computation, where vertex values are repeatedly updated until the computation converges. There are two typical execution modes: synchronous and asynchronous. The synchronous mode is also called bulk synchronous parallel (BSP), exemplified by Pregel, while the asynchronous mode is adopted by GraphLab and several other systems (especially those targeting machine learning workloads). The difference between these two modes is that, in the synchronous mode, there is a global barrier from one iteration to another, and out-going messages or updates of one iteration are only accessible in the next iteration; in the asynchronous mode, a vertex has immediate access to its in-bound messages or updates.

Asynchronous parallel computation incurs race conditions and thus requires additional effort to enforce data consistency (e.g., by using locks). Moreover, in a distributed environment, asynchronous execution tends to transmit a lot of small messages, since the update to a vertex value should be reflected in time. In contrast, BSP only requires updates to be synchronized at the end of each iteration, and messages can be sent in large batches. In fact, GraphLab has a synchronous mode that simulates the BSP mode of Pregel, and both Lu et al. [2014] and Han et al. [2014a] found that the synchronous mode is generally faster than the asynchronous mode. Further, for many algorithms, asynchronous execution is not an option because the indeterministic execution may lead to incorrect answers.

However, for some problems like PageRank computation, vertex values converge asymmetrically: most vertices converge quickly after a few iterations, but some vertices take a large number of iterations to converge. In that case, asynchronous execution can schedule those vertices that converge more slowly to compute for more iterations, while synchronous execution processes every vertex once in each iteration even if most vertices are converged. Therefore, asynchronous mode is much faster for such algorithms and is thus preferred. Moreover, asynchronous computation is always preferred in a single-machine system since data access no longer incurs network communication, and accessing the latest vertex value leads to faster convergence.

It is, however, worth noting that some asynchronous frameworks may not converge to the exact results (e.g., PageRank values), but the approximate results are often good enough while the significant improvement in performance (compared with synchronous execution) is highly attractive. More discussion can be found in Section 4.2.

Recently, PowerSwitch [Xie et al., 2015a] showed how to support mode switch between asynchronous execution and synchronous execution in GraphLab. They found that when the workload is low, asynchronous execution is faster due to the faster convergence rate provided by accessing the latest values. Race conditions (e.g., updates to the same vertex) are unlikely to occur since only a small portion of vertices participate in computation, and the number of messages is too small to benefit from sending in large batches. In contrast, when the workload is high, synchronous execution is faster since there is no need to handle race conditions (i.e., it avoids the expensive locking/unlocking cost required by asynchronous execution), and messages are sent in large batches. Thus, PowerSwitch constantly collects execution statistics on-the-fly, which are used to predict future performance and determine the timing of a profitable mode switch.

1.2.4 Other Features

There are also many other dimensions to categorize big graph systems. As for the **execution environment**, there are systems developed to process graphs in a single machine, or using a cluster of machines.

The single-machine environment can be further divided into two types, commodity PCs and high-end servers. The former targets processing big graphs efficiently using readily available resources; since the available memory on a commodity PC is limited, the graph is usually disk-resident, and loaded into memory for processing part-by-part or in a streaming fashion. The latter aims at beating distributed systems by eliminating the cost of network communication, and the graph is usually memory-resident. As for the **graph placement**, distributed systems usually keep the graph in main memory, since there are many machines and the total RAM size is sufficient, while single-PC systems tend to process disk-resident or SSD-resident graphs. There are also distributed systems that process disk-resident graphs in order to scale to giant graphs whose size is much larger than the total RAM size in a cluster, such as Pregelix [Bu et al., 2014], GraphD [Yan et al., 2016d] and Chaos [Roy et al., 2015].

There are also many design techniques that may significantly influence the system performance for specific algorithms. For example, disk-based single-machine systems like GraphChi [Kyrola et al., 2012] are designed for iterative batch processing, while TurboGraph [Han et al., 2013] maintains an in-memory page ID table for directly locating the disk page of any vertex. Given these differences in system design, a reader will not be surprised to see a claim like TurboGraph "significantly outperforms GraphChi by up to four orders of magnitude", for a query that is to find the neighbors of a particular vertex.

1.3 Organization of the Survey

The diverse features supported by different big graph systems, and the cross-cutting nature of many of the key designs, make it challenging to organize such a survey. As an example, the popular vertex-centric programming model is easy to support on top of a wide range of different underlying implementations, including distributed frameworks like Hadoop MapReduce, matrix-based systems, and relational databases. However, each of those implementations raises unique and different challenges despite their use of the vertex-centric model on top.

In this survey, we focus on presenting the key designs and features of the various graph processing systems, while endeavoring to place related systems together to summarize the common ideas underlying their designs. For quick reference, Table 1.1 presents a list of all the systems that we discuss in each chapter.

We broadly divide the survey into three parts. In Part I, we discuss the big graph systems that primarily use the vertex-centric programming model, which has been widely studied recently due to its simplicity in programming parallel graph algorithms. Specifically, Chapter 3 reviews the framework of Pregel, and introduces how to develop algorithms with performance guarantees in Pregel; it then discusses existing open-source Pregel-like systems with improvements in communication mechanism, load balancing, out-of-core support and fault tolerance. Chapter 4 walks through the various extensions to the basic framework of Pregel that could significantly improve the performance of graph computations. Chapter 5 covers a few important big graph systems that adopt shared memory abstraction, including the pioneering GraphLab system.

In Part II, we review other systems that attempt to provide support for more general graph programming models; most of these are motivated by the observation that complex graph algorithms or analysis tasks are often difficult to program using the simple vertex-centric programming framework. Chapter 6 describes several matrix-based big graph systems, including the pioneering MapReduce-based systems PEGASUS and GBASE, and the more powerful SystemML system. Chapter 7 explains why the vertex-centric and matrix-based frameworks are not sufficient for graph problems like graph matching and motif mining, and introduces two subgraph-centric systems, NScale and Arabesque, to process such graph problems efficiently. Chapter 8 reviews several systems that either offer database-style declarative query languages or leverage database-style query processing techniques.

Finally, in Part III, we discuss some miscelleneous issues. While some vertex-centric single-machine big graph systems are also introduced in Chapter 5, Chapter 9 surveys more single-machine systems that adopt a computation model beyond a pure vertex-centric one.

Table 1.1

Section	System
3.1	Pregel
3.3	Giraph, Pregel+, GPS, MOCgraph
3.4	WindCatch, PAGE
3.5	GraphD
4.1	Giraph++, Blogel
4.2	Maiter, GiraphUC
4.3	Quegel
5.1	GraphLab/PowerGraph
5.2	GraphChi, X-Stream, Chaos, VENUS, GridGraph
6.1	PEGASUS
6.2	GBASE
6.2	SystemML
7.1.1	Trinity
7.2	NScale
7.3	Arabesque
8.1	SociaLite, DeALS, Myria, Yedalog
8.2	GraphX, Pregelix, Vertexica
8.3	REX, Maiter
8.4	Aster Data
9.1	GraphMat, GraphTwist
9.2	Green-Marl, Ligra, GRACE, Galois
10.1	TurboGraph, FlashGraph
10.2	Medusa, MapGraph, CuSha
11.2	Chronos, DeltaGraph, LLAMA
11.3	Kineograph, TIDE

Chapter 10 discusses a few systems that utilize new hardware technologies to significantly boost the performance of big graph analytics. Then, in Chapter 11, we discuss the issues of managing time-evolving graphs and supporting real-time analytics over streaming graph data, and discuss several recent systems that focus on providing those capabilities. Finally, we conclude the survey in Chapter 12 and provide a discussion on future research in big graph analytics platforms.

2

Preliminaries

In this chapter, we present the architectures of typical single-machine and distributed graph analytics systems, and briefly discuss some of the key considerations; we also discuss some of the techniques used to improve the performance and robustness of such systems.

2.1 Data Models and Analytics Tasks

We begin with introducing some of the notation used throughout this survey. In this survey, we consider an input graph $G = (V, E)$ where each vertex $v \in V$ has a unique ID $id(v)$. For simplicity, we use v and $id(v)$ interchangeably. The number of vertices and edges are denoted by $|V|$ and $|E|$, respectively. If G is undirected, we denote the set of neighbors of v by $\Gamma(v)$, and denote the degree of v by $d(v)$. If G is directed, we denote the set of in-neighbors (and out-neighbors) of v by $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$, and denote the in-degree (and out-degree) of v by $d_{in}v$ (and $d_{out}(v)$). The graph diameter is denoted by δ. Both vertices and edges may have attributes associated with them, such as edge weights or text labels.

A graph can be equivalently seen as a $|V| \times |V|$ *adjacency matrix*, where each non-zero matrix entry implies the existence of an edge between the corresponding nodes in the graph. This equivalence allows using matrix operations and libraries to execute graph analytics tasks [Kepner and Gilbert, 2011].

This survey reviews general-purpose systems for running a graph analytics task that takes a graph as input, and produces an output which can be: (a) one or a small number of scalar values (e.g., global clustering coefficient of the graph), (b) a set of values, one per node/edge (e.g., PageRank of each node in the graph), (c) a set of values, one for each instance of a subgraph in the graph, with subgraphs of interest specified in some fashion (e.g., finding maximal cliques), or (d) other forms such as in graph summarization and deduplication. Graph analysis tasks of Category (b) are perhaps the most prevalent, and certainly the most well-studied from systems perspective. Note that, different analysis tasks from this class may exhibit very different computational complexity; e.g., PageRank and *betweenness centrality* are both centrality measures whose computation falls in this class, but while PageRank computation is relatively easy to parallelize and execute, betweenness centrality is much harder to compute.

We can make a further distinction between one-shot analytics and continuous analytics. In the latter case, the input graph itself is dynamically changing over time, and the goal is to execute the analytics task continuously or periodically. Most of the work in big graph analytics systems has focused on one-shot analytics; we discuss the work on continuous analytics towards the end of the survey.

Finally, an important consideration is how the graph is physically represented, and where it is stored. Most of the systems that we cover assume that the graph is stored on secondary storages (either a distributed file system like HDFS, or a local file system), in either human-readable text format or serialized binary format. Some systems expect the graph to be provided as a list of edges, whereas others may require an adjacency list representation, with all the edges of a vertex listed along with the vertex information. Many of the systems can also ingest data from other data stores like key-value stores, or relational

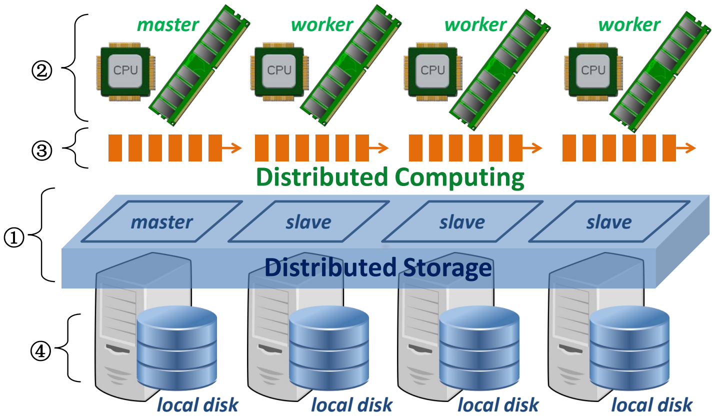


Figure 2.1: Components of a Distributed System

databases. In many cases, the graph may itself have to be generated by processing other non-graph datasets [Xirogiannopoulos et al., 2015]; although this is an important practical consideration, there is little systematic work on it so far and we do not discuss it further in this survey.

2.2 Distributed Architecture

Next we discuss the components of a typical distributed graph analytics system, and some of the key decisions that significantly impact the performance of such a system.

Components. The typical architecture of a distributed graph analytics system is shown in Figure 2.1, which consists of four components.

Component ① is an underlying distributed file storage for keeping graph datasets and analytics results, which is usually resilient to machine failures. Most graph analytics systems such as Giraph [Ching et al., 2015] and GraphLab [Gonzalez et al., 2012] support loading data from HDFS, which by default replicates each data block on three machines so that the failure of any two machines will not cause data loss. Besides HDFS, the storage component can also be other distributed key-value stores or databases, such as HBase[1] and Cassandra[2]. There also exist distributed systems designed with their own storage subsystems, e.g., Chaos [Roy et al., 2015], but this loses the interoperability among various Big Data frameworks, and also incurs expensive preprocessing cost of data conversion.

Component ② is the computation module that loads a graph dataset from the underlying distributed storage, and performs the actual computation. The key design issue of this component is how to expose a user-friendly programming interface to users, hiding low-level details of parallel computation. Various computation models have been designed as we survey in depth later.

Component ③ is the communication layer that is used by the computation module for communication between machines. The goal of this component is to expose a simple and clean communication interface to the computation module, in order to simplify the design of the computation module. This component does not directly interact with application developers (users), since a user-friendly programming interface should hide details like network communication from the users. Different systems use different communication mechanisms. For example, Giraph [Ching et al., 2015], Pregel+ [Yan et al., 2015], and GraphLab [Gonzalez et al., 2012] use Netty[3], MPI and Remote Procedure Call (RPC), respectively.

Component ④ is local disk storage, which is managed by the respective machines individually, to offload some data from memory to local disk(s) during the computation. This is usually used by out-of-core systems (or modes) such as GraphD [Yan et al., 2016d], Pregelix [Bu et al., 2014], and out-of-core Giraph. Some systems even offload data to HDFS and only exchange control information among the machines, an example being epiC [Jiang et al., 2014], but this may not be a good solution since data offloading incurs network communication which is often more expensive than local disk IO (e.g., HDFS always replicates data to other machines for fault tolerance).

[1] http://hbase.apache.org/

[2] http://cassandra.apache.org/

[3] http://netty.io/

Graph Partitioning. To execute the graph analysis task in a parallel fashion, the graph must be *partitioned* (*sharded*) across the machines in the cluster. The graph analytics system typically creates a set of partitions using some policy, which are then transparently mapped to the actual machines by the underlying framework (e.g., Yarn or Mesos); some systems explicitly "over-partition" to simplify load-balancing, in which case multiple logical partitions are mapped to the same physical machine. However, we often talk about mapping of a vertex to a machine to simplify discussion.

Most systems partition the graph by vertices, i.e., each partition consists of a set of vertices, all the properties of those vertices, and the outgoing edges for those vertices. A straightforward method is to partition the vertices by their IDs. For example, hash-based partitioning computes the partition for a vertex using a hash function that takes the vertex ID as input, while range-based partitioning divides vertices into different partitions by ID ranges. The benefit of this method is that we can compute the partition of a vertex (for sending messages to that partition) in $O(1)$ time. Optionally, one may create partitions with the goal of reducing the number of edges that are cut (and thus the total communication during analysis); however, this comes with the additional cost of having to run an expensive graph partitioning algorithm. This approach also makes it more costly to find the location of a vertex, and typically a distributed hash table needs to be used for this purpose incurring additional communication (and thus round-trip delay) [Shang and Yu, 2013, Khayyat et al., 2013]; one may also recode IDs in a preprocessing step like in Giraph++ [Tian et al., 2013]. Shao et al. [2015] even observed that Giraph exhibits worse performance when a better partitioning is provided, as there are insufficient processors to process local messages. Maintaining such partitions in presence of dynamic updates to the graph is quite tricky, and typically this option is not used in such cases.

Instead of partitioning by vertices, several systems advocate partitioning by edges instead [Gonzalez et al., 2012]. In such cases, the edges of a specific vertex v may be distributed to multiple machines, each also keeping a replica of v's data. This approach often leads to a more balanced workload across the machines, especially given that many graphs (e.g., social networks) obey power-law degree distribution, resulting in a few very high-degree vertices.

Synchronous vs Asynchronous Execution. Another issue is whether the execution is synchronous or asynchronous. Consider PageRank computation, where each vertex keeps collecting values from in-neighbors and distributing values to out-neighbors until its PageRank converges. Since different vertices have different convergence rates, asynchronous execution allows those vertices that converge slowly (resp., quickly) to run for more (resp., fewer) rounds, and is thus more efficient. On the other hand, synchronous execution has more deterministic behavior, and it is easier to *batch* messages there to reduce the overall communication cost. Usually asynchronous execution is managed by a (distributed) prioritized scheduler that allows prioritized execution, while guaranteeing sufficient parallelism. One method to determine priority is delta propagation, where vertices propagate values (or deltas) to neighbors according to the incremental values (or deltas) it receives, and a vertex with a large delta has a higher priority of execution [Zhang et al., 2014]. In contrast, to perform breadth-first search (BFS), synchronous execution where the i-th round activates the vertices i hops away from the source vertex guarantees minimum workload. Synchronous execution is also easy to analyze and debug due to its deterministic behavior, and avoids race conditions and thus the locking/unlocking cost. To sum up, synchronous execution is a desirable choice unless the algorithm has an asymmetric convergence behavior.

Fault Tolerance. The last issue we would like to mention is fault tolerance. Although the underlying distributed storage (e.g., HDFS) already provides resilience to data loss, if a long-running job fails, reloading data and computing from scratch wastes all the previous computation. Checkpointing provides a trade-off between (1) the failure-free cost paid for being fault-tolerant, and (2) fast recovery. During synchronous execution, all machines can periodically suspend their computation at synchronization barriers, to write the current state of computation to the resilient distributed storage. If machine failure happens later, the latest stored computation state can be loaded to continue execution (rather than starting from scratch). This method is called coordinated checkpointing. There also exist uncoordinated checkpointing methods [Chandy and Lamport, 1985] that do not need to suspend execution, which is ideal for asynchronous execution. For algorithms that always converge to a fixed point, checkpointing can be avoided and recovery can be done by re-initializing the states of the lost data [Schelter et al., 2013].

2.3 Single-Machine Architecture

A single-machine graph analytics system is designed for running with a single machine, thus avoiding the expensive network communication overhead; however, such a system cannot scale out and thus has fixed hardware resources. As a result, the running time is usually proportional to the graph size. There are two types of single-machine systems: (1) for commodity PCs with limited resources (especially memory), and (2) for servers with many cores and a large memory.

For simplicity, let us assume that the computation is performed at the granularity of vertices. A typical single-machine system will partition vertices into multiple vertex shards V_1, V_2, …, V_n, and each vertex shard V_i is associated with an edge shard E_i, which keeps the edges (e.g., adjacency lists) of vertices in V_i. For Type (1) systems, partitioning is performed so that each shard can be loaded into memory for computation at a time. Some systems like GraphChi [Kyrola et al., 2012] load both V_i and E_i into memory for processing, while others such as X-Stream [Roy et al., 2013] and VENUS [Cheng et al., 2015] load only V_i into memory and stream E_i from disk. For Type (2) systems, partitioning provides sufficient parallelism so that each shard is processed by one core. Some of the latter systems may not even create explicit shards, but rather load the entire graph into the shared memory, and parallelize in an adaptive manner based on the specific computation being performed.

Compared with a distributed architecture, a single-machine system may need to preprocess the graph by sharding, which may not pay off if only one light-workload job needs to be run subsequently.

In fact, most such systems simply use a straightforward range-based vertex partitioning [Kyrola et al., 2012, Roy et al., 2013, Cheng et al., 2015], rather than more sophisticated ones for sharding efficiency. A Type (2) single-machine system also requires one machine to load the entire graph, which can be much more costly than the parallel loading in a distributed system.

In general, we observe that a single-machine system can be more efficient for small graphs and computationally-light jobs, since there is no communication overhead. After all, distributed execution incurs round-trip delay, the cost of which could be already more than the actual computation. In contrast, a distributed system is more appropriate for big graphs and computationally-heavy jobs, since the scale-out solution requires each machine to process only a portion of the entire graph.

To better reason about efficiency of single-machine or distributed systems, McSherry et al. [2015] proposed a metric called COST (Configuration that Outperforms a Single Thread). For a given system and for a given problem, the COST is the hardware configuration (measured in the number of cores) required before the system out-performs a competent single-threaded implementation (which may be able to use a better, but non-parallelizable algorithm). Thus, the metric weighs the scalability of a system against the overheads introduced by the system to achieve that scalability. Their studies of popular systems like Giraph, GraphX and GraphLab show that those systems have very high COSTs, and often seem to perform worse that a single-threaded implementation despite using many more cores.

We caution that COST should only be used as one of the criteria to compare the pros and cons of different systems, and as a sanity check on the overheads. A single-threaded implementation is fundamentally limited in its ability to exploit parallelism. Further, for many graph analysis tasks, an easy-to-use programming framework that interacts with other widely used frameworks (like key-value stores, HDFS, etc.), is perhaps a more important criteria than pure performance.

Part I

Vertex-Centric Programming Model

3

Vertex-Centric Message Passing (Pregel-like) Systems

In this chapter, we first review the framework of Pregel in Section 3.1, and introduce how to develop Pregel algorithms with performance guarantees in Section 3.2. We then introduce the existing Pregel-like systems with improvements in communication mechanism (Section 3.3), load balancing (Section 3.4), out-of-core support (Section 3.5) and fault tolerance (Section 3.6).

3.1 The Framework of Pregel

Computation and Programming Model. A Pregel job starts by loading the input graph G from GFS (Google File System) into the main memories of the worker machines (or workers) in a cluster, where vertices are partitioned among the workers. Typically, vertices are partitioned by a hash function $hash(.)$ known by all workers: each vertex v is assigned to a worker $W = hash(v)$. Each vertex v maintains its adjacency list (which usually stores $\Gamma(v)$ or $\Gamma_{out}(v)$), a vertex value $a(v)$, and also a flag $active(v)$ indicating whether v is active or halted.

A Pregel job proceeds in iterations, where an iteration is also called a superstep. In Pregel, a user needs to specify a user-defined function (UDF) *compute*(*msgs*) to be called by a vertex v, where *msgs* is the set of incoming messages sent to v in the previous superstep. In v.*compute*(.), v may update $a(v)$, send messages to other vertices, and vote to halt (i.e., deactivate itself). Only active vertices will call *compute*(.) in a superstep, but a halted vertex will be reactivated if it receives a message. The program terminates when all vertices are halted and there is no pending message for the next superstep. Finally, the results are dumped to GFS.

We now illustrate how to write *compute*(.), using two graph algorithms. In these algorithms, a vertex only sends messages to its neighbors (or out-neighbors), whose IDs are directly available in its adjacency list. In Section 3.2, we will see some Pregel algorithms where a vertex sends messages to non-neighbors.

Example 1: PageRank. Consider the PageRank algorithm of [Malewicz et al., 2010] where $a(v)$ stores the PageRank value of vertex v, and $a(v)$ gets updated until convergence. In Step 1, each vertex v initializes $a(v) = 1/|V|$ and distributes $a(v)$ to its out-neighbors by sending each out-neighbor a message $a(v)/d_{out}(v)$. In Step i ($i > 1$), each vertex v sums up the received message values, denoted by *sum*, and computes $a(v) = 0.15/|V| + 0.85 \cdot sum$. It then distributes $a(v)/d_{out}(v)$ to each of its out-neighbors. If we want to perform PageRank computation for n supersteps, we can let every vertex vote to halt and exit *compute*(.) in Step $(n + 1)$.

Example 2: Hash-Min. We consider the Hash-Min algorithm of [Yan et al., 2014b] for computing the connected components (CCs) of an undirected graph G. Given a CC C, we denote the set of vertices of C by $V(C)$, and define the ID of a CC C to be $cc(v) = \min\{id(u) : u \in V(C)\}$. Hash-Min computes $cc(v)$ for each vertex $v \in V$. The idea is to broadcast the smallest vertex ID seen so far by each vertex v, which is stored in $a(v)$. In Step 1, each vertex v sets $a(v)$ to be the smallest ID among $id(v)$ and $id(u)$ of all $u \in \Gamma(v)$, broadcasts $a(v)$ to all its neighbors, and votes to halt. In Step i ($i > 1$), each vertex v receives messages (if any) from its neighbors; let *min* be the smallest ID received, if $min < a(v)$, v sets $a(v) = min$ and broadcasts $a(v)$ to its neighbors. All vertices vote to halt at the end of a superstep. When the process converges, $a(v) = cc(v)$ for all v.

We call the case where most vertices participate in the computation as **dense vertex access**, and the case where only a small fraction of vertices participate in the computation as **sparse vertex access**. Every superstep of the PageRank algorithm is dense, while for Hash-Min, earlier supersteps are dense while later supersteps are sparse.

Combiner. To reduce the number of messages transferred though the network, users may implement a message combiner to specify how to combine messages targeted at the same vertex v_t, so that messages on a worker W targeted at v_t will be combined into a single message by W locally, and then sent to v_t. In the PageRank (resp. Hash-Min) algorithm, the combiner can be implemented as the summation (resp. minimum) operation, since only the summation (resp. minimum) of incoming messages is of interest in *compute*(.).

Aggregator. Pregel also allows users to implement an aggregator for global communication. Each vertex can provide a value to an aggregator in *compute*(.) in a superstep. The system aggregates those values and makes the aggregated result available to all vertices in the next superstep. In the actual implementation, the values are first aggregated locally on each worker; then, the aggregated values are then aggregated globally at the master; the globally aggregated value is then broadcast back to all workers.

Fault Tolerance. To be fault tolerant, a Pregel job may be specified to periodically back up the state of computation to GFS at superstep boundaries as a checkpoint (e.g., every 10 supersteps). Since data on GFS is replicated on multiple machines, a checkpoint is resilient to machine failures. If a failure happens, all workers simply reload the state of computation from the latest checkpoint and then continue computation from the last checkpointed superstep.

Graph Mutations. Pregel also supports graph mutations, which are categorized into two types: (1) local mutations, where a vertex adds or removes its own edges or removes itself; and (2) global mutations, where a vertex adds or removes (i) the edges of other vertices or (ii) other vertices. Global mutations may incur conflicts and users may specify conflict resolution policies to avoid nondeterministic behavior. In a superstep, edge removals are performed first, and then vertex removals, followed by vertex addition, and finally edge addition.

Graph Mutations are useful in some graph algorithms, such as the algorithm for k-core finding [Quick et al., 2012] which finds the maximal subgraphs of an undirected graph G in which every vertex has a degree of at least k. In each superstep, the algorithm lets each vertex whose degree is less than k delete itself and its adjacent edges, until all the remaining vertices have degree at least k.

3.2 Algorithm Design in Pregel

Although there is an abundance of papers studying system improvements to Pregel, the number of papers studying algorithm design in Pregel is still very limited. However, as a general-purpose graph processing framework, it is important to study how to design scalable algorithms on top of Pregel.

There are a few papers studying Pregel algorithms: [Quick et al., 2012] demonstrated that many social network analytic tasks can be formulated as Pregel algorithms, while [Salihoglu and Widom, 2014] proposed four algorithm-specific techniques to improve the performance of some Pregel algorithms. However, these algorithms are still ad-hoc and there lacks any cost analysis.

We remark that it is important for users to be aware of the scalability of a Pregel algorithm. For example, in the triangle finding algorithm of [Salihoglu and Widom, 2014], assuming that $v_2, v_3 \in \Gamma(v_1)$; then, to determine whether a triangle $\triangle v_1 v_2 v_3$ exists, vertex v_1 sends a message to v_2 inquiring whether $v_3 \in \Gamma(v_2)$. Since there are $O(|E|^{\frac{3}{2}})$ triangles in a graph G, the number of messages generated in a superstep can be much larger than the graph size, leading to long-running supersteps or even memory overflow (note that Pregel buffers messages in main memories to be processed by *compute*(.)). If a user is aware of this scalability issue, he/she may simply design the algorithm to send inquiries for only a small subset of vertices in each superstep, so as to avoid memory overflow.

Practical Pregel Algorithm (PPA). [Yan et al., 2014b] identified a class of Pregel algorithms that have good scalability, called practical Pregel algorithms (PPAs). Specifically, a Pregel algorithm is called a **balanced practical Pregel algorithm** (**BPPA**) if it satisfies the following constraints:

1. *Linear space usage:* each vertex v uses $O(d(v))$, or $O(d_{in}(v) + d_{out}(v))$, memory space.
2. *Linear computation cost:* the time complexity of $v.compute(.)$ is $O(d(v))$, or $O(d_{in}(v) + d_{out}(v))$.
3. *Linear communication cost:* at each superstep, the volume of the messages sent/received by each vertex v is $O(d(v))$, or $O(d_{in}(v)+d_{out}(v))$.
4. *At most logarithmic number of rounds:* the algorithm terminates after $O(\log |V|)$ supersteps.

Constraints 1-3 offers good load balancing and linear cost at each superstep, while Constraint 4 controls the total running time (note that each superstep requires a global barrier at the end, which incurs some cost). For some graph problems, the vertex-grained requirements of BPPA can be too strict, and we can only achieve *overall linear space usage, computation cost, and communication cost* (still in $O(\log |V|)$ rounds). For example, in the S-V algorithm to be discussed at the end of this section, some vertices may send many more messages than their degrees, but the total number of messages is still linear to the graph size. We call a Pregel algorithm that satisfies these constraints simply as a **practical Pregel algorithm** (**PPA**).

The Hash-Min algorithm presented in Section 3.1 is not a PPA, since it takes $O(\delta)$ supersteps. For example, in a path graph where the smallest vertex is on one end of the path, Hash-Min runs for $|E|$ supersteps, the cost of which is prohibitive since each superstep requires message synchronization. To achieve the $O(\log |V|)$-superstep bound, it is usually necessary to use the pointer jumping (a.k.a. path doubling) technique where a vertex needs to send messages to non-neighbors. Pointer jumping is used in many Pregel algorithms, such as the S-V algorithm for computing connected components (CCs) [Yan et al., 2014b], the Pregel algorithm for computing biconnected components [Yan et al., 2014b], and the Pregel algorithm for computing minimum spanning forest [Salihoglu and Widom, 2014]. We illustrate the idea of pointer jumping using two PPAs, one for list ranking and the other for computing CCs.

Example 3: List Ranking. The list ranking problem is as follows, where we assume elements in a list are linked backwards from the tail to the head (the algorithm for the forward linking order can be similarly derived). Consider a linked list $\mathcal{L}$ with n vertices, where each vertex v is associated with a value $val(v)$ and a link to its predecessor $pred(v)$ (i.e., $a(v)$ consists of $val(v)$ and $pred(v)$). The vertex v at the head of $\mathcal{L}$ has $pred(v) = null$. For each $v \in \mathcal{L}$, let us define $sum(v)$ to be the sum of the values of all the vertices from v following the predecessor link to the head. The goal is to compute $sum(v)$ for every $v \in \mathcal{L}$. If $val(v) = 1$ for any $v \in \mathcal{L}$, then $sum(v)$ is simply the rank of v in the list, i.e., the number of vertices preceding v plus 1. Note that in the input data, vertices are in arbitrary order. Albeit simple, list ranking is important in parallel graph computation; for example, it is a building block of the PPA for computing biconnected components [Yan et al., 2014b].

We now describe a BPPA for list ranking. Initially, each vertex v assigns $sum(v) \leftarrow val(v)$. Then in each round, each vertex v does the following: If $pred(v) \neq null$, v sets $sum(v) \leftarrow sum(v) + sum(pred(v))$ and $pred(v) \leftarrow pred(pred(v))$; otherwise, v votes to halt. The if-branch is accomplished in three supersteps: (1) v sends a message (whose value is its own ID) to $u = pred(v)$ requesting for the values of $sum(u)$ and $pred(u)$; (2) u sends back the requested values to each requesting vertex v; and (3) v updates $sum(v)$ and $pred(v)$ using the received values. This process repeats until $pred(v) = null$ for every vertex v, at which point all vertices vote to halt and we have $sum(v)$ as desired.

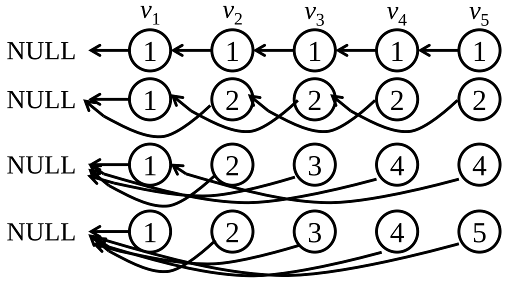


Figure 3.1: Illustration of the BPPA for List Ranking

Figure 3.1 illustrates how the algorithm works. Initially, objects v_1–v_5 form a linked list with $sum(v_i) = val(v_i) = 1$ and $pred(v_i) = v_{i-1}$. Let us now focus on v_5. In Round 1, we have $pred(v_5) = v_4$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_4) = 1 + 1 = 2$ and $pred(v_5) \leftarrow pred(v_4) = v_3$. One can verify the states of the other vertices similarly. In Round 2, we have $pred(v_5) = v_3$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_3) = 2 + 2 = 4$ and $pred(v_5) \leftarrow pred(v_3) = v_1$. In Round 3, we have $pred(v_5) = v_1$ and so we set $sum(v_5) \leftarrow sum(v_5) + sum(v_1) = 4 + 1 = 5$ and $pred(v_5) \leftarrow pred(v_1) = null$. Obviously, the algorithm takes $O(\log n)$ rounds and is a BPPA. Moreover, a vertex v communicates with $pred(v)$ which may not be its direct neighbor.

Example 4: S-V Algorithm. [Yan et al., 2014b] adapts Shiloach-Vishkin's PRAM algorithm for computing CCs [Shiloach and Vishkin, 1982] to work on Pregel, which is called the S-V algorithm. We present a simplified version of the S-V algorithm here which is more efficient. Throughout the algorithm, vertices are organized by a forest such that all vertices in a tree belong to the same CC. Each vertex v maintains a pointer $D[v]$ indicating its parent in the forest (i.e., $a(v) = D[v]$). We relax the tree definition a bit here to allow the tree root w to have a self-loop (i.e., $D[w] = w$).

At the beginning, each vertex v initializes $D[v] \leftarrow v$, forming a self loop as shown Figure 3.2(a). Then, the algorithm proceeds in rounds, and in each round, the pointers are updated in two steps: (1) *tree hooking* (see Figure 3.2(b)): for each edge (u, v), if u's parent $w = D[u]$

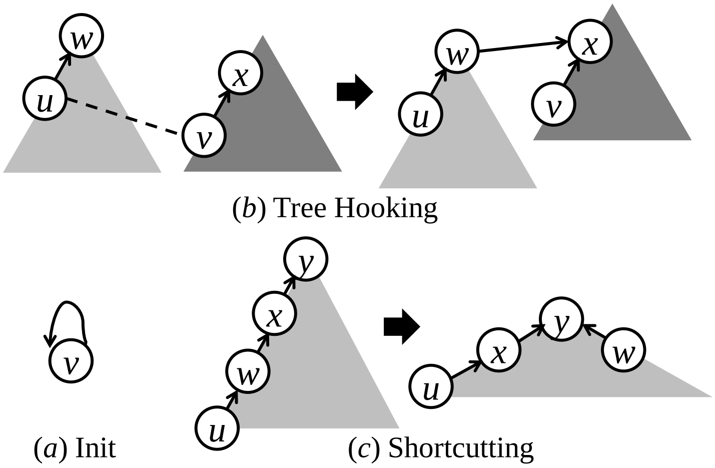


Figure 3.2: Illustration of the S-V Algorithm

is a tree root and $D[v] < D[u]$, we hook w as a child of v's parent $D[v]$ (i.e., we merge the tree rooted at w into v's tree); (2) *shortcutting* (see Figure 3.2(c)): for each vertex v, we move it closer to the tree root by pointing v to the parent of v's parent, i.e., $D[D[v]]$. Note that Step 2 has no impact on $D[v]$ if v is a root or a child of a root.

The algorithm repeats these two steps until no vertex v has $D[v]$ updated in a round (checked by using aggregator), by which time every vertex is in a star (i.e., tree of height 1), and each star corresponds to a connected component. Since $D[v]$ monotonically decreases during the computation, at the end $D[v]$ equals the smallest vertex in v's CC (which is also the root of v's star). Similar to the request-respond operation in list ranking, each step of S-V can be formulated in Pregel as a constant number of supersteps, and since shortcutting guarantees the $O(\log |V|)$-round bound, the algorithm is a PPA. However, it is not a BPPA since a parent vertex may communicate with many more vertices than its own neighbors (e.g., a root). Moreover, a vertex v communicates with $D[v]$, which may not be its direct neighbor.

3.3 Optimizations in Communication Mechanism

Since Google's Pregel is not public, a lot of open-source Pregel-like systems have been developed recently. The key feature of a Pregel-like system is that, it adopts a message passing model for programming and communication. This feature differentiates Pregel-like systems from those systems that adopt a shared memory abstraction. Existing Pregel-like systems improve the basic Pregel model from various aspects. This section introduce a few popular Pregel-like systems with improvements in the communication mechanism. We introduce more Pregel-like systems that improve the framework of Pregel from other aspects in the next three sections.

Giraph [Ching et al., 2015]. Apache Giraph is probably one of the most popular Pregel-like systems, which is written in Java and built on top of Hadoop. The good performance and scalability of Giraph is mainly contributed by Facebook's improvements to an earlier Giraph version. There are three important improvements which we list as follows. Firstly, Facebook researchers improved each worker to support multithreading in order to achieve fine grain parallelism. Secondly, the old Giraph version maintains vertices, edges and messages as native Java objects, which consumes excessive memory and garbage collection time; Facebook researchers solved the problem by serializing the edges and messages into byte arrays to reduce the number of objects. Thirdly, a superstep splitting technique is developed to split a message-heavy superstep into several steps, so that the number of messages transmitted in each step does not exceed the memory size. Superstep splitting is only effective when a received message can be aggregated to a value by the receiver vertex, or more formally, *compute*(.) is distributive (e.g., message summation in PageRank computation).

Pregel+ [Yan et al., 2015]. Pregel+ is written in C/C++, and thus recycles memory in time and keeps the memory footprint small. Since the system has full control of memory usage, all vertices, edges and messages are stored as main-memory objects and there is no need of serialization (except for sending messages). Pregel+ further developed two techniques to reduce the number of messages as described below.

The first technique is to create mirrors of each high-degree vertex v on all other workers that contain v's neighbor(s). The adjacency list of v is partitioned among its mirrors, where each mirror maintains the sub-list of v's neighbors in its local worker. In the PageRank (resp. Hash-Min) algorithm, v broadcasts the same value $a(v)/d_{out}(v)$ (resp. $a(v)$) to all its out-neighbors (resp. neighbors); for such an algorithm, v simply sends the value to every mirror, which then forwards the value to all (and possibly many) local neighbors. The message value towards a neighbor w (on worker W_w) may also be post-processed by the mirror of v on W_w using the edge value of (v, w) (according to an optionally user-defined function) before forwarding it to w. Since each worker has at most one mirror for v, the total number of messages incurred by v is bounded by the number of workers $|\mathbb{W}|$, which can be much smaller than v's degree.

However, since a mirrored vertex forwards its value directly to its mirrors, it loses the chance of message combining. Therefore, there is a tradeoff between vertex mirroring and message combining in reducing the number of messages, and we should only mirror high-degree vertices. [Yan et al., 2015] proves that the number of messages is minimized when we mirror all vertices with degree at least $|\mathbb{W}| \cdot \exp\{d_{avg}/|\mathbb{W}|\}$ where d_{avg} is the average vertex degree.

The second technique is designed for pointer jumping algorithms where a vertex needs to communicate with a large number of other vertices that may not be its neighbors. To see this, consider the last round of the S-V algorithm described in Section 3.2, where a root r vertex needs to communicate with all other vertices v in r's component, since they form a star rooted at r (i.e., $D[v] = r$). Here, each vertex v will send requests to r for the value of $D[r]$ in a superstep, and r will receive these requests and send $D[r]$ to every requesting vertex in the next superstep. Pregel+ prevents r from receiving and sending a lot of messages, by combining all requests on each worker as one request towards r, and r only responds to every requesting worker rather than every requesting vertex. To support this optimization, Pregel+ allows a vertex v to request for the value of another vertex (e.g., r), and the value (e.g., $D[r]$) can then be directly accessed in the next superstep.

GPS [Salihoglu and Widom, 2013]. GPS is written in Java, and can be more efficient than Giraph when a parameter called polling time is set as small value such as 10 ms (the default setting is 1 s, which forces any superstep to take at least 1 s) [Lu et al., 2014]. There are two main optimizations over the basic model of Pregel. Firstly, it supports a technique called LALP which is similar to vertex mirroring in Pregel+. However, to deal with the conflict between vertex mirroring and message combining, it simply does not perform sender-side message combining at all, and is thus less effective than Pregel+ in terms of message reduction. Secondly, since the workload may change during computation, GPS considers vertex migration for dynamic load balancing. However, vertex migration is too costly to be effective, as we shall discuss in Section 3.4.

MOCgraph [Zhou et al., 2014]. MOCgraph is developed on top of Giraph, and adopts the *message online computing* (MOC) model to eliminate the memory space consumed by messages. The idea is to let in-memory vertices absorb incoming messages directly without buffering them. While MOCgraph adopts asynchronous execution to allow faster convergence, it retains the concept of "superstep" which differentiates it from GraphLab. In each superstep of the MOC model, every vertex v calls a UDF $sendMessages(.)$ exactly once to send messages, whose values are computed from v's latest value; meanwhile, each message received by a vertex u reads and updates u's latest vertex value by calling another UDF $onlineCompute(.)$. Note that this essentially requires that $onlineCompute(.)$ is distributive (like message combiner of Pregel). Therefore, the MOC model generates and consumes messages in the same superstep, carrying no messages across supersteps.

To support synchronous execution like in Pregel, the UDF $v.sendMessages(.)$ sends messages according to v's old value updated by the last superstep, rather than using the latest value. In other words, each vertex v maintains two values, the old value for computing outgoing messages, and the latest value to be updated by incoming messages.

If the memories cannot hold all vertices, MOCgraph partitions vertices into partitions on disks, and incoming messages have to be buffered to disks. Consider a message m targeted at vertex v in a partition P. If P is not in memory, m has to be appended to a message file that corresponds to P. To process a vertex partition P, messages in the corresponding message file will first be loaded to update P, before vertices in P send messages. Note that keeping more vertices in memory allows more incoming messages to be absorbed without buffering, and thus MOCgraph separates edges from vertex partitions to allow more vertex partitions to be kept in memory (while only those edges of the current partition in processing are kept in memory). A hot-aware re-partitioning strategy is used to keep those partitions that tend to receive more messages in memory.

3.4 Load Balancing

In this section, we review some Pregel-like systems that improve load balancing during computation by two techniques: vertex migration and dynamic concurrency control.

Vertex Migration. A Pregel job may exhibit different workload distributions in different supersteps. For example, in Hash-Min, while all vertices send messages at the first superstep, few vertices are active in the last few supersteps. The set of active vertices (or more strictly, the set of vertices that send messages) in a superstep is called a working window, or simply, *wind*, by [Shang and Yu, 2013]. The idea of *vertex migration*, or *dynamic graph partitioning*, is to migrate vertices from workers with heavy workloads to those with light workloads during the computation, which also endeavors to reduce the communication cost (e.g., crossing machine edges).

However, there are two challenges in vertex migration: (1) migrating a vertex also requires migrating its adjacency list, which is more costly than sending a message; (2) it is difficult to *catch the wind*: for example, a migrated vertex may just converge and need no more computation, and a worker currently with a light workload may be heavily loaded in the next superstep due to the activation of many of its vertices. Even worse, since the vertex-to-worker relationship changes, we cannot simply derive the worker of a vertex by hashing its ID, and more costly method should be used. Note that the vertex-to-worker mapping is critical to system performance, because when a vertex u sends a message to another vertex v, it needs to know which machine the message should be sent to (i.e., the machine that v resides in). Therefore, existing efforts on vertex migration turn out to be not very effective, which we review next.

[Shang and Yu, 2013] developed a Pregel-like system on top of HAMA, which keeps track of the vertex-to-worker mapping by a Lookup Table [Tatarowicz et al., 2012]. A few policies are designed based on the immediate previous *wind* to catch the *wind* in the next superstep. The best reported result is a 31.5% reduction of execution time for PageRank computation, while the ratio is merely 2% and 9% for shortest path computation and maximal matching.

Mizan [Khayyat et al., 2013] identifies the cause of workload imbalance using distributed measurements of the performance characteristics of all vertices, and constructs a vertex migration plan without requiring centralized coordination. The vertex-to-worker mapping is maintained by a distributed hash table (DHT). [Khayyat et al., 2013] partitioned vertices using hash-based ID partitioning, range-based ID partitioning and METIS [Karypis and Kumar, 1998] in their experiments, and found that for both hash-based and METIS partitioning, dynamic migration did not improve the results; while around 40% improvement was observed for range-based partitioning. Recently, [Han et al., 2014a] reported that Mizan does not function correctly with dynamic migration.

GPS also supports dynamic graph partitioning. Instead of using a lookup table to keep the vertex-to-worker mapping, GPS relabels the IDs of the migrated vertices so that the mapping can still be computed by the hash function. The additional overhead is to update the old ID in the adjacency lists of other vertices with the relabeled ID. During vertex migration, GPS keeps the number of vertices in each worker unchanged, and a vertex is migrated only if the number of messages is significantly decreased after the migration (controlled by a threshold). However, even the developers of GPS themselves do not recommend to enable vertex migration. For example, Semih said the following comments:

"*... unless your job is going to run a large number of supersteps, running dynamic repartitioning will slow down your job ...*"[1]

"*In general, I advice not to do dynamic repartitioning and also not to work on it. I think it's very difficult to get benefits out of it in a real system implementation. The overheads are just too high.*"[2]

Dynamic Concurrency Control. While previous explorations on vertex migration turned out not very effective, there is an existing work that dynamically balances the workload within each individual worker, which exhibits reasonable performance improvement. Specifically, [Shao et al., 2015] observed that a high-quality graph partitioning (e.g., from METIS) sometimes even decreases the overall performance in existing big graph systems, despite the reduced number of crossing-machine edges. This is because, while the number of messages from remote machines is reduced, the number of messages sent by local vertices significantly increases. However, existing systems are not partition-aware, and still allocate the same amount of computing resources (e.g., threads) to process remote messages and local messages, respectively. Therefore, the increased volume of local messages leads to a performance bottleneck (since the number of threads processing them is not increased), and the cost of processing local messages dominates the overall cost.

[Shao et al., 2015] developed the PAGE system, which adopts a *dynamic concurrency control* model, to overcome the above limitation. Each worker in PAGE monitors measurements such as message generation speed, local message processing speed and remote message processing speed. Based on these measurements, PAGE dynamically adjusts the numbers of threads for processing local and remote messages, respectively, so that (1) the speed of message processing matches the speed of incoming messages, and that (2) the numbers of threads assigned to process *local* and *remote* messages, are proportional to the incoming speeds of *local* and *remote* messages, respectively.

[1] `https://groups.google.com/forum/#!searchin/stanfordgpsusers/repartition/stanfordgpsusers/3Wzlnm1eXbw/xf8c9hBUCGMJ`

[2] `https://groups.google.com/forum/#!searchin/stanfordgpsusers/repartition/stanfordgpsusers/HV04gc-2Tcs/T-tlgC2GpdgJ`

3.5 Out-Of-Core Execution

While most distributed big graph systems are in-memory systems, out-of-core support has recently attracted a lot of attention due to the real demands from academic institutes and small businesses that could not afford memory-rich clusters. For example, [Bu et al., 2014] reported that in the Giraph user mailing list there are 26 cases (among 350 in total) of out-of-memory related issues from March 2013 to March 2014. As another example, [Zhou et al., 2014] reported that to process a graph dataset that takes only 28GB disk space, Giraph and GraphLab need 370GB and 800GB memory space, respectively; and when memory resources become exhausted, the performance of Giraph degrades seriously while GraphLab simply crashes.

Two solutions are currently available for processing a big graph when memory space is insufficient. The first solution is to use a single-PC disk-based (or SSD-based) graph systems like GraphChi [Kyrola et al., 2012] and X-Stream [Roy et al., 2013]. Since these systems adopt the shared memory abstraction, we will review them in Chapter 5. However, the performance of these systems is limited by the disk bandwidth of one PC, and thus the processing time scales linearly with the graph size. To scale to larger graphs, the second solution is to use a distributed graph system that supports efficient out-of-core execution. In such a system, the graph is partitioned among all machines in a cluster, and during computation, each machine only processes its own part of the graph on the local disk. As a result, the bandwidth of all disks in a cluster is fully utilized, but the tradeoff is that the overhead of network communication is incurred.

We now briefly review the existing distributed big graph systems that support out-of-core execution. Giraph has been extended with out-of-core capabilities to solve the out-of-memory issues[3], where users can enable “out-of-core graph” (resp. “out-of-core messages”) to store graph partitions (resp. “messages”) to local disk(s) if the in-memory buffer overflows. Users may set the maximum number of partitions and/or messages allowed to be kept in memory, and provide a list of

[3] http://giraph.apache.org/ooc.html

paths corresponding to different disks on each machine, so that Giraph will access all disks in a round-robin fashion to fully utilize the disk bandwidth. Pregelix [Bu et al., 2014] models the semantics of Pregel by relational operations like join and group-by, and leverages a general-purpose dataflow engine for out-of-core execution. In contrast, GraphD [Yan et al., 2016d] tailors its out-of-core execution design to the computation model of Pregel, and is thus able to avoid expensive operations like join and group-by. Recently, Chaos [Roy et al., 2015] extends X-Stream to work in a distributed environment, but it is only efficient when network bandwidth far outstrips storage bandwidth (which is also an assumption in its system design).

In the next subsection, we review the GraphD system. Pregelix will be reviewed in Section 8.2 when we discuss dataflow-based systems, and GraphChi, X-Stream and Chaos will be reviewed in Sections 5.2.1 and 5.2.2 when we discuss systems that adopt shared memory abstraction.

3.5.1 GraphD

GraphD [Yan et al., 2016d] is designed to run on a cluster of commodity PCs connected by Gigabit Ethernet, which are readily available in academic institutes and small businesses. In this setting, the bandwidth of sequential disk scan is usually much higher than the actual network bandwidth [Yan et al., 2016d, Shen et al., 2014]. This is because (1) sequential disk scan is significantly accelerated by memory cache, and (2) the network resource is contended by all the workers in a cluster, limiting the point-to-point connection throughput. Therefore, if each worker streams disk-resident data (e.g., edges and messages) in parallel with message transmission, it is possible to hide the time for disk streaming inside the time for network communication. GraphD exploits this characteristic and achieves performance comparable to (and sometimes even faster than) an in-memory Pregel-like system.

Distributed Semi-Streaming Model. Like Pregel, GraphD partitions vertices among the workers in a cluster using a hash function $hash(.)$. However, GraphD adopts a novel *distributed semi-streaming* (DSS) computation model, where each worker only keeps the states of its vertices in main memory, and the adjacency lists are stored in a file on local disk, which we call as the edge stream, denoted by S^E. The state of a vertex v includes its ID, $a(v)$, $active(v)$, and $d(v)$ (or $d_{out}(v)$ for a directed graph), whose memory space is bounded by a small constant.

A worker W organizes its in-memory vertex states by an array A, whose elements are ordered by vertex ID. The adjacency lists in S^E are also ordered by the ID of their corresponding vertices. In a superstep, the $compute(.)$ function is called on the active vertices in A in serial. Since a vertex v needs to access $\Gamma(v)$ in $v.compute(.)$, the next $d(v)$ items are sequentially read from S^E to form $\Gamma(v)$.

To skip the adjacency lists of a series of inactive vertices efficiently, W sums the degree of these vertices as sum, and forwards the read position in S^E by sum items. If data at the new position is not contained in the in-memory streaming buffer of S^E, the buffer is refilled with data starting from the new position. This design avoids reading all items in S^E when only a small fraction of vertices are active.

Since only vertex states are kept in memory, the total RAM space required by GraphD is only $O(|V|)$. Moreover, [Yan et al., 2016d] proved that in a cluster of workers, $\mathbb{W}$, every worker requires only $O(|V|/|\mathbb{W}|)$ RAM space for the state array A, with a high probability of $(1 - O(1/|V|))$. The RAM space required by streaming buffers and buffers for sending and receiving messages are negligible compared with A.

Message Streams. Like edges, messages are also treated as disk streams in GraphD. For simplicity, we only discuss Pregel algorithms where message combiner is applicable. Each worker W maintains $|\mathbb{W}|$ outgoing message streams $S_1^O, \ldots, S_{|\mathbb{W}|}^O$, where each stream S_i^O stores messages towards the i-th worker, denoted by W_i. If a vertex on W sends a message to another vertex on W_i, the message is simply appended to S_i^O.

In order to perform vertex-centric computation (which generates messages) and sending generated messages in parallel, a message stream S_i^O is split into multiple files as follows: if the size of the current file written by S_i^O reaches a threshold $\mathcal{B}$, a new file is created for appending more messages. A sending thread probes all the $|\mathbb{W}|$ message streams, and whenever it finds a stream S_i^O that has unsent message files, it performs message combining over these messages and sends them in one batch to W_i.

In GraphD, a message is first appended to a message stream on local disk, and then loaded by the sending thread for sending. This design may appear to be slower than sending a message directly without going through local disk. However, since message generation is much faster than message transmission, if we buffer all messages in main memory, vertex-centric computation has to be stalled when too many messages are generated, waiting for some buffered messages to be sent to leave room for generating more messages. In contrast, by buffering messages to local disks, GraphD avoids stalling vertex-centric computation (and thus message generation). Moreover, since network bandwidth is lower than disk bandwidth, the performance bottleneck is message sending rather than streaming outgoing message streams.

Finally, we show that expensive disk-based operations like external-memory join and group-by are not needed. Let us denote the set of vertices on a worker W by $V(W)$. The sending thread of a worker maintains an in-memory table with $\max_{W \in \mathbb{W}} |V(W)|$ message elements. To combine messages in S_i^O before sending, each message is processed in turn: if the message targets at a vertex u in W_i, it is directly combined to the table entry that saves u's combined message. Similarly, the receiving thread of a worker W maintains an in-memory table with $|V(W)|$ message elements, and each received message is processed in turn: if the message targets at a vertex u in W, it is directly combined to the table entry that saves u's combined message. Each combined message is used as the input to *compute*(.) of the corresponding vertex in the next superstep. Recall that $\max_{W \in \mathbb{W}} |V(W)|$ is bounded by $O(|V|/|\mathbb{W}|)$ with high probability, and thus, the in-memory tables maintained by the sending and receiving threads of a worker do not breach the $O(|V|/|\mathbb{W}|)$ RAM space bound of DSS.

3.6 Fault Tolerance

Fault tolerance is important for distributed systems: a long-running job should survive (or recover quickly from) the crash of any machine, rather than restart from the very beginning. For this goal, Pregel periodically saves the computation state of the current superstep (including vertex states, adjacency lists and messages) as a checkpoint to GFS, so that when failure happens, computation can roll back to the latest checkpointed superstep. Open-source Pregel-like systems like Giraph also support checkpointing, by backing up the state of a superstep to HDFS. Note that the contents of a checkpoint are not lost even if some machines crash, since a DFS replicates data on multiple machines.

A checkpoint can be written for every few (e.g., 10) supersteps, but since the running time of different supersteps may be different, a better solution is to use time-based checkpointing, e.g., to checkpoint the current superstep if at least 5 minutes have passed since the last checkpoint is written. Since a checkpoint is written by all workers after they finish a superstep, this method is called *coordinated checkpointing*.

Besides coordinated checkpointing, many other rollback-recovery protocols have been extensively studied for message-passing systems and are well surveyed by [Elnozahy et al., 2002], such as *incremental checkpointing* and *uncoordinated checkpointing*. Incremental checkpointing avoids rewriting portions of states that do not change between consecutive checkpoints. For example, for Pregel algorithms without topology mutations, adjacency lists only need to be saved in the first checkpoint. Uncoordinated checkpointing protocols like Chandy-Lamport snapshot [Chandy and Lamport, 1985] is designed for asynchronous message-passing systems. For example, Maiter uses Chandy-Lamport snapshot while GraphLab adapts it for systems that use a shared-memory abstraction.

Recently, many novel and efficient fault recovery mechanisms were designed for Pregel-like systems and other vertex-centric systems, such as message logging [Shen et al., 2014], lightweight checkpointing [Yan et al., 2016c], optimistic recovery [Schelter et al., 2013] and replication-based fault-tolerance [Wang et al., 2014]. We now introduce the above fault tolerance mechanisms in the following subsections.

3.6.1 Chandy-Lamport Snapshot

We describe Chandy-Lamport snapshot in the context of vertex-centric computation. Specifically, a checkpointing request is initiated at fixed intervals, where each worker checkpoints the current states of its vertices (and their related messages) to HDFS. However, the checkpointed states may be inconsistent. To see this, consider two vertices u and v, and assume that the following four events happen in order: (1) u's state is checkpointed, (2) u updates $a(u)$ and sends a message to v, (3) v receives the message and updates $a(v)$, (4) v's state is checkpointed. Then, any checkpoint containing the saved states of u and v is inconsistent, since $a(u)$ refers to the old value before Event (2), but $a(v)$ is affected by the updated value of $a(u)$ after Event (2).

Chandy-Lamport snapshot assumes that communication channels are FIFO, and prevents the above inconsistency as follows. Whenever a vertex u is checkpointed, it broadcasts a checkpointing request to all vertices that u will send messages to, before sending any messages. When a vertex v receives a checkpointing request, it ignores the request if it has already checkpointed its state for the current round of checkpointing; otherwise, v checkpoints its state and broadcasts a checkpointing request to all its neighbors.

We now illustrate how this protocol eliminates state inconsistency in the previous example. Specifically, u will send a checkpointing request to v right after Event (1), and since communication channels are FIFO, v receives the checkpointing request before the message from u is received. Therefore, v checkpoints $a(v)$ before Event (3), and thus both $a(u)$ and $a(v)$ do not reflect the effect of Event (2) and are thus consistent.

3.6.2 Recovery by Message-Logging

When a Pregel job fails at a superstep (let it be Step s_{fail}), the latest checkpoint (at Step s_{cp}) is loaded from HDFS and all vertices roll their states back to Step s_{cp}. Then, the computation restarts from Step s_{cp} as in normal execution. However, this approach wastes computation. Specifically, while a vertex in a failed worker has to be reassigned to another alive worker and to perform recomputation from Step s_{cp}, the state of a vertex in a surviving worker is already at Step s_{fail} and there is no need to recompute it.

However, when coordinated checkpointing is used alone, a surviving vertex has to perform recomputation from Step s_{cp} to Step s_{fail}. This is because its messages towards vertices in failed workers are needed as input to *compute*(.) when those vertices recompute their states. To avoid surviving vertices from rolling their states back, [Shen et al., 2014] proposed to let each vertex log the messages that it generates (in *compute*(.)) to the local disk of its worker, before sending them. Since network bandwidth is much lower than disk streaming bandwidth in a Gigabit Ethernet environment, [Shen et al., 2014] observed negligible cost for logging messages. During recovery, only those messages that target at the reassigned vertices need to be transmitted. Specifically, (1) a surviving vertex simply forwards its logged messages towards those reassigned vertices, while (2) a reassigned vertex performs vertex-centric computation, logs all its generated messages (for forwarding in case some other workers fail later), but only sends those messages that target at the reassigned vertices. This is sufficient to guarantee that a reassigned vertex receives messages from all vertices between Step s_{cp} and Step s_{fail}, and the recovery is much faster as the communication workload is much lower than during normal execution.

However, it is not sufficient to classify vertices into only two classes, surviving ones and reassigned ones. This is because a cascading failure may happen at a superstep $s_{cas} < s_{fail}$ during the recovery, at which time vertices may be at three different states: (1) vertices surviving both failures are at Step s_{fail}, (2) vertices that are reassigned due to the first failure but survive the second failure are at Step s_{cas}, and (3) vertices that does not survive the second failure are reassigned and are at Step s_{cp}.

Let us denote the state (in terms of superstep number) of a vertex v by $s(v)$, then a recovery algorithm that is robust to cascading failures should classify vertices by their states, and a vertex v whose state is at Step $s(v)$ should perform vertex-centric computation only after Step $s(v)$ is recovered. Moreover, a vertex v that does not perform vertex-centric computation only needs to forward logged messages to those vertices u where $s(u) < s(v)$. The detailed recovery algorithms can be found in [Shen et al., 2014] and in [Yan et al., 2016c].

When a failure happens, [Shen et al., 2014] also reassigns vertices in failed workers to multiple alive workers using a cost-sensitive reassignment algorithm, to achieve parallelism of recomputation and to reduce the recovery time. The reassignment is computed by the master and written to a zookeeper slave; each worker slave then obtains the reassignment from the zookeeper and loads those failed vertices that are assigned to it.

3.6.3 Lightweight Checkpointing

In existing Pregel-like systems, the checkpoint of a superstep contains (1) vertex states which take $O(|V|)$ space, (2) adjacency lists which take $O(|E|)$ space, and (3) all messages generated in the superstep. The messages usually take $O(|E|)$ space (e.g., in PageRank computation), but can be much larger (e.g., $O(|E|^{3/2})$ in triangle counting).

While we can use incremental checkpointing to save the $O(|E|)$ amount of adjacency lists in a checkpoint, it is still expensive to write a checkpoint to HDFS due to the large message volume. [Yan et al., 2016c] proposed a lightweight checkpointing method to further remove messages from a checkpoint, which improves the checkpointing time by tens of times. To recover from a failure, the vertex states are loaded from the latest checkpoint, and outgoing messages are then generated from the vertex states for sending (rather than loaded directly from the checkpoint).

The idea of lightweight checkpointing is motivated by the observation that, in many Pregel algorithms, the logic of $v.compute(msgs)$ can be reformulated as two steps: (1) to update the vertex state of v using the incoming messages *msgs*, and (2) to generate outgoing messages solely based on the updated state of v. In other words, given the updated state of a vertex, outgoing messages do not have to be computed by looking at the incoming messages.

Lightweight checkpointing is directly applicable to many Pregel algorithms. For example, in PageRank computation, incoming messages are first summed up to update $a(v)$, and then outgoing message to each out-neighbor is computed as $a(v)/d_{out}(v)$. For other Pregel algorithms, it may be necessary to include additional fields to $a(v)$. For example, in Hash-Min, $a(v)$ should include not only the minimum vertex ID seen by v, denoted by $min(v)$, but also a boolean flag, denoted by $updated(v)$, indicating whether $min(v)$ is updated by incoming messages. If the smallest incoming message min^* is less $min(v)$, v sets $min(v) \leftarrow min^*$ and $updated(v) \leftarrow true$; otherwise, $updated(v)$ is set as *false*. The outgoing messages are generated from $a(v)$ as follows: if $updated(v)$ is *false*, no message is generated; otherwise, a message with value $min(v)$ is sent to each neighbor of v.

In fact, lightweight checkpointing is applicable even when the outgoing messages need to be computed by looking at an incoming message, since the incoming message can be included into $a(v)$. For example, in the list ranking algorithm in Section 3.2, a vertex v sends its ID to its predecessor $u = pred(v)$ requesting for $a(u)$. Therefore, we need to add the message that u receives (i.e., v) into $a(u)$; otherwise, u does not know to which vertex (i.e., u's successor) a response message should be sent to.

However, [Yan et al., 2016c] indicates that there exist some algorithms where a vertex u may receive requests from many other vertices, and it needs to respond to all of them. In this case, lightweight checkpointing is not applicable to the responding superstep. For example, in the S-V algorithm in Section 3.2, a vertex u may have many children (i.e., each child v has $D[v] = u$), and thus need to respond to many vertices. Instead of checkpointing a responding superstep by also saving the many requesting messages, [Yan et al., 2016c] proposed to postpone the checkpointing to the first superstep after it such that lightweight checkpointing is applicable.

[Yan et al., 2016c] also applied this idea to the message-logging based recovery algorithm of [Shen et al., 2014], which avoids the slow down of failure-free performance due to garbage collecting locally logged messages. Specifically, to avoid logged messages from using up disk space, after a checkpoint is written, it is necessary to delete all outdated messages logged before the commit of the checkpoint. However, the deletion is time-consuming since all messages generated between two checkpoints need to be deleted. In [Yan et al., 2016c], only vertex states are logged to local disks, and messages to be forwarded to the reassigned vertices during recovery are generated from the logged vertex states. Since the log volume becomes very small, garbage collection takes negligible time. However, for a superstep to which lightweight checkpointing is not applicable, message logging should be used instead of vertex-state logging.

3.6.4 Other Methods

For a narrower class of self-correcting fix-point algorithms, optimistic recovery [Schelter et al., 2013] eliminates the need of checkpointing at all. Specifically, since such an algorithm converges to the same result regardless of the initial vertex states, when failure happens, [Schelter et al., 2013] simply re-initiates the states of those vertices in failed workers and continues execution. For example, in PageRank computation, as long as the sum of all vertex values (denoted by sum_{all}) are fixed, no matter how the vertex values are initialized, the converged values are the same. Therefore, when failure happens, a user-defined compensate function obtains (1) the sum of values of all surviving vertices, denoted by sum_{alive}, and (2) the number of surviving vertices, denoted by n_{alive}; the function then re-initializes the value of each vertex in a failed worker by $(sum_{all} - sum_{alive})/(|V| - n_{alive})$, so that the new sum of all vertex values still equals sum_{all}. The computation then continues until converged.

Imitator [Wang et al., 2014] avoids checkpointing by constructing k replicas of each vertex on k different workers. As long as less than k workers crash, each vertex still contains at least one replica and computation is not lost. However, replicas consume additional memory space, and any update to a vertex should be synchronized to all replicas, which leads to additional communication overhead during normal execution.

Spark [Zaharia et al., 2012] is a distributed system for general computation, which features its lineage-based fault tolerance. Specifically, a dataset in Spark is represented as a Resilient Distributed Dataset (RDD) consisting of many partitions of records. Lineages of partitions are logged: for each partition in an RDD, those partitions (from other RDD(s)) that are involved in computing its records are logged. Since coarse-grained partitions are considered instead of individual records, the lineage DAG tends to be small. However, this approach is mainly useful for operations with narrow dependency, where a lost partition only depends on one or a few other partitions. The message-passing model of Pregel has a wide dependency, since a vertex may send messages to neighbors in many other partitions, and thus lineage-based recovery is ineffective. In fact, Spark also uses coordinated checkpointing when computing PageRank [Zaharia et al., 2012].

3.7 Summary

Starting from the pioneering vertex-centric system, Google's Pregel, we took a journey through various vertex-centric systems that adopt message passing for communication, and execute synchronously. The synchronous model of these systems makes it easy to analyze and debug the behavior of an algorithm, and avoids the overhead of solving race conditions. Also, a user writes an application code by specifying how each vertex performs computation by sending other vertices messages. The flexibility of the message passing interface is that, a vertex can send messages to any other vertex that it can keep track of, not just its neighbors. This allows many of these systems to implement efficient parallel algorithms that perform pointer jumping like in their PRAM algorithm counterparts (See Section 3.2). However, note that some message passing systems sacrifice expressiveness to reduce memory consumption. For example, MOCgraph and superstep splitting of Giraph, are both designed for algorithms where a vertex sends messages to neighbors, and aggregates its received messages (see Section 3.3). We also reviewed how research has been actively conducted on improving the efficiency and robustness of the neat model of Pregel from different aspects in Sections 3.3–3.6. In the next chapter, we review those systems that go beyond the simple vertex-centric computation model rather than simply improving upon it.

4

Vertex-Centric Message-Passing Systems Beyond Pregel

The vertex-centric model of Pregel may not provide satisfactory performance to some important graph problems, and this chapter reviews three variants of Pregel's model that overcomes its inefficiency from different aspects. (1) For processing graphs with high-degree vertices and a large diameter, Section 4.1 reviews how several systems use a novel block-centric computation model to avoid heavy communication and large number of iterations. (2) For algorithms with asymmetric convergence rate, Section 4.2 reviews several systems that adapts Pregel's model for asynchronous execution, which schedule the computation frequency of each vertex according to its convergence rate. (3) Pregel's model is designed for offline analytics, and Section 4.3 reviews a system that inherits the vertex-centric interface of Pregel, but designs its runtime engine to be efficient for online graph querying.

4.1 Block-Centric Computation

We first review an important extension to the vertex-centric model of Pregel, i.e., the block-centric computation model. The vertex-centric model is mainly designed to process small diameter graphs like social networks. This is because it requires one superstep to propagate data for merely one hop, and thus, let the graph diameter be δ, the number of supersteps is often $O(\delta)$ (e.g., consider Hash-Min) unless pointer jumping is used. However, many real big graphs have a large diameter, such as continental road networks and terrain meshes. Even non-spatial graphs may have a large diameter, such as web graphs which exhibit spatial locality: a local webpage is more likely to link to another local webpage than to link to a webpage elsewhere (e.g., abroad). For example, [Salihoglu and Widom, 2014] reported that it takes 4546 and 6509 supersteps to compute the strongly connected components of two web graphs *uk-2005* and *sk-2005*. To solve this problem, they designed an algorithmic optimization called FCS (Finishing Computations Serially), which monitors the number of active vertices, and once the number is small enough, these active vertices (and their adjacency lists) are sent to the master and serial computation is performed on the constructed subgraph. After applying FCS, the number of supersteps is reduced to 3278 and 2857, respectively, which is still very large.

A satisfactory solution to this problem is to extend the vertex-centric computation model with a novel block-centric computation model, the idea of which is briefly introduced next. Specifically, the vertices of a graph are partitioned into multiple blocks, such that each block has a strong cohesion: a vertex in a block B is more likely to connect to another vertex in B than to a vertex in another block. All vertices in a block is assigned to one worker. When vertices in a block B receive incoming messages, they do not just update their own states using these messages, but also propagate the state updates through all vertices in B until convergence. Since in-block state propagation is performed in serial without communication, the additional computation overhead incurred is negligible compared with the significant reduction in message number and in superstep number.

Block-centric computation well solves the problem of large graph diameter: for example, [Yan et al., 2014a] reported that single-source shortest path computation on the USA road network takes 10789 supersteps (and 2832 seconds) in the vertex-centric model, and finishes in only 59 supersteps (and 11 seconds) with block-centric computation.

Challenges and Existing Solutions. However, there are two major challenges of applying a block-centric computation model. Firstly, an input graph should be pre-partitioned into blocks, but graph partitioning is expensive, especially for big graphs. Secondly, in Pregel, the worker that a vertex resides in can be computed directly from its vertex ID, but when block-centric computation is used, it is non-trivial to find a function that maps the IDs of all vertices in a block B to the ID of the worker that contains B. There also exist other challenges, such as how to define the stop (or convergence) condition of the computation.

Giraph++ [Tian et al., 2013] pioneered the idea of block-centric computation, which is termed “graph-centric” or “think like a graph”. The term “graph” here is equivalent to the concept of “block” described above: each block holds one partition of the input graph that is processed by a computing thread. The input graph is partitioned into blocks by a METIS-like algorithm [Karypis and Kumar, 1998], and vertex IDs are recoded by an independent MapReduce job, so that the worker that a vertex resides in can be directly computed from the new vertex ID. GoFFish [Simmhan et al., 2014] further decomposes the block of a worker into many subgraphs, where each subgraph is a maximal (weakly) connected component of the block. Instead of performing computation on each block, GoFFish performs computation on each connected subgraph, and terms the model as “subgraph-centric” computation. [Simmhan et al., 2014] claims that their new model has two benefits over Giraph++: (1) decomposing a partition into connected subgraphs increases the opportunity of parallelism, and (2) since each subgraphs is connected, a prioritized serial subgraph traversal (e.g., Dijkstra’s algorithm) is often sufficient, eliminating the need to process every vertex in the subgraph iteratively until convergence.

Blogel [Yan et al., 2014a] further allows each block to contain data structures like an adjacency list and a value, so that computation can be directly performed in the unit of blocks without the involvement of individual vertices. Since there are much less blocks than vertices, the workload is significantly reduced. In Blogel, each block is a connected subgraph, and a worker contains multiple blocks. This overpartitioning approach allows vertices and blocks to be distributed among workers in a more balanced manner. The ID of each vertex v is expanded to also store the IDs of the block and the worker that contain v, so that it is trivial to determine whether two vertices are in the same block, and which worker a vertex resides in. Blogel also proposed partitioning algorithms that are way more efficient than METIS-like methods.

The block-centric model has also been applied in single-machine in-memory graph processing. For example, GRACE [Xie et al., 2013] partitions vertices into blocks by METIS, so that each block fits in the CPU cache. All vertices in a block are processed together (possibly until convergence) without cache miss, before processing another block. This block-centric solution improves cache locality and mitigates the problem of limited memory bandwidth. Unlike Giraph++ and Blogel, GRACE only requires a user to specify the vertex-centric computing logic, and the block-centric computation is handled by GRACE as a proper scheduling of vertex-centric computation inside each block.

In the next two subsections, we introduce the two most important distributed block-centric systems: Giraph++ and Blogel. Since GRACE is a single-machine system that follows the shared-memory abstraction, we review it in more detail in Section 9.2.3. In Section 4.2.1, we will see a delta-based accumulative computation model that can be easily implemented in block-centric systems with a guarantee of result exactness.

4.1.1 Giraph++

The block-centric computation model was first introduced by Giraph++ [Tian et al., 2013], where it is also called a *graph-centric* model. In a nutshell, instead of exposing the view of a single vertex to the programmers, this graph-centric model opens up the entire subgraph of each partition to be programmed against.

Just like the vertex-centric model, the graph-centric model also divides the set of vertices in the original graph into partitions. Let $G = (V, E)$ denote the original graph with its vertices and edges, and let $P_1 \cup P_2 \cup \ldots \cup P_k = V$ be the k partitions of V, i.e. $P_i \cap P_j = \emptyset, \forall i \neq j$. For each partition P_i, the vertices in P_i, along with vertices they link to, define a subgraph G_i of the original graph. To be more precise, let V_i denote all the vertices that appear in the subgraph G_i, i.e. $V_i = P_i \cup \{v|(u, v) \in E \wedge u \in P_i\}$. Any vertex $u \in P_i$ is an *internal* vertex of G_i and any vertex $v \in (V_i \backslash P_i)$ is a *boundary* vertex. Note that a vertex is an internal vertex in exactly one subgraph, which is called the *owner* of the vertex, but it can be a boundary vertex in zero or more subgraphs. In the graph-centric model, subgraphs and partitions are used interchangeable when there is no ambiguity.

In the Giraph++ graph-centric model, for each internal vertex in a partition, we have all the information of its vertex value, edge values and incoming messages. But for a boundary vertex in a partition, we only associate a vertex value with it. This vertex value is just a temporary *local* copy. The *primary* copy of the vertex value resides in its owner's corresponding internal vertex. The local copies of vertex values are essentially caches of local computation in different partitions, and thus they have to be propagated to the primary copies through messages.

The distinction between internal vertices and boundary vertices is crucial, as in Giraph++ messages are only sent from boundary vertices to their primary copies. This is because the whole subgraph structure is available in the graph-centric model, and thus information exchange between internal vertices is cheap and immediate. An algorithm can arbitrarily change the state of any internal vertex at any point in time, without a need for a network message or a wait for the next superstep. Boundary vertex values can also be arbitrarily changed, but these changes will have to be propagated to the owners through messages, at the end of the superstep.

A program in Giraph++ is still executed in sequence of supersteps, separated by global synchronization barriers. However, in each superstep, the computation is performed on the whole subgraph in a partition. A new class *GraphPartition* is introduced to support the graph-centric programming model. This class allows users to 1) access all vertices in a graph partition, either internal or boundary, 2) check whether a particular vertex is internal, boundary or neither, 3) send messages to internal vertices of other partitions, and 4) collectively deactivate all internal vertices in this partition. The user defined *com-*

Algorithm 1: Connected Component Algorithm in Giraph

```
compute()
    if getSuperstep()==0 then
        setVertexValue(getVertexID());
    minValue=min(getMessages(), getVertexValue());
    if getSuperstep()==0 or minValue<getVertexValue() then
        setVertexValue(minValue);
        sendMsgToAllEdges(minValue);
    voteToHalt();
// combiner function
combine(msgs)
    return min(msgs);
```

pute() function in the *GraphPartition* class is on the whole subgraph instead of on individual vertex.

Example: Weakly Connected Component Algorithm

We use weakly connected component (WCC) algorithm for undirected graphs to demonstrate the difference between the vertex-centric model and the graph-centric model. Algorithm 1 and Algorithm 2 show the implementation of this algorithm in Giraph and Giraph++, respectively.

The WCC algorithm in the vertex-centric model (Algorithm 1) is based on label propagation. Initially in superstep 0, each vertex uses its own ID as its component label (each vertex is itself a connected component), then propagates the component label to all its neighbors. In subsequent supersteps, each vertex first finds the smallest label from the received messages. If this label is smaller than the vertex's current component label, the vertex modifies its label and propagates the new label to all its neighbors. For the example graph in Figure 4.1(a), Figure 4.1(b) depicts the vertex labels and the message passing in every superstep for this connected component algorithm.

Algorithm 2 demonstrates how the connected component algorithm is implemented in the graph-centric model. For the example graph in Figure 4.1(a), Figure 4.1(c) and 4.1(d) depict the subgraphs of its two partitions and the execution of the graph-centric algorithm, respectively. Since the graph-centric programming model exposes the whole

Algorithm 2: Connected Component Algorithm in Giraph++

```
compute()
    if getSuperstep()==0 then
        sequentialCC(); // run a sequential CC algorithm
        foreach bv in boundaryVertices() do
            sendMsg(bv.getVertexId(), bv.getVertexValue());
    else
        equiCC=∅;// store equivalent CCs
        foreach iv in activeInternalVertices() do
            minValue=min(iv.getMessages());
            if minValue<iv.getVertexValue() then
                equiCC.add(iv.getVertexValue(), minValue);
        equiCC.consolidate();// get min for equivalent CCs
        foreach iv in internalVertices() do
            changedTo=equiCC.uniqueLabel(iv.getVertexValue());
            iv.setVertexValue(changedTo);
        foreach bv in boundaryVertices() do
            changedTo=equiCC.uniqueLabel(bv.getVertexValue());
            if changedTo!=bv.getVertexValue() then
                bv.setVertexValue(changedTo);
                sendMsg(bv.getVertexId(), bv.getVertexValue());
    allVoteToHalt();
```

subgraph in a partition, an existing sequential algorithm can be utilized to detect the connected components in each graph partition. If a set of vertices belong to the same connected component in a partition, then they also belong to the same connected component in the original graph. After information is exchanged across different partitions, some small connected components will start to merge into a larger connected component.

Exploiting the above property, superstep 0 first runs a sequential connected component algorithm on the subgraph of each graph partition and then sends the locally computed component label for each boundary vertex to its corresponding owner's internal vertex. For the example in Figure 4.1(a), superstep 0 finds one connected component in the subgraph G1 and assigns the smallest label A to all its vertices including the boundary vertex D. Similarly, one connected component with label C is detected in G2. Messages with the component labels are then sent to the owners of the boundary vertices. In each of the subsequent supersteps, the algorithm processes all the incoming messages and uses them to find out which component labels actually represent

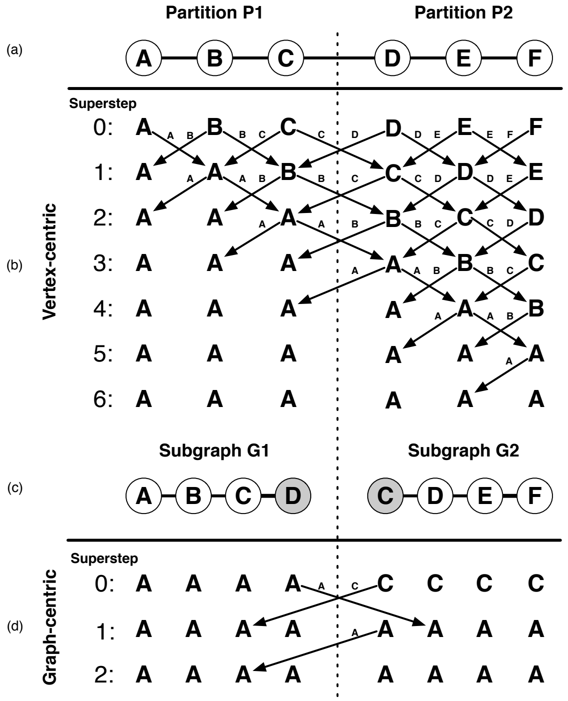


Figure 4.1: Example execution of connected component algorithms in vertex-centric and graph-centric models

equivalent components (i.e. they will be merged into a larger component) and stores them in a data structure called equiCC. In the above example, vertex D in superstep 1 receives the message A from G1, while its previous component label is C. Thus, pair (A, C) is put into equiCC to indicate that the connected components labeled A and C need to be merged. In equiCC.*consolidate*() function, we use the smallest label as the unique label for the set of all equivalent components. In our example, the new label for the merged components should be A. Then the unique labels are used to update the component labels of all the vertices in the partition. If a boundary vertex's component label is changed, then a message is sent to its owner's corresponding internal vertex. Comparing the two algorithms illustrated in Figure 4.1(b) and 4.1(d), the graph-centric algorithm needs substantially fewer messages and supersteps. In superstep 0, all the vertices in P1 already converge to their final labels. It only takes another 2 supersteps for the whole graph to converge.

The graph-centric programming model in Giraph++ is more general and flexible than the vertex-centric model. The graph-centric model can mimic the vertex-centric model by simply iterating through all the active internal vertices and performing vertex-oriented computation. In other words, any algorithm that can be implemented in the vertex-centric model can also be implemented in the graph-centric model. However, the performance of some algorithms can substantially benefit from the graph-centric model.

The graph-centric programming model is not intended to replace the existing vertex-centric model. Both models can be implemented in the same system as demonstrated in Giraph++. The vertex-centric model has its simplicity. However, the graph-centric model allows lower level access, often needed to implement important algorithm-specific optimizations. At the same time, the graph-centric model still provides sufficiently high level of abstraction and is much easier to use than, for example, MPI.

4.1.2 Blogel

Programming Interface. In addition to the *Vertex* base class, Blogel has another base class, *Block*, which takes the user-defined vertex subclass as a template argument indicating the data type of vertices in a block. Like a vertex object, a block object b has a flag $active(b)$ and can vote to halt; b can also maintain its own attributes, such as a value $a(b)$ and a block-level adjacency list $\Gamma(b)$ that links to other blocks. In addition, b can access an array of its own vertices, which is actually a subarray of the vertex array maintained by the worker that b resides in.

Both a vertex object and a block object can send two types of messages: those towards another vertex, and those towards another block. Accordingly, two message combiners can be specified, each for combining one type of messages. The *Block* class also has a UDF $compute(.)$, to be called by each block during the computation, whose input are those messages towards the current block.

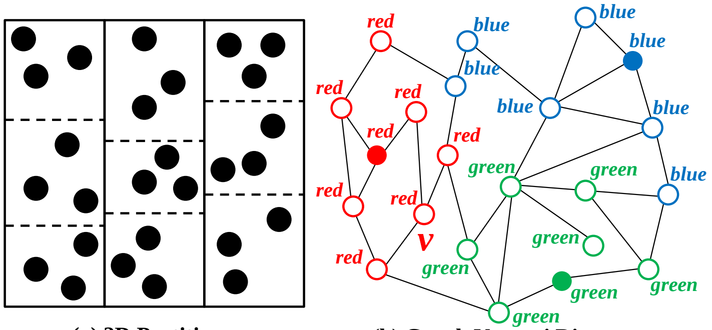
After a worker loads its vertices from HDFS, it groups them by their block ID to construct an array of block objects automatically. Before computation begins, each block b calls another UDF *block_init*(), which specifies how to initialize the attribute of b, e.g., from its vertices.

Execution Modes. Three execution modes are supported in Blogel: (1) **V-mode**, which is exactly like the vertex-centric model, but is usually more efficient since vertices are grouped into blocks with strong cohesion, and messages transmitted between two vertices in the same block do not incur network communication; (2) **B-mode**, where only blocks call *compute*(.) functions and message passing only happens among blocks, and computation terminates when all blocks are inactive and there are no pending messages; (3) **VB-mode**, where in a superstep, all active vertices call *Vertex::compute*(.) first, and then all active blocks call *Block::compute*(.), and computation terminates when all vertices and blocks are inactive and there are no pending messages.

We illustrate how to write a B-mode algorithm by considering Hash-Min. Instead of letting vertices broadcast the smallest vertex ID they have seen, we let blocks broadcast the smallest block ID they have seen. In b.*block_init*(), we construct $\Gamma(b)$ from $\Gamma(v)$ of every vertex in b as follows: if $u \in \Gamma(v)$, then we add the block of u (available in u's ID) to $\Gamma(b)$. When reporting results, each vertex assigns its value $a(v)$ with the value of its block b, i.e., $a(b)$. Since vertices in a block are all connected, all vertices with the same value constitute a connected component. This B-mode algorithm is more efficient than the vertex-centric Hash-Min algorithm since there are much less blocks than vertices.

While B-mode algorithms are not supported by Giraph++, VB-mode algorithms have a similar flavor to *GraphPartition.compute*(.) in Giraph++, except that a VB-mode algorithm can separate the computing logic related to blocks and vertices to *Block::compute*(.) and *Vertex::compute*(.), respectively.

As an illustration of VB-mode, consider the computation of single-source shortest paths from a source vertex s, where we denote the weight of each edge (u, v) by $w(u, v)$. In this algorithm, the vertex value



(a) 2D Partitioner (b) Graph Voronoi Diagram

Figure 4.2: Blogel Partitioners

$a(v)$ keeps an estimated distance from s to v, and messages are only sent to vertices. In a superstep, (1) *Vertex::compute*(.) is first called by vertices that receive messages, where each vertex v receives the distance estimations from its neighbors in other blocks. If the smallest message is less than $a(v)$, v updates $a(v)$ as the new value and remains active. Otherwise, v votes to halt. Then, (2) *Block::compute*(.) is called by blocks that contain active vertices. This does not need user intervention since Blogel also activates the block of a vertex v when v is activated. In $b.compute(.)$, b first collects all its active vertices into a priority queue Q (since their values are updated), and votes them to halt (thus all vertices are halted at the end of a superstep). Then, b runs Dijkstra's algorithm, where a vertex v with the smallest value $a(v)$ is popped from Q each time, and for each neighbor $u \in \Gamma_{out}(v)$, (i) if u is in b, $a(u)$ is updated with $(a(v) + w(v, u))$ and u is added to Q (or u's position in Q is properly adjusted), while (ii) if u is not in b, a message $(a(v) + w(v, u))$ is sent to u.

Graph Partitioning. Blogel supports three kinds of partitioners for processing a graph data, whose output can be used as the input data of a block-centric program. The first one is URL partitioner, which is only applicable to web graphs where each vertex contains a URL. The URL partitioner groups vertices under the same host name or domain name into one block. The second one is 2D partitioner, which is only applicable to spatial graphs where vertices have 2D coordinates. The 2D partitioner first samples a small fraction of vertices, which are sent to the master to construct a balanced spatial partitioning as Figure 4.2(a) illustrates. The spatial partitioning is broadcast back to every worker, which then assigns each of its vertex to a proper partition. The workers then exchange vertices with each other, so that each worker W contains all vertices in those partitions that are assigned to W. Since we require a block to be connected, each worker then runs BFS in each of its partitions, to get connected subgraph blocks.

For a general graph, Blogel provides a graph Voronoi diagram (GVD) partitioner for computing blocks. Since the partitioning algorithm only works for undirected graphs, a directed graph will be converted into an undirected one before partitioning. This conversion is simple in vertex-centric model: in Step 1, every vertex v sends its own ID to every out-neighbor $u \in \Gamma_{out}(v)$; in Step 2, every vertex u collects all messages to form $\Gamma_{in}(u)$. Finally, the neighbors of v in the converted graph is obtained as $\Gamma(v) = \Gamma_{out}(v) \cap \Gamma_{in}(v)$.

The idea of GVD partitioning is illustrated in Figure 4.2(b). Firstly, a small fraction of vertices are sampled as sources (solid circles in Figure 4.2(b)). Then, multi-source breadth-first traversal is performed from the sampled sources in vertex-centric model: the vertex value $a(v)$ indicates the source that v is assigned to, and in $v.compute(.)$ (of the GVD partitioner), if (1) $a(v)$ is already assigned, v votes to halt directly; (2) otherwise, v assigns $a(v)$ with the first received source ID, and broadcast it before voting to halt. The multi-source traversal is fast since each vertex broadcasts messages for at most once, and $a(v)$ equals the source closest to v. For example, the vertices in Figure 4.2(b) are grouped into three blocks, represented by three different colors.

However, since sources are randomly sampled, some blocks may contain too many vertices. The GVD partitioner then marks the states of those vertices back as “unassigned”, and samples sources from the remaining unassigned vertices with an increased sampling probability. Then, multi-source BFS is performed among the unassigned vertices again to obtain new blocks. This process is repeated until some quality requirement is met. Finally, Hash-Min is run on all unassigned vertices, and each connected component is treated as a block. The last step is necessary, since a graph may contain many small connected components, and it is likely that none of the vertices in a component is ever sampled due to low sampling probabilities.

Experiments in [Yan et al., 2014a] demonstrate that the GVD partitioner scales almost linearly with the graph size, and the partitioning time is comparable to the time for loading the input graph from HDFS and dumping the partitioned graph to HDFS.

4.2 Asynchronous Execution

The BSP model of Pregel has two limitations in performance:

(1) for algorithms where vertex values converge asymmetrically, execution priority cannot be given to those vertices that converge more slowly;

(2) local messages and remote messages are processed with equal priority (or frequency), i.e., once per superstep, although their transmission speeds are vastly different; to make things worse, the frequency (i.e. the time of a superstep) depends on the slowest worker.

Two asynchronous message-passing systems were developed to solve the above limitations. Maiter [Zhang et al., 2014] adopts a novel asynchronous programming model, which has more limited expressive power than Pregel but has a strict guarantee on correctness, and supports prioritized vertex-centric computation, which solves Limitation (1). GiraphUC [Han and Daudjee, 2015] does not change the programming model of Pregel, but allows local messages to be immediately processed within each superstep, which solves Limitation (2). However, GiraphUC might not generate exactly the same results as Pregel does for some algorithms like PageRank computation, which, on the other hand, is guaranteed by Maiter. We introduce Maiter and GiraphUC in the next two subsections.

4.2.1 Maiter

Maiter [Zhang et al., 2014] proposed a new computation and programming model called *delta-based accumulative iterative computation* (DAIC), which iteratively updates the vertex values by accumulating the value changes between iterations. Since updates are computed from value changes rather than vertex values themselves, DAIC enables asymmetric vertex-centric execution prioritized by value changes.

The DAIC Model. We now present DAIC, and illustrate its idea using PageRank computation as a running example. DAIC requires a vertex-centric algorithm to be formulated into the following 2-step update function:

$$\begin{cases} a(v)^{(i)} = & a(v)^{(i-1)} \oplus \Delta a(v)^{(i)} \\ \Delta a(v)^{(i+1)} = & \bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)}(\Delta a(u)^{(i)}) \end{cases}, \tag{4.1}$$

where $a(v)^{(i)}$ denotes the value of vertex v at the i-th iteration, and $\oplus$ is a generalized summation operator that is commutative and associative. The first equation states that $\Delta a(v)^{(i)}$ is the value change from $a(v)^{(i-1)}$ to $a(v)^{(i)}$, and the second equation states that this change $\Delta a(v)^{(i)}$ can be computed from the value changes of v's in-neighbors in the $(i-1)$-th iteration.

To write a DAIC algorithm, the update function should satisfy two conditions. The first condition is that, the value update function can be formulated into the following form:

$$a(v)^{(i+1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)}(a(u)^{(i)}) \right) \oplus c(v), \tag{4.2}$$

where $c(v)$ is a constant associated with v. Note that Equation (4.2) operates on vertex values rather than value changes, and it also defines the formula of $g_{(u,v)}(.)$ as required by Equation (4.1).

For example, the UDF $v.compute(.)$ in PageRank computation has the following form:

$$a(v)^{(i+1)} = 0.85 \cdot \left(\sum_{u \in \Gamma_{in}(v)} \frac{a(u)^{(i)}}{d_{out}(u)} \right) + 0.15,$$

and therefore, it can be formulated into the form of Equation (4.2) if we specify $\oplus$ as the summation operator, and specify

$$\begin{cases} g_{(u,v)}(x) = 0.85 \cdot \frac{x}{d_{out}(u)} \\ c(v) = 0.15 \end{cases}, \tag{4.3}$$

The second condition is that, $g_{(u,v)}(x)$ should have the distributive property over $\oplus$:

$$g_{(u,v)}(x \oplus y) = g_{(u,v)}(x) \oplus g_{(u,v)}(y), \tag{4.4}$$

which is obviously satisfied by $g_{(u,v)}(x)$ of Equation (4.3).

We now show that Equations (4.2) and (4.4) guarantee the correctness of applying Equation (4.1) for computation. By replacing $a(u)^{(i)}$ in Equation (4.2) with $(a(u)^{(i-1)} \oplus \Delta a(u)^{(i)})$, and using Equation (4.4), we obtain

$$\begin{aligned} a(v)^{(i+1)} &= \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(i-1)} \oplus \Delta a(u)^{(i)} \right) \right) \oplus c(v) \\ &= \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(\Delta a(u)^{(i)} \right) \right) \oplus \\ & \quad \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(i-1)} \right) \right) \oplus c(v), \end{aligned}$$

and using Equations (4.2), we obtain

$$a(v)^{(i+1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(\Delta a(u)^{(i)} \right) \right) \oplus a(v)^{(i)},$$

or equivalently,

$$\Delta a(v)^{(i+1)} = \bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(\Delta a(u)^{(i)} \right),$$

which is consistent with Equation (4.1).

Finally, $a(v)^{(0)}$ and $\Delta a(v)^{(1)}$ should be initialized so that

$$a(v)^{(0)} \oplus \Delta a(v)^{(1)} = a(v)^{(1)} = \left(\bigoplus_{u \in \Gamma_{in}(v)} g_{(u,v)} \left(a(u)^{(0)} \right) \right) \oplus c(v).$$

For example, in PageRank computation, we may set $a(v)^{(0)} = 0$ and $\Delta a(v)^{(1)} = 0.15$.

The DAIC model is expressive enough to represent many other Pregel algorithms. For example, Hash-Min can be formulated as DAIC by specifying $\oplus$ as taking the minimum, $g_{(u,v)}(x) = x$, $a(v)^{(0)} = \infty$ and $\Delta a(v)^{(1)} = v$.

Asynchronous Execution. [Zhang et al., 2014] proves that Equation (4.1) is equivalent to the following asynchronous operations. Specifically, each vertex v maintains two fields $a(v)$ and $\Delta a(v)$. Whenever a delta message $m = g_{(u,v)}(\Delta a(u))$ sent from v's in-neighbor u is received by v, v sets $\Delta a(v) \leftarrow \Delta a(v) \oplus m$. When v performs computation, it (1) sets $a(v) \leftarrow a(v) \oplus \Delta a(v)$, (2) sends $g_{(v,w)}(\Delta a(v))$ to each out-neighbor $w \in \Gamma_{out}(v)$ (if it is not 0), and (3) clears $\Delta a(v)$ back to 0. Note that 0 here refers to the identity element of the $\oplus$ operator (i.e., $x \oplus 0 = x$). The only race-condition is that the update to $\Delta a(v)$ should be atomic.

As an illustration using our PageRank example, whenever a vertex v receives a delta message m, it will add it to $\Delta a(v)$. When v performs computation, it adjusts the received cumulative delta value $\Delta a(v)$, by multiplying it with $0.85/d_{out}(v)$, which is then broadcast to every out-neighbor u (to be added to $\Delta a(u)$); then, v clears the processed cumulative delta value by setting $\Delta a(v)$ to 0, for accumulating more delta values.

Maiter supports prioritized execution. For example, in PageRank computation, each worker may choose the top-1% vertices with the highest $\Delta a(v)$ for vertex-centric computation at each time, and repeat this operation until the terminate condition holds. The master periodically broadcasts a progress request signal to all workers asking for the convergence progress of each worker, and makes a global termination decision based on the responses. Moreover, to send messages in relatively large batches, a worker buffers the outgoing messages and flushes them after a timeout.

Since Maiter adopts asynchronous execution, to be fault-tolerant, it exploits Chandy-Lamport snapshot [Chandy and Lamport, 1985] for checkpointing, which we introduced in Section 3.6.1. However, we remark that Maiter is a message-passing system and is thus different from asynchronous shared-memory abstractions like GraphLab. In fact, Maiter is the only message-passing system that guarantees the exactness of PageRank results with asynchronous execution, i.e., the final rank values of all vertices remain unchanged. This, however, may not be guaranteed by the GiraphUC system to be presented in the next subsection.

The DAIC model also makes it very suitable for block-centric computation. For example, consider PageRank computation. In $b.compute(.)$, when the value of a vertex v in block b is updated, for every out-neighbor $w \in \Gamma_{out}(v)$, (1) if w is not in b, a delta message $g_{(v,w)}(\Delta a(v))$ is sent to w; (2) otherwise, $g_{(v,w)}(\Delta a(v))$ is directly added to $\Delta a(w)$, which will be processed later when $b.compute(.)$ processes u. In fact, this algorithm has been implemented in Giraph++ [Tian et al., 2013].

4.2.2 GiraphUC

GiraphUC [Han and Daudjee, 2015] proposed a new computation model called *barrierless asynchronous parallel (BAP)*, which (1) reduces message staleness by using the latest received message of an in-neighbor rather than that from the previous global superstep, which allows faster convergence; and (2) allows computation of a worker to proceed without synchronizing with other workers as long as new messages are received, which prevents fast workers from blocking and waiting for the stragglers. However, the programming model of GiraphUC remains exactly the same as that of Pregel.

The BAP model works as follows. Each worker W calls $compute(.)$ on all active vertices once, which accomplishes a *local superstep* and enters a *local barrier*. At the local barrier, W performs graph mutations, and decides whether it should wait on a global synchronization or start another local superstep.

A worker W waits on a global synchronization only if there are no more incoming messages to process, and all local vertices are inactive. This condition can be further simplified to check whether there are no remote messages (i.e., messages from other workers) since if there is any local message, it will reactivate a local vertex. However, if W receives any remote message during the waiting stage, W unblocks itself to start another local superstep that processes the new incoming messages. A global synchronization is performed only when all workers are in the waiting stage, which accomplishes a *global superstep*.

In $v.compute(.)$, if v requires the message value of an in-neighbor $u \in \Gamma_{in}(v)$, the latest received message value is used. Note that the value of a message may still be stale: the latest message from an in-neighbor u (on another worker) may still be in transmission, in which case the latest received message used by $v.compute(.)$ is stale. The idea of computing with stale values to allow faster convergence has also been used by a machine learning system called parameter server [Ho et al., 2013], where the computation model is called *Stale Synchronous Parallel* (SSP).

The results of BAP are guaranteed to be the same as those of BSP in algorithms where $compute(.)$ does not require messages from all in-neighbors, or more precisely, the state of every vertex can only change monotonically towards the convergence value after processing each individual message separately. For example, in Hash-Min, as long as v receives a message smaller than $a(v)$, it updates $a(v)$ with the message and broadcasts the new $a(v)$ to all neighbors. A stale message only slows down the convergence rate of Hash-Min, but the results are not influenced.

While [Han and Daudjee, 2015] claims that BAP also guarantees the result exactness for algorithms where $compute(.)$ requires messages from all in-neighbors, this may not always hold. For example, in PageRank computation, v needs to sum up messages from all its in-neighbors. If an in-neighbor u is on another worker and its latest message has not been received, then the summation computed with the stale message of u is different from the exact summation.

To support multi-phase algorithms where $compute(.)$ has different logic in different phases, messages sent in different phases should not be mixed. For this purpose, GiraphUC tags each message to indicate whether the message targets at the current phase, or at the next phase. Two message stores are maintained, one to receive messages of the current phase (denoted by M_c), and the other to receive messages of the next phase (denoted by M_n). In the current phase, only messages in M_c are used; when a new phase begins, the old M_n becomes the new M_c, and the new M_n becomes empty.

4.3 Vertex-Centric Query Processing

In this section, we introduce the Quegel system [Yan et al., 2016b] which adapts the computation model of Pregel for answering lightweight online queries efficiently. A Quegel user only needs to specify the Pregel-like algorithm for a generic query, and Quegel processes light-workload graph queries on demand by effectively utilizing the cluster resources. Quegel also provides a convenient interface for constructing graph indexes, which are not supported by other big graph systems, and which significantly reduce the querying workload.

Motivation. Most Pregel-like systems are designed for offline graph analytics, where a job visits most (if not all) vertices in a big graph for many iterations. However, there is also a need to answer graph queries online, where each query usually accesses only a small fraction of vertices during the whole period of evaluation. However, offline analytics systems cannot support query processing efficiently, nor do they provide a user-friendly programming interface to do so, which we explain below.

If we write a vertex-centric algorithm for a generic query, we have to run an independent job for each incoming query. In this solution, each superstep transmits only the few messages of one light-weight query and cannot fully utilize the network bandwidth. Also, there are a lot of synchronization barriers, one for each superstep of each query. Moreover, some systems such as Giraph bind graph loading with graph computation (i.e., processing a query in our context) for each job, and the loading time can significantly degrade the performance.

An alternative solution is to hard code a vertex-centric algorithm to process a batch of k queries, where k can be an input argument. However, in the *compute*(.) function, one has to differentiate the incoming messages and/or aggregators of different queries and update

k vertex values accordingly. In addition, existing vertex-centric framework checks the stop condition for the whole job, and users need to take care of additional details such as when a vertex can be deactivated (e.g., when it should be halted for all the k queries), which should originally be handled by the system itself. Last but not least, this approach does not solve the problem of low utilization of network bandwidth, since in later stage when most queries finish their processing, only a small number of queries (or stragglers) are still being processed and hence the number of messages generated is too small to sufficiently utilize the network bandwidth.

Execution Model. Quegel solves the above problems by adopting a superstep-sharing execution model. Specifically, Quegel processes graph queries in iterations called *super-rounds*. In a super-round, every query that is currently being processed proceeds its computation by one superstep; while from the perspective of an individual query, Quegel processes it superstep by superstep as in Pregel. Intuitively, a super-round in Quegel is like *many queries sharing the same superstep*. For a query q whose computation takes n_q supersteps, Quegel processes it in (n_q+1) super-rounds, where the last super-round prints the results of q on the console or dumps them to HDFS.

Quegel allows users to specify a capacity parameter C, so that in any super-round, there are at most C queries being processed. New incoming queries are appended to a query queue (in the master), and at the beginning of a super-round, Quegel fetches as many queries from the queue as possible to start their processing, as long as the capacity constraint C permits. During the computation of a super-round, different workers run in parallel, while each worker processes (its part of) the evaluation of the queries serially. And for each query q, if q has not been evaluated, a worker serially calls *compute*(.) on each of its vertices that are activated by q; while if q has already finished its evaluation, the worker reports or dumps the query result, and releases the resources consumed by q.

For the processing of each query, the supersteps are numbered. Two queries that enter the system in different super-rounds have different superstep number in any super-round. Messages (and aggregators) of

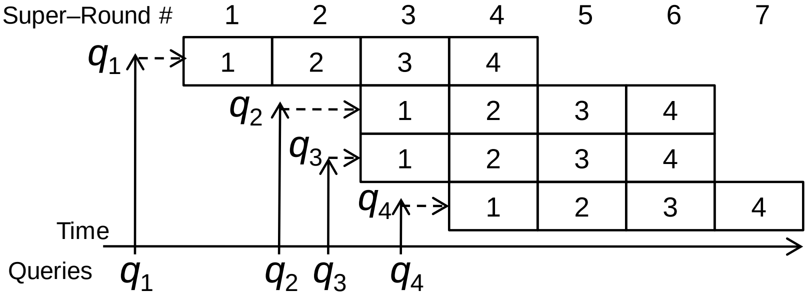


Figure 4.3: Illustration of Superstep-Sharing

all queries are synchronized together at the end of a super-round, to be used by the next super-round. Figure 4.3 illustrates the execution of four queries q_1, q_2, q_3 and q_4 in our superstep-sharing model, where we assume for simplicity that every query takes 4 supersteps.

System Design. Quegel manages three kinds of data: (1) *V-data*, whose value only depends on a vertex v, such as v's adjacency list; (2) *VQ-data*, whose value depends on both a vertex v and a query q, including $a(v)$, $active(v)$, and v's incoming message queue; (3) *Q-data*, whose value only depends on a query q, such as query content, superstep number, aggregator, and control information. Q-data are maintained by every worker, and kept consistent at the beginning of each superstep (e.g., by creating Q-data for new incoming queries). Each vertex maintains its V-data, and a table of VQ-data for queries in processing.

A Quegel user writes vertex-centric programs exactly like in Pregel, and the processing of concrete queries is transparent to users. For example, a user may access $a(v)$ (resp. superstep number) in $v.compute(.)$, and if a worker is processing v for query q, the VQ-data of v and q (resp. the Q-data of q) is actually accessed.

To be space efficient, a vertex v allocates a state (i.e., VQ-data) for a query q only if q accesses v during its processing. Specifically, when vertex v is activated for the first time during the processing of q, the VQ-data of q is initialized and inserted into the VQ-data table of v. After a query q reports or dumps its results at superstep $(n_q + 1)$, the

VQ-data of q is removed from the VQ-data table of every vertex that has been accessed by q.

Graph queries usually start from a few querying vertices. For example, a point-to-point shortest path query (v_s, v_t) finds the shortest path from vertex v_s to vertex v_t, and can be evaluated by performing bidirectional breadth-first search from s and t. Quegel allows a user to activate a few vertices efficiently without scanning all vertices, which is performed when a query is fetched from the query queue for processing.

Quegel also provides a convenient API for users to build distributed graph indices. After a worker loads those vertices that are assigned to it from HDFS, the worker may build a local index from its vertices using the user-defined logic before Quegel starts to process queries. For example, for graph matching and graph keyword search, the algorithms usually start from those vertices whose text attribute values contain the required labels or keywords. By building a local inverted index that maps each label (or keyword) to those vertices in the worker that contains the label (or keyword), the initial vertices can be activated by looking up the index, eliminating the need to check the text attribute of every vertex for an incoming query.

4.4 Summary

This chapter reviewed three variants of the basic vertex-centric model, to account for the features of real graphs (e.g., large-diameter and relatively high density) and algorithms (e.g., asymmetric value convergence), and to meet the need of various applications (e.g., on-demand querying). These systems demonstrate the need to have a rich toolkit of different graph analytics frameworks, each is optimized at solving a particular class of graph problems. Although users need to get familiar with several computation models rather than just a unified vertex-centric framework like Pregel, the introduced systems are all user-friendly to programmers familiar with vertex-centric interfaces, and the performance improvement can often be orders of magnitude.

5

Vertex-Centric Systems with Shared Memory Abstraction

In this chapter, we review another class of vertex-centric big graph systems, which adopt the shared memory programming abstraction instead of message passing for exchanging information among the vertices. We first review the pioneering distributed system GraphLab [Low et al., 2012], as well as its successor, PowerGraph [Gonzalez et al., 2012]. We then introduce four single-machine systems that adopt shared memory abstraction: GraphChi [Kyrola et al., 2012], X-Stream [Roy et al., 2013], VENUS [Cheng et al., 2015] and GridGraph [Zhu et al., 2015]. These systems are designed to perform out-of-core graph processing on a single PC, but they adopt different execution models.

In a system that adopts shared memory programming abstraction, a vertex v directly accesses the data values of its adjacent vertices and edges, rather than passively receiving messages pushed by other vertices like in Pregel. The shared memory abstraction does not mean that the underlying execution is shared memory (such as PRAM), but rather the execution is often distributed or out-of-core, and thus a vertex cannot access the value of a non-neighbor even if it knows the ID of that vertex. As a result, these systems cannot support algorithms that use pointer jumping (e.g., those described in Section 3.2).

We denote the value of a vertex v by D_v, and the value of an edge (u, v) by $D_{(u,v)}$. A vertex v can access the following five types of data, where we regard u (and w) as an in-neighbor (and an out-neighbor) of v:

1. D_u for any $u \in \Gamma_{in}(v)$;
2. $D_{(u,v)}$ for any $u \in \Gamma_{in}(v)$;
3. D_v;
4. $D_{(v,w)}$ for any $w \in \Gamma_{out}(v)$;
5. D_w for any $w \in \Gamma_{out}(v)$.

We call the set of these values as the *scope* (or *full-scope*) of vertex v. We also defined two other scopes: (1) *value-scope*, where a vertex v can only access D_u, D_v and D_w; and (2) *edge-scope*, where a vertex v can only access $D_{(u,v)}$, D_v and $D_{(v,w)}$. Full scope access is supported by GraphLab and PowerGraph, while the two more restricted scopes are adopted by single-machine systems that emerge later.

5.1 Distributed Systems with Shared Memory Abstraction

We now review the GraphLab system, which pioneered the shared-memory programming abstraction for distributed graph processing.

5.1.1 Distributed GraphLab

GraphLab was originally developed for a single machine, and was extended to a distributed setting in 2012 [Low et al., 2012]. Distributed GraphLab keeps the shared memory programming abstraction of a single-machine environment, where a vertex can access its full scope (including data of adjacent vertices and edges).

Computation and Programming Model. Since GraphLab is originally proposed for writing machine learning algorithms, it adopts vertex scheduling to favor iterative algorithms where vertex values converge asymmetrically. Specifically, a scheduler (FIFO or prioritized) is employed to schedule the computation of vertices in each machine. Similar to *compute*(.) in Pregel, a GraphLab program requires a user to specify a UDF *update*() to be called by a vertex v. In $v.update()$, v may read and update the values in its scope, and submit any of the vertices in its scope to the scheduler for further processing. The execution in GraphLab guarantees serializability, i.e., the parallel computation has an equivalent sequential execution counterpart, where the scheduler calls the *update*() function on the vertex with the highest priority at a time, and more vertices may be submitted to the scheduler for execution during each *update*() call.

To illustrate how to write *update*(), we consider PageRank computation. In $v.update()$, v directly reads D_u of all $u \in \Gamma_{in}(v)$ to compute its new PageRank D_v^{new} and checks whether $|D_v^{new} - D_v|$ is larger than a convergence threshold ϵ. If not, D_v is considered as converged and there is nothing to do; otherwise, v updates $D_v \leftarrow D_v^{new}$ and adds all out-neighbors to the scheduler for further processing. This algorithm is much faster than the PageRank algorithm of Pregel, since more and more vertices have their PageRank values converge during the execution, and thus, fewer and fewer vertices call the *update*() function[1].

GraphLab adopts an asynchronous computation model and hides network communication from its programming model. This is in contrast to the bulk synchronous parallel (BSP) model of Pregel, where users explicitly send messages in their programs, and messages are synchronized (i.e., transmitted) in batches among all workers in every superstep. However, the asynchronous model requires additional effort to enforce data consistency under race conditions (e.g., by using locks), and it does not guarantee exactness of results nor determinism for algorithms like PageRank. In fact, GraphLab has a synchronous mode that simulates the computation model of Pregel, and both [Lu et al., 2014] and [Han et al., 2014a] found that the synchronous mode is faster than

[1] Arguably, one may also simply vote a vertex v to halt in Pregel if v's PageRank changes by less than ϵ; both this method and that of GraphLab's may omit the transmission of some small values, leading to approximate results

the asynchronous mode for algorithms where asymmetric convergence does not help (e.g., Hash-Min).

Data Organization & Communication Model. While GraphLab adopts a shared memory abstraction, network communication is inevitable and GraphLab achieves transparent communication by synchronizing overlapped data values across the cluster. Specifically, an input graph is first partitioned among different worker machines, where each worker W is assigned a subset of vertices V_W along with their adjacent edges. The partitioning can be done either by hashing, or by more expensive algorithms like ParMETIS [Karypis and Kumar, 1998]. In practice, GraphLab over-partitions a graph into many more partitions (called *atoms*) than the number of workers, so that these partitions can be evenly distributed to worker machines in a cluster of arbitrary size, avoiding the need of repartitioning.

Now consider an edge (u, v), where u is assigned to a worker W_u while v is assigned to another worker W_v. In this case, both W_u and W_v need to maintain D_u, $D_{(u,v)}$ and D_v, which belong to the set of overlapped data values between W_u and W_v. The overlapped data values are called *ghosts*, and if a worker updates a data value in its ghosts, the update needs to be synchronized with the corresponding data values in other workers using a versioning system.

This data organization has some weaknesses in scalability, which we analyze next. For each partition, it is desirable to reduce the overlap with other partitions (i.e., reduce the number of ghosts) since those data values need to be replicated and synchronized. However, computing a high-quality partitioning (e.g., using ParMetis [Karypis and Kumar, 1998]) for a big graph is very expensive. On the other hand, using hashing leads to a large amount of overlap: suppose that there are $|\mathbb{W}|$ workers, then for an edge (u, v), the probability that u and v are on the same worker is only $1/|\mathbb{W}|$; in other words, approximately $(1 - 1/|\mathbb{W}|)$ fraction of data are in the ghosts in expectation. Consider a vertex u on W_u whose out-degree is much higher than $|\mathbb{W}|$; the out-neighbors of u may be spread across almost all workers, and thus D_u may need to be replicated $|\mathbb{W}|$ times. The high replication factor limits the scalability of

GraphLab. For example, in [Low et al., 2012], the largest graph tested has merely 200M edges, which is too small for distributed processing.

Fault Tolerance. GraphLab supports two checkpointing-based mechanisms for fault tolerance: (1) it can construct synchronous snapshots (i.e., checkpoints) as in Pregel, where computation is suspended during the construction of a snapshot; or (2) it can incrementally construct such snapshots without suspending execution. The second mechanism adapts the Chandy-Lamport snapshot technique described in Section 3.6.1 to the shared-memory abstraction, which treats the snapshotting of a vertex's scope as an independent operation just like the calling of *update*() on a vertex. When a vertex v snapshots its scope, it adds all its neighbors into the scheduler to be snapshotted before any UDF *update*() is called.

5.1.2 PowerGraph: the GAS Model

Motivation of the GAS Model. GraphLab soon replaced its vertex-centric programming model by a GAS programming model which associates UDFs with the adjacent edges of each vertex, in a subsequent version called PowerGraph [Gonzalez et al., 2012]. This is motivated by the following observation.

Many natural graphs follow a power-law like vertex degree distribution, and hence often contain some vertices with very high degree (e.g., having millions of neighbors), while most vertices have only a few neighbors. An example is the Twitter who-follows-who graph where celebrities are followed by many others (i.e., in-neighbors). In Pregel, a high-degree vertex v needs to compute and send many more messages to its neighbors than an average vertex; while in distributed GraphLab, $v.update()$ needs to process all its neighbors in v's scope, leading to a much larger workload than that of an average vertex. Due to the large workload variance associated with different vertices, partitioning vertices among workers tends to lead to an unbalanced workload distribution. Partitioning edges among workers solves this problem, since the edges (and the associated computation) of a high-degree vertex can be distributed among multiple workers.

The GAS Model. PowerGraph adopts a Gather-Apply-Scatter (GAS) model for both programming and computation. Put simply, a vertex v first gathers values along adjacent edges, and aggregates these values to compute its vertex value $a(v)$, and then scatters value updates along adjacent edges.

Four UDFs need to be specified:

1. *gather*(u, v), which is invoked on each edge adjacent to v to compute a value towards v, and has access to D_u, $D_{(u,v)}$, and D_v;
2. *sum*(*combined*, *value*), which accumulates the value along a local edge to a locally combined value;
3. *apply*(D_v, *combined*), which uses the globally combined value and the old value of D_v to compute a new value for D_v; and
4. *scatter*(v, u), which is invoked on each edge adjacent to v (e.g., to activate the neighbor u or to update $D_{(v,u)}$), and has access to the updated D_v, and $D_{(v,u)}$ and D_u.

We illustrate how to program with the GAS model using PageRank computation: (i) *gather*(u, v) simply returns $a(u)/d_{out}(u)$ where both $a(u)$ and $d_{out}(u)$ are accessed from D_u; (ii) *sum*(.) simply sums the values returned by *gather*(.); (iii) *apply*(.) adjusts the summation by a damping factor (e.g., 0.85) to compute a new value for $a(v)$, and the difference $\Delta a(v)$ between the old and new values; (iv) *scatter*(v, u) activates out-neighbor u only if $\Delta a(v) > \epsilon$.

When a vertex v performs computation, it needs to gather values from every neighbor, but it is possible that only one neighbor u has updated its data D_u. For example, in the late stages of Hash-Min, the values of most vertices are usually converged. PowerGraph uses *delta caching* to avoid redundant gather operations. Specifically, PowerGraph caches the globally combined values from the previous gather phase for each vertex. The UDF *scatter*(u, v) can optionally return $\Delta a(v)$ to be atomically added to the cached value for v, so that v can bypass the gather phase and call *apply*(.) with the cached value directly.

Edge Partitioning. PowerGraph evenly partitions the edges of a graph among the workers in a cluster. However, since the edges of a vertex v may be distributed to different workers, v (and thus D_v) may span multiple machines which incurs synchronization overhead. There are two goals for edge partitioning: (1) load balancing, i.e., to distribute edges uniformly over worker machines; (2) to minimize the total number of vertex replicas. We now formalize the second goal. Let $A(v)$ be the set of workers that contains at least one edge adjacent to v; then every worker in $A(v)$ maintains a replica of D_v. Goal (2) aims to minimize $N = \sum_{v \in V} |A(v)|$, and $N/|V|$ is called the *replication factor* (i.e., the size of $A(v)$ averaged over all $v \in V$). We say that Goal (2) minimizes the size of *vertex-cut*, since a vertex may span different machines and causes communication; this is in contrast to the more common vertex-partitioning approach where edges may span different machines and the goal is to minimize the size of *edge-cut* (or simply cut).

PowerGraph uses greedy edge placement rules to assign edges to different worker machines. Let us first assume that the edges are processed one after another. Assume that we are assigning the i-th edge (u, v) to a worker, and thus for any vertex w, $A(w)$ is obtained according to the assignment of the previous $(i - 1)$ edges. Then, the rules are given as follows, which correspond to three different cases:

- **Case 1: $\boldsymbol{A(u) \cap A(v) \neq \emptyset}$.** In this case, (u, v) is assigned to the least loaded worker in $A(u) \cap A(v)$.
- **Case 2: $\boldsymbol{A(u) \cup A(v) \neq \emptyset}$ and $\boldsymbol{A(u) \cap A(v) = \emptyset}$.** In this case, (u, v) is assigned to the least loaded worker in $A(u) \cup A(v)$.
- **Case 3: $\boldsymbol{A(u), A(v) = \emptyset}$.** In this case, (u, v) is assigned to the least loaded worker.

The actual partitioning of edges is performed in parallel by the workers, and each worker periodically coordinates with other workers to keep $A(v)$ relatively up to date, with some loss in the accuracy of estimating $A(v)$. Similar greedy strategies have also been studied and compared in the context of vertex partitioning by [Stanton and Kliot, 2012].

Notably, since the greedy rules reduce the replication factor, the ghost size becomes much smaller than in distributed GraphLab, and thus PowerGraph scales to much larger graphs. For example, the largest graph tested in [Gonzalez et al., 2012] contains 1.5B edges.

5.2 Out-of-Core Systems for a Single PC

While GraphLab is sometimes less efficient than Pregel-like systems in the distributed setting, the shared memory abstraction is dominantly adopted by single-machine big graph systems. This is because a single machine is a natural shared-memory environment, and the overhead in GraphLab, e.g., ghost state synchronization, remote locking, etc., is no longer a problem. In this section, we review four popular single-PC systems that adopt shared memory abstraction and support efficient out-of-core execution, and we analyze the disk-IO cost tradeoffs of their designs.

5.2.1 GraphChi

GraphChi [Kyrola et al., 2012] was proposed as a single-PC counterpart to distributed GraphLab, which keeps the GAS programming model but eliminates the requirement of a cluster of machines with large cumulative main memory space. Instead, GraphChi loads a disk-resident graph part by part into the main memory of a single PC for processing, and is efficient for moderate-sized graphs due to the elimination of network communication. However, since the disk-IO bandwidth of a single PC is fixed and limited, GraphChi is often an order of magnitude slower than distributed big graph systems [Lu et al., 2014] for very large graphs.

Edge-Scope GAS Programming. GraphChi adopts a simplified version of the GAS programming model, where a vertex v only has access to its *edge-scope* during computation, including (1) $D_{(u,v)}$ for all $u \in \Gamma_{in}(v)$, (2) D_v, and (3) $D_{(v,w)}$ for all $w \in \Gamma_{out}(v)$. However, for a broad range of vertex-centric graph algorithms, it is sufficient to only access the edge-scope of a vertex. For example, in PageRank computation, a vertex v may distribute $a(v)/d_{out}(v)$ to every out-edge (i.e., $D_{(u,v)} \leftarrow a(v)/d_{out}(v)$); and v may update $a(v)$ by summing the values from all in-edges, and adjusting it by a damping factor (e.g., 0.85).

To realize the above programming model, it is important to make sure that when v performs vertex-centric computation, the data values of v and all its in-edges and out-edges are in main memory. There are two challenges: (1) since the memory space of a single PC is limited, we cannot even assume that D_v of all vertices $v \in V$ can fit in memory; and (2) although $D_{(u,v)}$ will be written by u and read by v, we only want to keep one copy of $D_{(u,v)}$ on secondary storage to avoid expensive value synchronization.

Data Organization & Computation Model. We now explain how GraphChi achieves the above goals. Specifically, GraphChi requires that the IDs of the vertices in a graph with $|V|$ vertices should be numbered as 1, 2, ..., $|V|$. The IDs are partitioned into P disjoint intervals, I_1, ..., I_P. All vertices whose IDs fall into an interval I_i constitute a shard (we also denote the shard by I_i for simplicity). In GraphChi, each shard I_i stores not only (1) D_v of every $v \in I_i$, but also (2) all in-edges to vertices in I_i, i.e., (u, v) (along with $D_{(u,v)}$) for all $v \in I_i$ and $u \in \Gamma_{in}(v)$. The in-edges in a shard are stored in the order of their source u.

All vertices in a shard are loaded from secondary storage into main memory as one batch for computation. To load a shard I_i for processing, for any vertex $v \in I_i$, the data of v and its in-edges are already in main memory, but we still need to load the data of every out-edge (v, w) into memory. For this goal, GraphChi loads out-edges of vertices in I_i from every other shard I_j $(j \neq i)$ on secondary storage. Since in-edges (v, w) of all vertices $w \in I_j$ are already ordered by source v, all out-edges of vertices $v \in I_i$ are stored consecutively in I_j and can be loaded with only one sequential read. The updated edge values are then written back to the shard I_j by one sequential write. Therefore, processing a shard I_i takes one sequential loading of I_i and $(P - 1)$ sequential reads and writes of out-edges. GraphChi processes every shard once in each iteration, and thus requires $\Theta(P^2)$ non-sequential seeks to secondary storage.

If D_v of all vertices $v \in V$ can fit in memory, GraphChi supports a more efficient semi-streaming model, where a vertex v updates D_v by directly accessing D_u of every in-neighbor $u \in \Gamma_{in}(v)$, eliminating the need of transmitting values through adjacent edges. This optimization saves a lot of disk-IO cost, since otherwise $D_{(u,v)}$ of an edge (u, v) needs to be written by u's shard and read by v's shard. In fact, this model is extended to the case when D_v of all vertices $v \in V$ cannot fit in memory by VENUS [Cheng et al., 2015] to achieve better performance than GraphChi, which we will introduce in Section 5.2.3.

Shard Preparation. One drawback of GraphChi is the requirement of expensive preprocessing to prepare shards. Specifically, a user needs to first preprocess a graph to make sure that vertex IDs are numbered as 1, 2, . . ., $|V|$. Then, GraphChi takes one pass over the disk-resident graph to collect the in-degree of every vertex. Using the in-degree information, GraphChi then divides the vertices into P intervals with approximately the same number of in-edges, to form P shard files. In-edges in each shard file are sorted by source vertex. Finally, the in-degree and out-degree of every vertex are written to a binary degree file for later use.

So far, the preprocessing step guarantees that each shard file has roughly the same size. However, when loading a shard I_i into memory for processing, we still need to load out-edges of vertices in I_i from other shards, and if many vertices in I_i have high out-degree, the number of out-edges to load may exceed the memory space. To solve this problem, GraphChi further divides an interval into sub-intervals, using the information loaded from the degree file, so that each sub-interval can be processed in main memory. The use of sub-intervals allows the same set of shard files to be applicable to machines of different memory sizes.

Other Features. GraphChi also supports graph mutation. Specifically, each shard I_i has an edge-buffer B_i for appending new in-edges to vertices in I_i, while removed edges are simply flagged and ignored. After an iteration, each shard I_i is merged with its edge buffer B_i to form a new shard, and if the shard becomes too large, it will be split into two shards.

GraphChi adopts multithreading in two places. Firstly, the processing of different vertices in a shard I_i is performed in parallel. However, to solve race conditions, vertices that have an edge (u, v) where $u, v \in I_i$ are flagged as critical and updated in serial. Secondly, GraphChi overlaps disk operations and in-memory computation as much as possible, i.e., while one thread is loading/writing data, another thread is performing computation on a shard.

Finally, GraphChi supports selective scheduling so that vertices can be processed with different frequencies. Specifically, in an iteration, a vertex that performs computation can flag a neighboring vertex to be updated in the next iteration. A bitmap of all vertices is maintained for each iteration, and the bit corresponding to a vertex is set as 1 iff the vertex is flagged. Using the bitmap, GraphChi skips the computation on unflagged vertices.

5.2.2 X-Stream & Chaos

In this section, we first introduce X-Stream [Roy et al., 2013], a single-machine big graph system that follows a different design from GraphChi; we then introduce Chaos, a system that scales X-Stream out to run on multiple machines.

X-Stream is designed for both in-memory execution and out-of-core execution. According to Roy et al. [2013], X-Stream has very good performance for in-memory execution. For out-of-core execution, X-Stream does not require preprocessing, and the computation of a job finishes before GraphChi finishes shard preparation. Even excluding the preprocessing time of GraphChi, [Roy et al., 2013] reports that X-Stream is faster than GraphChi when SSD is used. However, when hard disk is used, X-Stream is usually slower than GraphChi, as reported by [Cheng et al., 2015] and [Yan et al., 2016d], possibly due to the lack of a mechanism to skip streaming inactive vertices.

Computation Model of X-Stream. Like GraphChi, X-Stream adopts the edge-scope GAS programming model, but the execution model is different. X-Stream eliminates the need for sorting edges, but instead streams a completely unordered list of edges. Streaming disk-resident (resp., memory-resident) edge lists allows X-Stream to fully utilize the high sequential bandwidth of a hard-disk (resp., main memory) with the help of a main-memory buffer (resp., CPU cache).

The computation model of X-Stream is edge-centric, and each iteration consists of two sequential passes. We now illustrate the two passes by considering PageRank computation (with damping factor of 0.85), assuming that the vertex states are all maintained in an in-memory array, and edges are stored on disk in random order. *The first pass streams* *the edge list and performs edge-centric scattering.* For each edge (u, v), X-Stream gets $a(u)$ and $d_{out}(u)$ from the in-memory state of u, and computes an update value $a(u)/d_{out}(u)$ for the edge (u, v), which is then appended to an update stream on disk. After the first pass, X-Stream re-initializes all in-memory vertex values $a(v)$ as $(1-0.85)/|V|$. *The second pass streams the list of updates and performs edge-centric gathering.* For each update $m = a(u)/d_{out}(u)$ of an edge (u, v), we add $0.85 \cdot m$ to the in-memory value field $a(v)$. After the second pass, the PageRank values of all vertices are advanced for one iteration.

To sum up, there are three important UDFs in X-Stream: (1)*init*(.), which indicates how to re-initialize the value of a vertex before the gathering pass; (2)*apply_one_update*(.), which indicates how to update the value of a vertex using a gathered update; and (3) *generate_update*(.), which indicates how to generate the update value of an edge (u, v) using u's state (and possibly the edge value of (u, v)).

Out-Of-Core Engine of X-Stream. The above example assumes that all vertex states fit in main memory. When the assumption does not hold, the vertices are partitioned into P disjoint intervals like in GraphChi, so that all vertices in an interval I_i fit in main memory and constitute a vertex partition, which we denote by V_i. Since a vertex partition does not keep edges, when we are given a memory space constraint, a partition in X-Stream contains more vertices than a shard in GraphChi. Therefore, the number of partitions in X-Stream is much smaller than the number of shards in GraphChi.

Each vertex partition V_i is also associated with an edge partition E_i, which contains all out-edges of vertices in V_i, i.e. $\{(u, v) \in E \mid u \in V_i\}$. In the first pass for edge-centric scattering, each vertex partition V_i is loaded into memory to generate updates by streaming the edge partition file E_i. This pass writes to P update files, $U_1, \ldots, U_P$, one for each partition. When an update on edge (u, v) (where $v \in V_j$) is generated, the update is appended to the update file U_j.

In the second pass for edge-centric gathering, each vertex partition V_i is loaded into memory to update vertex values by streaming the update file U_i generated by the first pass.

The edge partition can be generated in one pass over the whole edge list, and by appending each edge (u, v) where $u \in V_i$ to the edge file E_i. This is the only preprocessing required by X-Stream.

In-memory Engine of X-Stream. When all edges also fit in main memory, the goal becomes to fully exploit the parallelism of all available cores, which is possible since disk bandwidth is no longer the bottleneck. To hide the memory bandwidth, the data required in processing of each vertex partition V_i should fit in a CPU cache. However, unlike in main memory where we can pin the vertex states and only use a small buffer for streaming edges and updates, users do not have direct control over CPU caches. Therefore, X-Stream partitions vertices into intervals so that for each interval I_i, the total space required by V_i, E_i and U_i does not exceed the size of a CPU cache. This requirement leads to much smaller partition size and thus a much larger number of partitions.

Since the number of partitions is large, there are many update lists U_j to write to. To shuffle the generated updates to the update lists efficiently, X-Stream adopts a tree-structured multi-stage shuffler to improve the cache locality of shuffling. Moreover, to avoid write contention, each thread only appends updates to its own private buffer, and flushes them to update lists in batches. Work stealing is also used to avoid workload imbalance: if a thread finishes processing all its assigned partitions, it will steal partitions from other threads to process.

The Weakness of X-Stream. As admitted in [Roy et al., 2013], X-Stream is inefficient for *graphs whose structure requires a large number of iterations*, such as a large graph diameter. This is because each iteration streams all edges of a graph even if only a small number of vertices participate in computation. This weakness is also observed by other works such as [Yan et al., 2016d].

Chaos: Scaling-out X-Stream. In a single-machine environment, there is only one disk (or RAID), which limits the disk streaming bandwidth. Chaos [Roy et al., 2015] scales out X-Stream by distributing the partitions (i.e., (V_i, E_i) pairs) evenly to multiple machines for processing, so that each machine only streams part of the partitions. In this way, the disks of all machines are streamed in parallel, which leads to a much larger cumulative disk streaming bandwidth.

Chaos uses a storage sub-system to manage the vertex states, edges and updates. A master keeps track of the vertices and edges of every partition, and the generated updates towards every partition; while a computing thread sends requests to the master for the necessary data for processing a partition. This approach assumes that *cluster network bandwidth far outstrips storage bandwidth*, which is *a fundamental assumption underlying the design of Chaos* [Roy et al., 2015]. In fact, [Roy et al., 2015] reported that Chaos only achieves good performance by using large-SSD machines connected by 40 Gigabit Ethernet, and the performance is undesirable when the more common Gigabit Ethernet is used.

When a machine W_a finishes all its assigned partitions in an iteration, it may send request to another machine W_b to steal the workload of processing a partition $P_i = (V_i, E_i)$ assigned to W_b. If W_b accepts the request, W_a may pull the data of P_i for processing. Since both W_a and W_b may be processing P_k, the master of the storage sub-system should assign disjoint sets of edges (during scatter) and updates (during gather) to W_a and W_b. Multiple accumulated values of a vertex (processed by different computing threads) need to be combined before completing the gather phase.

5.2.3 VENUS

We now introduce VENUS [Cheng et al., 2015], which is designed for vertex-centric iterative computation on a static graph and thus does not support graph mutation.

Programming Model. Unlike the edge-scope GAS programming model of GraphChi and X-Stream, VENUS adopts a value-scope GAS programming model where a vertex v only accesses D_v, and D_u for all $u \in \Gamma_{in}(v)$. For example, in PageRank computation, when a vertex v computes D_v by summing $a(u)/d_{out}(u)$ of every $u \in \Gamma_{in}(v)$, it directly accesses the vertex value of u. This is in contrast to the approach of GraphChi and X-Stream, where u first distributes the value (or update) to edge (u, v), and then v gets the value (or update) from (u, v). As a result, this approach saves the disk-IO cost of writing $O(|E|)$ values (or updates) to the edges.

Data Organization. In the above programming model, each iteration only needs to scan the static graph topology data once, but the value of a vertex may be accessed multiple times to update the value of its out-neighbors. Therefore, VENUS separates the read-only structure data from mutable vertex values on disk: the structure data are streamed (or sequentially read) during the computation which only requires a small in-memory buffer, and thus the available main memory can be used to cache as many mutable vertex values as possible (for efficient access during the computation). This is in contrast to GraphChi where all adjacent edges of a shard need to be loaded into main memory before the shard can be processed. This new computation model is called *vertex-centric streamlined processing* (VSP).

We now introduce the data organization of VENUS in more detail. Like in GraphChi, VENUS requires the IDs of the vertices in a graph to be numbered as $1, 2, \cdots$. The vertices are partitioned into P disjoint intervals according to their IDs, $I_1, \ldots, I_P$, and each interval I_i defines a g-shard and a v-shard as follows: (1) the g-shard stores all the edges (and the associated read-only attributes) with destinations in I_i, and (2) the v-shard contains all vertices in the corresponding g-shard, including the source and destination of each edge. Moreover, edges in a g-shard are ordered by destination, and thus the in-edges of each vertex are stored consecutively. In other words, the g-shard of I_i stores $\Gamma_{in}(v)$ (and the associated edge attributes) of every vertex $v \in I_i$ one by one. All g-shards are further concatenated to form the *structure table*, which is streamed during VSP.

Computation Model. In each iteration, the VSP model processes vertices of $I_1, \ldots, I_P$ one by one, where the in-edges of each vertex is streamed from the structure table. If the values of all vertices fit in memory, VENUS supports a semi-streaming in-memory mode similar to that of GraphChi. Otherwise, VENUS supports two IO-friendly algorithms described as follows.

The first algorithm materializes all v-shard values as a view for each shard. To process the vertices in I_i, the relevant vertex values are loaded from the v-shard of I_i. After all vertices in I_i are processed, their new values need to be updated to all the relevant v-shards. This is because, a vertex $u \in I_i$ may be the in-neighbor of another vertex $v \in I_j$ $(j \neq i)$, and thus u's value is included in the v-shard of I_j. VENUS orders the values of the vertices in each v-shard by their vertex ID, and thus for each v-shard I_j, the values of those vertices whose IDs are in interval I_i are stored consecutively and can be updated by one sequential write.

To eliminate the cost of materializing all (possibly replicated) vertex values in v-shards, the second algorithm merge-joins each v-shard with the table of all vertex values. In this case, the v-shard of I_i only stores the IDs of the relevant vertices, and their values are obtained by joining the v-shard with the vertex value table by vertex ID. Since data in both a shard and the vertex value table are ordered by vertex ID, merge-join is applicable and efficient. The join results include all relevant vertex values and are cached in memory for processing I_i.

The vertex decomposition and the use of joins to reconstruct vertices in VENUS is very similar to that in Vertexica [Jindal et al., 2014b], a graph processing system built on top of Vertica[2]. Although Pregelix [Bu et al., 2014] provides out-of-core supports, it does not decompose vertices. The details of both Vertexica and Pregelix can be found in Section 8.2.

5.2.4 GridGraph

GridGraph [Zhu et al., 2015] is a single-machine out-of-core system which represents a graph by a grid to reduce the amount of I/O during the processing. Similar to GraphChi, the vertices are partitioned into P intervals. Using the P intervals, the vertex vector is broken into P chunks, and the adjacency matrix A is broken into a grid of $P \times P$ edge blocks. Here, a block (I_a, I_b) includes all edges with source in interval I_a and destination in interval I_b, and are stored (in any order) as a file on the disk. Note that unlike GraphChi, X-Stream and VENUS, the edges of a vertex may be stored in multiple files in GridGraph.

[2] Vertica: `http://www.vertica.com`

Computation Model. GridGraph combines the scattering and gathering phases into one “streaming-apply” phase, which streams every edge and applies the generated update instantly onto the vertex. Let $a_i(v)$ be the vertex value of v at iteration i. In an iteration i, GridGraph processes each block (I_a, I_b) once if any vertex $v \in I_a$ is active. To process the edges in block (I_a, I_b), GridGraph pins the following data in main memory: (1) the vertex value chunk of I_a for iteration i, (2) the vertex value chunk of I_b for iteration $(i + 1)$. The update on an edge (u, v) is computed from $a_i(u)$ (obtained from the chunk of I_a) and the value of edge (u, v), which is then aggregated to $a_{i+1}(v)$ in the chunk of I_b.

GridGraph processes the blocks in the grid column by column, and for each column, the blocks are processed from top to bottom. The benefit of column-oriented computation is that, all blocks in a column for interval I_b updates the vertex values in chunk I_b, and thus chunk I_b can be pinned in memory during the processing of the whole column, and flushed to the disk after the column is processed. Therefore, the processing of each column needs to read P chunks of I_a (i.e., $O(|V|)$ vertex values) and write one chunk of I_b. Accordingly, each iteration needs to read $O(P \cdot |V|)$ data of vertex chunks and write $O(|V|)$ data of vertex chunks, in addition to streamingly reading $O(|E|)$ edges in the grid. The efficiency of this computation model lies in the fact that the data written to disk is linear to the vertex number, rather than the edge number as in GraphChi and X-Stream.

Since the edges are processed in parallel, different threads may apply in-place updates to the same vertex in chunk I_b, and thus data aggregations are implemented by atomic operations.

Other Optimizations. Besides the BSP model described above, GridGraph can also run in asynchronous mode, in which case for a block (I_a, I_b), there is no concept of iteration number for the corresponding vertex chunks I_a and I_b, and they are just the states of subsets (i.e. intervals) of vertices that can be both read and updated. This allows faster convergence for algorithms like Hash-Min.

Note that the edges in a block are streamed during the “streaming-apply” computation, which only requires a small in-memory buffer to achieve full sequential bandwidth for large block files. However, some blocks can be very sparse, and the small file size may lead to frequent disk seeks. Therefore, the blocks are appended into one large file for streaming, with block boundaries recorded so that GridGraph knows when to pin/unpin a vertex chunk.

Recall that each iteration in GridGraph reads $O(P \cdot |V|)$ data of vertex chunks, and thus a small value of P reduces the I/O accesses of vertices. GridGraph sets P as the smallest integer that still guarantees that a vertex chunk (with $|V|/P$ vertices) fits in main memory. Moreover, since each block (I_a, I_b) is processed in main memory, and to maximize the CPU-to-memory access locality, each block is further partitioned by a $Q \times Q$ grid, and accordingly, each vertex chunk (i.e., I_a and I_b) is partitioned into Q smaller sub-chunks such that each sub-chunk fits into the last-level CPU cache. Like the processing in the upper-level grid, the lower-level grid of each block is also processed in a column-oriented manner to reduce cache misses.

5.3 Summary

We have reviewed two kinds of systems that adopt the shared memory programming abstraction: (1) GraphLab and PowerGraph that keep data in main memory and execute asynchronously, and (2) single-machine out-of-core systems that execute synchronously. We remark that unlike the real shared memory systems that we shall introduce in Chapter 9.2, the systems we reviewed here do not actually perform shared memory computation at the backend. As a result, a vertex can only access data of its adjacent vertices/edges, rather than any vertex it keeps track of like in Pregel. We regard the edge-streaming models of X-Stream and GridGraph as a special vertex-centric model (that is disk-friendly), since the edge values still need to be aggregated at the receiving vertices. Also, we consider single-machine out-of-core systems as performing synchronous execution, since each iteration needs to stream the whole graph on disk(s); but this is different from Pregel's model where a vertex always receives a value from the last iteration. Here, a vertex may be accessing the value of another vertex that is written to disk when processing a previous shard in the same iteration, and thus the total number of iterations can be less than that of a Pregel algorithm. Last but not least, we remark that single-machine out-of-core systems may not a wise choice for algorithms running for many iterations but only a small number of vertices compute in each iteration, since the entire graph needs to be streamed again and again.

Part II

Beyond Vertex-Centric Programming Model

6

Matrix Algebra-Based Systems

Graphs and matrices are inherently related, so a number of big graph systems expose a matrix-based interface for users, instead of using the vertex-centric programming model. In this chapter, we review three important matrix-based systems for big graph processing, PEGASUS [Kang et al., 2009], GBASE [Kang et al., 2011], and SystemML [Ghoting et al., 2011]. All three systems expose a matrix-based interface for users, and rely on a general purpose data processing system, such as MapReduce or Spark, for distributed execution.

6.1 PEGASUS

Before the advent of Pregel, big graphs were processed using MapReduce, and many tailor-made MapReduce algorithms were proposed for solving specific graph problems. [Kang et al., 2009] pioneered the research in developing general-purpose MapReduce-based frameworks for designing parallel graph algorithms. They developed a MapReduce-based system called PEGASUS, which models graph computation by a generalization of matrix-vector multiplication. Compared with the vertex-centric systems that were proposed later, PEGASUS suffers from two problems: (1) it is not user-friendly to programmers that are not familiar with matrix computations, and (2) it relies on the disk-based Hadoop MapReduce for execution, which limits its performance.

We now review the PEGASUS system. PEGASUS models each iteration in iterative graph computation by a generalized matrix-vector multiplication operation, which is repeated until the vertex values in the vector converge. Taking the PageRank algorithm as an example, each superstep can be formulated by the following matrix-vector multiplication operation:

$$\vec{v} \leftarrow (0.85 \cdot A^T + 0.15 \cdot U) \cdot \vec{v}. \tag{6.1}$$

In Equation (6.1), (1) $\vec{v}$ is a column vector with $|V|$ elements, and each element $\vec{v}[i]$ refers to the value of a vertex v_i (i.e., $a(v_i)$); (2) A is the adjacency matrix, and $A[i][j] = 1/d_{out}(v_i)$ if edge (v_i, v_j) exists, and $A[i][j] = 0$ otherwise; (3) U is a $|V| \times |V|$ matrix with all elements set to $1/|V|$. Let us denote $M = 0.85 \cdot A^T + 0.15 \cdot U$, then Equation (6.1) has the form $\vec{v} \leftarrow M \cdot \vec{v}$. This matrix-vector multiplication operation is defined by three operators:

1. combine2: to multiply $M[i][j]$ and $\vec{v}[j]$;
2. combineAll: to sum the $|V|$ multiplication results for v_i (i.e., $\sum_{j=1}^{|V|}\{M[i][j] \cdot \vec{v}[j]\}$);
3. assign: to overwrite the old value of v_i with the new sum.

In the PageRank algorithm, combine2 simply performs multiplication while combineAll simply performs summation. PEGASUS lets a user customize the above three operators to implement different graph algorithms. For example, to implement the Hash-Min algorithm, we simply set M to be the 0-1 adjacency matrix of an undirected graph G, and the three operators are defined as follows:

1. $combine2(i, j) = M[i][j] \cdot \vec{v}[j]$;
2. $combineAll(i) = \min_{j=1}^{|V|}\{combine2(i, j)\}$;
3. assign: $\vec{v}[i] \leftarrow \min\{\vec{v}[i], combineAll(i)\}$.

Note that every iteration in PEGASUS processes all vertices in a graph. For example, in the above Hash-Min algorithm, even in later iterations, every vertex v_i needs to recompute its value since there is no mechanism for individual vertices to halt.

To improve performance, PEGASUS partitions the matrix M into $b \times b$ submatrices, and partitions the vector $\vec{v}$ into groups of b elements, so that the matrix-vector multiplication is performed in coarser granularity: combine2 now becomes multiplying a $b \times b$ matrix by a column vector with b elements, rather than multiplying two scalars. The blocking approach provides several performance advantages: (1) the coarser-grained computation decreases the number of items to sort during the shuffling phase of MapReduce; (2) each submatrix is represented by two 4-byte integers (*block-rowID*, *block-columnID*) followed by non-zero elements represented by two $\log b$-bit integers (*rowID*, *columnID*); this is more compact than associating each non-zero element with two 4-byte integers individually.

PEGASUS further adopts two optimizations to the block multiplication approach. Firstly, PEGASUS preprocesses the matrix M by co-clustering (i.e., reordering rows and columns), so that there are fewer submatrices but each submatrix contains more non-zero elements. As a result, there are fewer (*block-rowID*, *block-columnID*) pairs, reducing the data volume (and number of data items) to be streamed (and shuffled) by MapReduce. Secondly, to reduce the number of iterations (i.e., MapReduce jobs), PEGASUS multiplies each diagonal matrix block with the corresponding vector block repeatedly in each iteration until the contents of the vector block do not change. This optimization propagates vertex states throughout the whole submatrix block (i.e., a subset of vertices), and thus reduces the number of iterations. This method shares the same idea as the block-centric computation discussed in Section 4.1.

6.2 GBASE

GBASE [Kang et al., 2011] is another MapReduce-based big graph system, with the following differences from PEGASUS. Firstly, GBASE aims to efficiently support "targeted" queries which need to access only part of the graph, in addition to "global" queries that traverse the whole graph. Secondly, a user of GBASE uses built-in graph operations to process and mine graphs, rather than write their own algorithms. Thirdly, each built-in graph operation is implemented by the exact matrix-vector multiplication(s) (run on MapReduce), not the generalized matrix-vector multiplication(s).

Querying. GBASE unifies different graph operations by the matrix-vector multiplication operation $\vec{v}_{out} \leftarrow M \cdot \vec{v}_{in}$. Here, $\vec{v}_{in}$ is a column vector with $|V|$ elements (one for each vertex), which serves as the query input. There are two cases for M and $\vec{v}_{out}$, which we describe below.

Case 1: M is the adjacency matrix A or its transpose A^T, and $\vec{v}_{out}$ is a column vector with $|V|$ elements (one for each vertex). For example, the vector of in-neighbors (resp. out-neighbors) of a vertex v_i is computed by $A \cdot \vec{e}_{v_i}$ (resp. $A^T \cdot \vec{e}_{v_i}$), where $\vec{e}_{v_i}$ is the indicator vector of v_i (i.e., $\vec{e}_{v_i}[i] = 1$ and all other elements of $\vec{e}_{v_i}$ are 0).

Case 2: M is the $|E| \times |V|$ incidence matrix B, where $B[i][j] = 1$ if edge e_i is adjacent to v_j, and $B[i][j] = 0$ otherwise; and $\vec{v}_{out}$ is a column vector with $|E|$ elements (one for each edge). For example, to compute the subgraph induced by a subset of vertices S, we can compute $B \cdot \vec{v}_{in}$ where $\vec{v}_{in}[i] = 1$ if $v_i \in S$ and $\vec{v}_{in}[i] = 0$ otherwise. The result $\vec{v}_{out}$ consists of elements with values 0, 1, or 2, and the induced subgraph is given by those edges whose corresponding values in $\vec{v}_{out}$ are 2 (since it means that both endpoints of the edges are in S).

The built-in operations can be further combined to perform more advanced graph operations. For example, to get the set of vertices within k hops from v, one may left-multiply $\vec{e}_v$ by A^T for k times; while to get the k-ego network of v, one may left-multiply B to the result vector of v's k-hop neighbors.

Storage. To support the querying of a graph G, GBASE needs to first preprocess the adjacency matrix A into multiple files, so that during the processing of a query q, only those files relevant to q are read by MapReduce. As a result, a "targeted" query usually reads only a small portion of files.

For this goal, GBASE first reorders the rows and columns of A and partitions A into homogeneous submatrices called blocks. This is achieved by using an existing graph partitioning algorithm like METIS, and each block is either very dense or very sparse. This step is similar to the optimization of matrix co-clustering in PEGASUS. However, instead of encoding a block with the coordinates of its non-zero entries as in PEGASUS, GBASE converts each block into a binary string (e.g., by concatenating the rows of the 0-1 submatrix), and then compresses the string using GZip. [Kang et al., 2011] reported that the total size of the compressed files is less than 2% of the original graph size. As a trade-off, the files have to be unzipped for processing, but the storage savings and the reduced amount of data transfer outweigh the decompression overhead.

The compressed blocks (or, submatrices) are then grouped into files. One may group the blocks by rows, so that for an out-neighbor query where only a row of A is required, only one file is read; however, all files need to be read for an in-neighbor query. Symmetrically, grouping blocks by columns is undesirable for out-neighbor queries. To solve the problem, GBASE divides the matrix of blocks by a coarser-grained grid, and groups all blocks in each grid cell into a file. Let the total number of files be n, then with the grid placement of blocks, both an in-neighbor query and an out-neighbor query read $O(\sqrt{n})$ files.

6.3 SystemML

Different from the above two matrix-based systems, SystemML [Ghoting et al., 2011] takes a declarative approach to graph analytics and, more generally, to machine learning (ML). Instead of exposing programming APIs to the end users, SystemML provides a high-level scripting language for users to express their analytics algorithms, without them worrying about how to execute individual operations. SystemML compiles the algorithm scripts, and then optimizes and executes them in a hybrid runtime of a single node and a distributed cluster by using either MapReduce or Spark. Furthermore, SystemML supports general linear algebra operations, besides just matrix-vector multiplication, and is thus applicable for any ML or analytics algorithms that can be expressed using linear algebra.

Programming Language. Algorithms in SystemML are written in a high-level language called **D**eclarative **M**achine learning **L**anguage (DML). This language includes linear algebra, statistical functions, and control flow constructs like loops and branches. Script 1 shows how the PageRank algorithm can be written in just 11 lines of code in this high-level language.

Script 1. PageRank

```
G = readMM("in/G", rows=1e6, cols=1e6, nnzs=1e9); // G: input graph
p = readMM("in/p", rows=1e6, cols=1); //p: initial uniform pagerank
e = readMM("in/e", rows=1e6, cols=1); //e: all-ones vector
ut = readMM("in/ut", rows=1, cols=1e6); //ut: a vector for per-node
personalization, e.g. $ut_i = \frac{1}{d(v_i)}$, where $d(v)$ is the out-degree of $v$
alpha = 0.85; //teleport probability
max_iteration = 100;
i = 0;
while(i < max_iteration) {
  p = alpha * (G %*% p) + (1 - alpha) * (e %*% ut %*% p);
  i = i + 1; }
writeMM(p, "out/p");
```

Optimization. The high-level scripts are compiled into DAGs of high-level operators (HOPs), then DAGs of low-level operators (LOPs), and eventually executable instructions. SystemML applies optimizations at different levels of abstraction during algorithm compilation, including constant propagation/folding, common subexpression elimination (CSE), operator ordering, operator selection, recompilation decisions, and piggybacking (packing multiple instructions into a single MapReduce job). The cost-based optimizer in SystemML generates execution plans based on data characteristics, like the dimensionality and sparsity of matrices, as well as cluster and machine characteristics, such as memory and CPUs.

Matrix Representation. Similar to PEGASUS and GBASE, a matrix in SystemML is divided into blocks for efficient processing and storage. However, unlike the other two systems, SystemML does not require the expensive clustering of vertices in a preprocessing step to form the matrix blocks. At the per matrix-block level, dynamic runtime optimizations are also applied for choosing block layout (sparse or dense layout) and implementations of block-level operations, based on the statistics (e.g. density of a matrix block) gathered at runtime. In addition, lightweight database compression techniques can be applied to matrix blocks, and then linear algebra operations, such as matrix-vector multiplication, can be executed directly on the compressed representations [Elgohary et al., 2016].

Runtime. SystemML generates hybrid runtime execution plans that range from in-memory, single node execution to large-scale cluster execution of operators on either MapReduce or Spark [Boehm et al., 2016], hence enabling scaling up or down of computation. For cluster execution, SystemML utilizes resource negotiation frameworks like YARN [Vavilapalli et al., 2013], to achieve automatic resource elasticity in a cluster [Huang et al., 2015]. Besides data parallelism that would be normally achieved using MapReduce or Spark, SystemML also supports task parallelism through which independent iterations in loops can be executed in parallel [Boehm et al., 2014]. Furthermore, SystemML also strives to achieve the numerical accuracy [Tian et al., 2012] of operations in addition to efficiency.

6.4 Summary

This chapter describes three matrix-based big graph systems. We now summarize their commonalities and differences, and compare them to the vertex-centric graph systems.

6.4.1 Comparison of the Matrix-Based Systems

In a high level, the three matrix-based graph systems share commonalities, such as exposing a matrix-based interface for graph analytics and relying on a underlying general-purpose distributed processing system for execution. But they are also very different in many aspects. Table 6.1 summaries the major differences of the three systems.

In terms of supported analytics, PEGASUS and GBASE are purely designed for graph analytics, whereas SystemML is for general ML (including graph analytics). As a result, SystemML supports general linear algebra as its primitive operations, beyond just matrix-vector multipli-

	PEGASUS	**GBASE**	**SystemML**
Supported Analytics	global graph analysis	targeted & global graph analysis	graph analysis & ML
Core Operations	matrix-vector multiplication	matrix-vector multiplication	general linear algebra
User Interface	customizable APIs	built-in algorithms	R-like language
Matrix Blocking	square	general rectangular	general rectangular
Block Formation	clustering nodes	clustering nodes	no clustering needed
Matrix Storage	uncompressed	compressed, special placement	compressed, per-block layout
Query Optimization	NA	NA	cost-based & rule-based
Runtime Platform	MapReduce	MapReduce	single node & MapReduce/Spark

Table 6.1: Comparison of PEGASUS, GBASE and SystemML

cation. Although all are matrix based, the three systems provide very different interfaces, hence different levels of flexibilities to the users: GBASE only exposes built-in algorithms for users to choose, PEGASUS allows users to customize the API implementation for matrix-vector multiplications, whereas SystemML provides a rich scripting language to compose a graph/ML algorithm using linear algebra operations.

All three systems break a matrix into blocks for efficient processing and storage. PEGASUS only uses square blocks, whereas the other two support general rectangular blocks. Both PEGASUS and GBASE require a preprocessing step to cluster the nodes in order to generate compact blocks. In terms of storage, both GBASE and SystemML support compression on the blocks. To support efficient access to a matrix, GBASE uses a grid placement to minimize file access, whereas SystemML optimizes the physical layout at the matrix-block level.

Although a number of system optimizations are used in PEGASUS and GBASE, these two systems do not apply any query-specific optimizations. Taking a declarative approach, SystemML employs both cost-based and rule-based optimizations for each graph/ML algorithm. In addition, SystemML also generates a hybrid runtime to scale up and down the computation.

6.4.2 Comparison of Matrix-Based and Vertex-Centric Systems

Finally, let's compare the matrix-based graph systems to the vertex-centric graph systems.

Which interface is better, matrix-based or vertex-centric, is in the eye of the beholder. For data analysts who are more familiar with writing algorithms using linear algebra operations in R or MATLAB, the matrix-based systems may be more intuitive to them. But for programmers who are comfortable writing code, the vertex-centric systems can give them more flexibility and control, as they can express more customized logic for their graph algorithms, which can be hard to express using matrix operations.

As for the runtime, the three matrix-based systems all rely on an existing general-purpose data processing engine for execution, whereas many vertex-centric graph systems implement their own runtime engines specially designed for graph processing. On one hand, this makes graph analytics on the matrix-based systems more easily integrated with other types of data processing tasks, such as Extract-Transform-Load (ETL) and ML, on the same underlying processing platform. On the other hand, this also means that the vertex-centric systems can often avoid some unnecessary overhead incurred in matrix-based systems. For example, MapReduce materializes results after each Map-and-Reduce phase. As a result, the matrix-based systems built on MapReduce have to read and write graph data many times throughout an algorithm. In comparison, many vertex-centric systems can keep graph data in memory across iterations without incurring IO.

In addition, vertex-centric systems often keep track of whether a vertex is active or not throughout an algorithm, so that computation is only occurred on active vertices. Many iterative graph algorithms, such PageRank and Connected Component, only involve a very small number of active vertices for computation, as the algorithm approaches to its convergence. But in matrix-based systems, each iteration requires a full matrix computation, even though only a few of the matrix elements will change their values. As a result, the matrix-based systems may show some disadvantage in performance compared to the vertex-centric systems.

In summary, both matrix-based and vertex-centric graph systems have their pros and cons. There are many factors that affect the choice of a system for a particular user, including the intuitiveness and the expressiveness of the user interface, the ease of integration with other analytics tasks, and of course the runtime performance.

7

Subgraph-Centric Programming Models

While the vertex-centric and matrix-based systems presented in the previous sections are expressive enough to solve many graph problems, they are not good at solving problems like graph matching and motif mining, where the outputs are sets of subgraphs that may overlap with each other. In fact, existing vertex-centric and matrix-based systems are mainly designed for graph problems whose output size is linear or sub-linear in the size of the input graph, e.g., tasks that require computing one value for each vertex. However, the size of the outputs for graph matching and motif mining can be much larger than the graph size.

Further, many graph analysis tasks need to reason about the local neighborhoods of different nodes; doing this using a vertex-centric programming model can result in very high messaging and memory overheads. A simple example of such a task is the computation of *local clustering coefficient* (a measure of local density around a node) for all nodes of the graph; computing this metric for a node requires access to the adjacency lists of the neighbors of that node, and cannot be easily done in the vertex-centric systems.

In order to handle such problems, several recent systems bind the user-defined computation logic to individual subgraphs, rather than individual vertices or edges. We call such a computation model a *subgraph-centric* programming model, to differentiate it from the block-centric model discussed in Section 4.1 (the scope of the computation for block-centric is similarly limited as vertex-centric models and the outputs there have linear size).

In this chapter, we first look at some existing solutions for graph matching, motif mining, and several other analysis tasks in Section 7.1, and explain why these problems cannot be efficiently processed using a vertex-centric system. Then, in Sections 7.2 and 7.3, we review two recent subgraph-centric systems for solving these problems, NScale [Quamar et al., 2016] and Arabesque [Teixeira et al., 2015].

7.1 Complex Analysis Tasks

7.1.1 Graph Matching & Motif Mining

Graph Matching. In graph matching, we are given a query graph (aka a pattern graph) where each vertex (and/or edge) may have a label, and the goal is to find all subgraph instances in a large data graph that are isomorphic to the query graph. This problem cannot be simply solved by graph traversal as in existing vertex-centric systems, since a cycle in a query graph implies a join operation on the data graph. To see this, consider the query graph to be a cycle of 4 vertices with labels a, b, c and d. In this case, if we do graph traversal in the data graph starting from vertices matching a till vertices matching d, we finally need to check each vertex matching d to see (1) whether its has a neighbor with label a, and (2) whether the neighbor is exactly the vertex matching a when we start the traversal. This essentially requires a join operation on those vertices with label a.

Vertex-Centric Filtering. We are only aware of one solution to graph matching that uses a vertex-centric platform: Gao et al. [2014] solve the problem of approximately detecting a given pattern over a large dynamic graph using Giraph. Since Giraph is not good at problems whose outputs are overlapping subgraphs, their technique simply outputs a flag for each vertex v indicating whether there exists a subgraph containing v that matches the pattern graph. Their approach first finds the highest-degree vertex in the query graph, called the sink vertex s, and then breaks the query graph into multiple smaller components that only connect with each other through s. Subgraph instances that match each component are then found by graph traversal, and joined at s at last to find those subgraphs that match the whole query graph.

To find the subgraph instances that match a component, the edges in the component are first given directions so that the whole component forms a DAG. Edge directions are computed by BFS from s. The data graph is traversed according to the edge directions in the DAG, starting from those vertices with zero in-degree in the query graph. If all incoming edges of a vertex is matched, then the vertex is flagged as matching and it sends messages to match its downstream neighbors in the DAG. However, since the neighbors may be traversed from the same upstream vertex (e.g., with label a) in the query graph, it is necessary to check whether the two matching vertex of a are the same vertex in the data graph. For this goal, Gao et al. [2014] propose to expand each message by including the ID of the matched upstream vertex in the data graph for filtering. Thus, the algorithm is exact if each component contains only one cycle, but false positives may be introduced if there is more than one cycle (the tradeoff being that the message sizes are constant).

Twig Joining. Another solution is to decompose the query into acyclic subgraphs called twigs: the matching subgraphs of each twig are first found by (join-free) graph traversal, and then, these subgraphs are joined on connecting vertices to compute the exactly matched subgraphs. For example, Sun et al. [2012] implement such an algorithm on top of their Trinity system, which we briefly review next.

Trinity [Shao et al., 2013] (now called *Graph Engine*) is a distributed graph engine built on top of a memory cloud. It stores billions of runtime vertex objects with high memory utilization ratios, to enable fast vertex access. A unified declarative language called Trinity Specification Language (TSL) is provided for data modeling (e.g., to define data schemata) and message passing. While vertex-centric computation paradigms are supported by Trinity for both online querying and offline analytics, Trinity can also support other flexible computations on top of its memory cloud, such as to build a local inverted index on each machine W that maps each label a to those vertices in W that match a, or to perform distributed join, both of which are required by their graph matching algorithm [Sun et al., 2012].

While the algorithm by Sun et al. [2012] is designed for labeled graphs, there is another MapReduce-based solution for graph matching on unlabeled graphs [Lai et al., 2015], which also joins small twigs to compute the matched subgraphs (in terms of topology only).

Motif Mining. There are also a few MapReduce solutions to finding motifs. For example, Suri and Vassilvitskii [2011] proposed two MapReduce algorithms for triangle counting, while Xiang et al. [2013] proposed a MapReduce algorithm for finding the maximum clique. For these two problems, it is sufficient to check the 1-ego network of every vertex since a triangle and a clique are both complete graphs. Some motifs may require checking a larger neighborhood of every vertex. For example, to find quasi-cliques, the algorithm by Liu and Wong [2008] checks the two-hop neighborhood of each vertex.

Both of the above-mentioned approaches [Suri and Vassilvitskii, 2011] and [Xiang et al., 2013] divide a graph into subgraphs to be processed by different reducers in parallel. Although a Pregel-like algorithm has been proposed for triangle counting [Quick et al., 2012], a superstep may generate $O(|V|^{1.5})$ amount of messages, and thus, that approach is not scalable.

Summary. To summarize, while there exist some vertex-centric solutions to graph matching and motif mining, they either provide approximate outputs [Gao et al., 2014] or are not scalable [Quick et al., 2012]. Exact algorithms for graph matching may require join operations that are not well-supported by vertex-centric systems, while for motif mining, the size of the intermediate results is usually superlinear to the graph size, and those problems are hence often solved by the disk-based MapReduce framework to avoid memory overflow.

7.1.2 Neighborhood-Oriented Analysis Tasks

The vertex-centric model limits the compute program's access to a single vertex's state and so the overall computation needs to be decomposed into smaller local tasks that can be (largely) independently executed; it is not clear how to do this for many graph algorithms of interest, without requiring a large number of iterations. For example, to analyze a 2-hop neighborhood around a vertex to find social circles [Leskovec and Mcauley, 2012], one would first need to gather all the information from the 2-hop neighbors through message-passing, and reconstruct those neighborhoods locally (i.e., in the vertex program local state). Even something as simple as computing the number of triangles for a node requires gathering information from 1-hop neighbors (since we need to reason about the edges between the neighbors). This requires significant network communication and an enormous amount of memory.

Consider some back-of-the-envelope calculations (from Quamar et al. [2016]) for estimating the message passing and memory overhead for constructing neighborhoods of various sizes at each vertex for the Orkut social network graph with approximately 3M nodes, 234M edges and an average degree of 77. The original graph occupies 14GB of memory for a data structure that stores the graph as a bag of vertices in adjacency list format. Constructing the 1-hop neighborhoods for all vertices through message passing requires 231M messages and consumes a total of 233 GB of cluster memory, whereas constructing 2-hop neighborhoods would require approximately 18B messages and 18TB of memory. It is clear that a vertex-centric approach requires inordinate amounts of network traffic, beyond what can be addressed by "combiners" in Pregel or GPS, and impractical amount of cluster memory. Although GraphLab is based on a shared memory model, it too would require two phases of GAS (Gather, Apply, Scatter) to construct a 2-hop neighborhood at each vertex and suffers from duplication of state and high memory overhead. Furthermore, because most existing graph processing frameworks hash-partition vertices by default, this approach will create much duplication of neighborhood data structures.

Some specific applications which require analyzing neighborhoods are discussed below.

Local Clustering Coefficient (LCC). In a social network, the LCC quantifies, for a user, the fraction of his or her friends who are also friends of each other—this is an important starting point for many graph analytics tasks. Computing the LCC for a vertex requires constructing its ego network, which includes the vertex, its 1-hop neighbors, and all the edges between the neighbors. Even for this simple task, the limitations of vertex-centric approaches are apparent, since they require multiple iterations to collect the ego-network before performing the LCC computation.

Identifying Social Circles. Given a user's social network (k-hop neighborhood), the goal is to identify the social circles (subsets of the user's friends), which provide the basis for information dissemination and other tasks. Current social networks either do this manually, which is time consuming, or group friends based on common attributes, which fails to capture the individual aspects of the user's communities. Automatic identification of social circles can be formulated as a clustering problem in the user's k-hop neighborhood, for example, based on identifying sets of densely connected neighbors [Leskovec and Mcauley, 2012]. Once again, vertex-centric approaches are not amenable to algorithms that consider subgraphs as primitives, both from the point of view of performance and ease of programming.

Social Recommendations. Random walks with restarts (such as personalized PageRank [Backstrom and Leskovec, 2011]) lie at the core of several social recommendation algorithms. These algorithms can be implemented using Monte-Carlo methods [Gupta et al., 2013] where the random walk starts at a vertex v, and repeatedly chooses a random outgoing edge and updates a visit counter with the restriction that the walk jumps back only to v with a certain probability. The stationary distribution of such a walk assigns a PageRank score to each vertex in the neighborhood of v; these provide the basis for link prediction and recommendation algorithms. Implementing random walks in a vertex-centric framework would involve one iteration with message passing for each step of the random walk.

7.2 NScale

NScale [Quamar et al., 2016] is a MapReduce-based framework, which pioneered the overlapping-subgraph centric model for solving problems like graph matching and motif mining. They call their programming model as *neighborhood-centric* or *subgraph-centric*. One benefit of this model is that, since computation on a subgraph is performed in the memory of a single machine, there is no network communication. Another benefit is that the memory of each machine only needs to be able to keep a subgraph, and there is no need to keep the whole graph in main memories like in a vertex-centric system.

To be more specific, the programming model of NScale requires users to specify: (a) subgraphs of interest on which to run the computations through a *subgraph extraction query*, and (b) a user program. NScale supports extraction queries that are specified in terms of four parameters: (1) a predicate on vertex attributes that identifies a set of *query vertices*, (2) k – the radius of the subgraphs of interest, (3) edge and vertex predicates to select a subset of vertices and edges from those k-hop neighborhoods, and (4) a list of edge and vertex attributes that are of interest. The user computation to be run against the subgraphs is specified as a Java program against the BluePrints API [1], a collection of interfaces analogous to JDBC but for graph data.

However, in order to call user-defined computation logic on subgraphs, NScale needs to first extract all relevant subgraphs from the original graph, and then pack them into larger compact subgraph partitions to be processed by different reducers, so that if a vertex (and its adjacency list) is shared by many subgraphs in a subgraph partition, the partition only needs to store it once.

Graph extraction is performed by computing the k-hop neighborhood of each relevant vertex (e.g., using k rounds of MapReduce), and then pruning each obtained subgraph by edge and vertex predicates. To

[1] BluePrints API. `https://github.com/tinkerpop/blueprints/wiki`

group subgraphs into compact partitions for use by different reducers, each mapper then computes a Minhash signature (a kind of locality sensitive hashing on sets) from the vertex set of each subgraph, and shuffles the subgraphs according to the signatures.

Since a partition may still be too large to be kept in the memory of a machine, each reducer then packs its received subgraphs into larger units called bins, such that the graph corresponding to each bin can be kept in the memory of a machine. Bins of a reducer are constructed from its assigned subgraphs, using either a greedy one-pass strategy or a graph clustering algorithm. Each vertex in a bin also maintains a bitmap indicating whether it is contained in each individual subgraph in that bin. During subgraph-centric computation, a reducer loads each of its bin to memory at a time, and processes those subgraphs packed in it.

While NScale supports subgraph-centric computation with small memory requirement, the expensive preprocessing cost is a concern. Specifically, for each graph task, it needs to extract all relevant subgraphs using MapReduce. For example, to obtain the 3-hop neighborhood of a vertex using MapReduce, its 1-hop and 2-hop neighborhoods need to be dumped to and loaded from HDFS. Moreover, it is possible that the graph packing algorithm is even more expensive than if we call user-defined computation logic directly on each subgraph right after its extraction, as the extraction is task-specific and thus cannot be easily reused.

The original NScale system was built on top of Apache Hadoop MapReduce; a later work reimplemented some aspects of it on top of Apache Spark, which helps alleviate some of the efficiency concerns [Quamar and Deshpande, 2016].

7.3 Arabesque

Arabesque [Teixeira et al., 2015] proposed a similar programming model called "think like an embedding" to automate the exploration of a large number of subgraphs, where an embedding is simply a subgraph instance of the original data graph. Besides graph matching and motif mining, Arabesque also supports the mining of frequent subgraph patterns from a large graph, and they term all these problems as graph mining problems.

Unlike NScale where vertices and their adjacency lists are distributed to subgraphs using MapReduce, Arabesque assumes that the entire data graph resides in the memory of every machine, so that graph topology and attribute values can be directly accessed. This allows each machine to grow an embedding without generating any communication.

The computation model of Arabesque is iterative: in the i-th iteration, it grows the set of subgraphs (i.e., embeddings) with i edges/vertices by one adjacent edge/vertex, to construct subgraphs with $(i + 1)$ edges/vertices for processing. Those subgraphs with $(i + 1)$ edges/vertices that pass the processing are collected into a set for processing by the $(i + 1)$-th iteration. Automorphism checking is performed to avoid generating and processing redundant subgraphs with $(i+1)$ edges/vertices. The processing of each iteration is distributed (where each machine only processes a portion of the embeddings), and the computation stops if there are no more subgraphs to process at the beginning of an iteration.

Arabesque adopts a *filter-process* programming model where a user defines two UDFs: (1) *filter*(e), which indicates whether a newly-grown embedding e should be filtered out; if e is not filtered, then (2) *process*(e) is called to compute output subgraphs from e, and to pass e for processing by the next iteration (e.g., to be grown further to generate candidate embeddings). For example, to find cliques, *filter*(e) checks whether e is a clique, and if so, e gets output by *process*(e), and it is passed to the next iteration to grow larger clique candidates. For frequent subgraph pattern mining, additional UDFs need to be defined to specify the aggregation logic.

Multiple embeddings may be grown into one identical embedding, and Arabesque avoids this redundancy by a coordination-free exploration strategy that only keeps canonical embeddings. An embedding e is canonical iff its vertices were visited in the following order: start by visiting the vertex with the smallest ID, and then recursively add the neighbor in e with smallest ID that has not been visited yet. Arabesque characterizes an embedding as the sequence of its vertices ordered by the order in which they were visited.

Since the embeddings may overlap with each other and have a much larger data volume than the graph itself, Arabesque uses a new data structure called Overapproximating Directed Acyclic Graph (ODAG) to store embeddings compactly. Recall that each embedding is canonical and can be represented as a node sequence according to the coordination-free exploration order. ODAG collapses all nodes at the same sequence level that correspond to the same vertex in the graph into one single node. Since restoring the embeddings from the edges between nodes may generate false-positive embeddings (called spurious embeddings), Arabesque filters these spurious embeddings by the canonicality check and the UDF *filter*(e).

7.4 Summary

The two systems discussed in this chapter exemplify efforts to develop more expressive graph programming models that can efficiently handle a larger variety of graph analysis tasks. Of course, another option would be to use single-machine, shared-memory systems like Ligra [Shun and Blelloch, 2013] that expose lower-level programming models; however, those models are typically not as simple and intuitive to program against, and cannot scale out to run with a cluster of machines.

8

DBMS-Inspired Systems

Google's Pregel platform allows problem-solvers to "think like a vertex" by writing a few user-defined functions (UDFs) that operate on vertices, which the framework can then apply to an arbitrarily large graph in a parallel fashion. Open source versions of Pregel such as Giraph[1] [Ching et al., 2015] and Hama[2] have since been developed in the systems community. Each such platform is a distinct new system that had to be built from the ground up. Usually, these systems follow a process-centric design, in which a set of worker processes are assigned partitions (containing sub-graphs) of the graph data and scheduled across a machine cluster. When a worker process launches, it reads its assigned partition into memory, and executes a Pregel (message passing) algorithm.

Meanwhile, the DBMS (database management system) community has spent nearly three decades building efficient shared-nothing parallel dataflow engines [DeWitt and Gray, 1992] that support bulk data processing operators (such as join and group-by [Graefe, 1993]), and query optimizers [Chaudhuri, 1998] that choose an "optimal" execu-

[1] Giraph: `http://giraph.apache.org`

[2] Hama: `http://hama.apache.org`

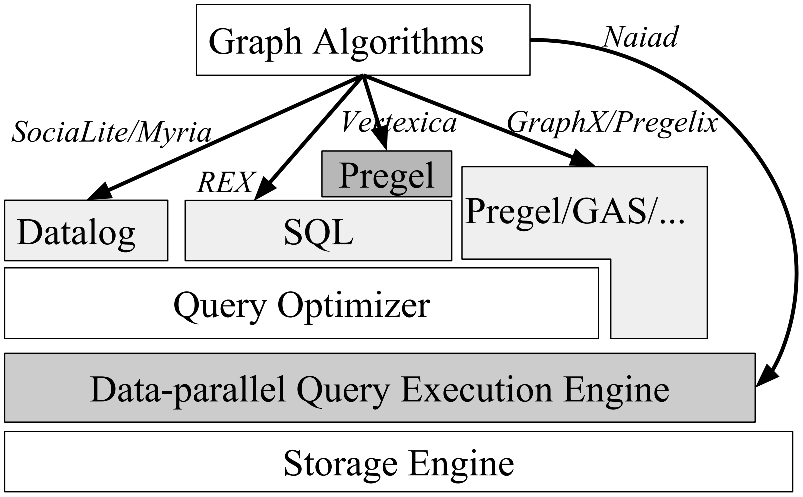


Figure 8.1: The architectural diagram of DBMS-style Big Graph analytics systems.

tion plan among different alternatives. In addition, deductive database systems—based on Datalog—were proposed to efficiently process recursive queries [Bancilhon and Ramakrishnan, 1986], which can be used to solve graph problems such as transitive closure. Extended versions (such as [Beeri et al., 1987], [Mumick et al., 1990], [Seo et al., 2013a] and [Mazuran et al., 2013]) of Datalog with aggregations provide a more declarative user-level programming abstraction than graph-specific "think like a vertex" programming models for expressing graph algorithms. Though recursive queries might not be as intuitive as "think like a vertex" programming models for users who are not familiar with logic programming languages, techniques for rewriting and evaluating recursive queries—most notably magic-sets rewriting [Bancilhon et al., 1986] and semi-naïve evaluation [Balbin and Ramamohanarao, 1987] as well as other general-purpose query evaluation techniques [Graefe, 1993] — still apply and can be used to implement a scalable, fault-tolerant graph processing runtime.

In this section, we provide a review of recently developed Big Graph analytics platforms that are inspired by database query languages, query processing primitives, and applications, including:

- systems that provide declarative programming abstractions like Datalog for Big Graph analytics (Section 8.1),
- systems that provide vertex-centric programming models but internally leverage DBMS-style query evaluation and query optimization techniques for graph processing (Section 8.2),
- systems that support incremental graph processing (Section 8.3),
- systems that integrate a vertex-centric programming model with a declarative query language to support complex, end-to-end analytical pipelines where graph analysis is a sub-task. (Section 8.4).

Figure 8.1 provides a high-level architectural view of several represenitive DBMS-inspired graph analytics systems that we discuss in this chapter. In spite of differences in programming models, i.e., Datalog, SQL, or vertex-centric programming models, those systems share a similar dataflow-based execution model under the hood.

8.1 The Recursive Query Abstraction

There had been a flurry of studies on semantics and evaluation techniques for recursive queries in 1980s and 1990s. Due to the space limitations, we only cover several important results that are relevant to Big Graph analytical systems here.

We use a dialect of Datalog with recursive aggregate functions [Seo et al., 2013a][3], to describe the basic concepts. A Datalog program P is a finite set of rules, or Horn Clauses, where rule r in P has the following form:

$$A : -A_1, ..., A_n$$

This rule can be read declaratively as "A_1 and A_2 and .... and A_n implies A". Each of A (the head) and the A_i's (the subgoals of the body) is an atomic formula, consisting of a predicate applied to terms, which are either constants, variables, or function symbols applied to terms. An atom has the form $p(t_1, . . . , t_j\ )$ where p is a predicate and t_1, . . . , t_j are terms which can be constants, variables or functions. Each

[3] Yedalog [Chin et al., 2015] has a similar syntax and semantics for aggregates.

rule can have multiple bodies. Aggregate functions are expressed as an argument in a head predicate and such an aggregate function is applied to all the terms matching the subgoals. In most literature, initial facts are called EDB (existential database) while derived facts are called IDB (intensional database). The meaning of a Datalog program is defined by the fixpoint semantics, i.e., an interpreter starts with the facts in EDB and derives new facts for IDB by applying the rules iteratively until no new facts can be derived.

Given a graph represented by EDB relation Edge(*src*, *dest*, *distance*) where *distance* is a positive number and each edge is a fact (i.e., tuple) in the relation, let us describe following example graph algorithms in Datalog programs.

Program 8.1. Transitive closure for directed graph [Even, 2011]. The following set of rules generates facts of the form *TC(u, v)* if there is a directed path from u to v.
TC(u, u) :- Edge(u, _)
TC(v, v) :- Edge(_, v)
TC(u, v) :- TC(u, w), Edge(w, v), $u \neq v$

Program 8.2. Single source shortest path for a directed graph [Even, 2011] (from vertex u_0).
Path(u_0, 0.0*)*
Path(v, min(d)) :- Path(u, d_1*), Edge(u, v,* d_2*),* $d = d_1 + d_2$

Program 8.3. All pair shortest path for directed graph [Even, 2011] .
Path(u, v, min(d)) :- Edge(u, v, d);
:- Path(u, w, d_1*), Edge(w, v,* d_2*),* $d{=}d_1{+}d_2$

Program 8.4. Connected Components for undirected graph [Even, 2011]. Here, we use the minimum vertex id as the identifier of a connected component and the following rules recursively propagate connected component ids. The last rule is to count the number of vertexes in each connected component.
Vertex(s) :- Edge(s, _)

Vertex(t) :- Edge(_, t)
Comp(t, min(c)) :- Vertex(t), c=t;
                    :- Edge(s, t), Comp(s, c)
Comp_Count(c, count(1)) :- Comp(_, c)

Program 8.5. PageRank for directed graph (with the convergence threshold being θ). The first two rules obtain the set of vertexes from the *Edge* relation; the third rule "computes" the total number of vertexes in the graph; the fourth rule retrieves the out-bound edge count of each vertex; the fifth fule initializes rank values; the six rule describes how rank values are propagated from one iteration to the other; the seventh rule removes vertexes whose rank value change is less than the convergence threshold θ; the last two rules retrieves the final rank values of all vertexes.
Vertex(s) :- Edge(s, _)
Vertex(t) :- Edge(_, t)
N=count(1) :- Vertex(_)
EdgeCount(v, count(1)) :- Edge(v, _)
Rank(v, 0, 1.0/N) :- Vertex(v)
*Rank2(v, i, 0.85*sum(*r_o*)+0.15/N) :- Rank(s, i-1, r), Edge(s, v),*
                                        EdgeCount(s, c), r_o*=r/c*
Rank(v, i, r_2*) :- Rank2(v, i,* r_2*), Rank(v, i-1,* r_1*), |*r_2*-*r_1*|>*θ
Round(v, max(i)) :- Rank(v, i, _)
Rank_Final(v, r) :- Rank(v, i, r), Round(v, rd), i=rd

A straightforward evaluation according to the fixpoint semantics is to apply rules one round after another until no more facts can be derived, which obviously is too slow for large volumes of data. Thus, a typical Datalog implementation would compile user-defined rules into a query plan, including bulk operators such as joins and group-by aggregations to deal with the large number of facts (i.e., tuples). For example, the third rule in Program 8.1 could be compiled to a join between IDB relation *TC* and EDB relation *Edge*, which is executed one round after the other until no more new tuples could be derived for relation *TC*. To scale for Big Graphs, database-style bulk operators like joins and group-bys can be parallelized to multiple machines by partitioning the join keys or group-by keys [DeWitt and Gray, 1992]. For instance, systems like Myria [Wang et al., 2015] scale Datalog evaluations to a cluster of machines in this way.

To further improve the efficiency, a number of evaluation techniques have been proposed. Due to the space limitations, we cannot cover all the techniques but will enumerate several important ones with examples.

- **Semi-naïve evaluation [Balbin and Ramamohanarao, 1987]:** In Program 8.1, re-applying the rules on discovered *TC* facts multiple times could not derive new facts. Therefore, in each round, we can use only the new *TC* facts from the previous round, to join with existing *Edge* facts to derive new facts. Since in general, only a small fraction of the *TC* facts will be new in any one round, we can significantly reduce the amount of work required. The key concept of semi-naïve evaluation could be summarized by the following formula, by which re-computing R ⋈ S could be avoided:

  $$(R \cup \Delta R) \bowtie (S \cup \Delta S) = (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$$

- **Magic-sets rewriting [Bancilhon et al., 1986]:** In Program 8.3, the same Datalog program can be used to answer a single source shortest path query $Path(u_0, \_, \_)$. In order to efficiently evaluate this query, a query compiler can push the filter down to avoid unnecessary computations. The technique is called magic-sets rewriting. After the rewriting, the query becomes Program 8.6. The resulting query prunes computations over vertexes that are not reachable from u_0 and hence can have the same performance as Program 8.2.
- **Stratified evaluation [Mumick et al., 1990]:** If a rule contains aggregation in the head, all existing facts must be fully examined against the subgoal predicates before deriving a new fact using this rule. In other words, there has to be an execution barrier imposed by every rule containing aggregation in the head and the computation is done one iteration (stratum) after the other until reaching a fixpoint. Bulk-synchronous processing in Big Graph analytics systems has a similar execution model to stratified evaluation.

- **Monotonic queries [Mumick et al., 1990, Ross and Sagiv, 1992, Eisner and Filardo, 2010]:** Though in general, an aggregation in a rule head requires fully examining existing facts, it is not necessary for certain cases. For instance, in Programs 8.2 to 8.4, intuitively, the aggregates can only be monotonically decreased, therefore it is possible to generate and use intermediate partially aggregated tuples when existing facts are not fully examined without changing the final answer to the query. The theoretical results presented in literature [Mumick et al., 1990, Ross and Sagiv, 1992], and [Eisner and Filardo, 2010] are vital for us to understand why asynchronous execution could be used for cases such as Program 8.1 to 8.4 but not for Program 8.5.

Program 8.6. Program 8.3 for query *Path*(u_0, _, _) after applying magic-sets rewriting.
Path(u, v, min(d)) :- Magic_Path(u), Edge(u, v, d);
        :- Magic_Path(u), Path(u, w, d_1*), Edge(w, v,* d_2*),* $d{=}d_1{+}d_2$
Magic_Path(v) :- Magic_Path(u), Path(u, w, _), Edge(w, v, _)
Magic_Path(u_0*)*

Note that expressing Programs 8.1 and 8.3 in vertex-centric programming models is possible but the state size in a vertex has to be proportional to the total number of vertices in the graph, which makes the user defined vertex-centric programs difficult to scale to very large graphs. In addition, Datalog programs such as Program 8.1 to 8.5 are more concise and provide better logical-physical separation than their vertex-centric counterparts.

A few recent research projects are “resurging” Datalog in the context of scalable Big Graph analytics:

- Distributed SociaLite [Seo et al., 2013a,b]. The Datalog dialect we used to express Program 8.1 to 8.5 was proposed by the SociaLite project. The work has shown that a small class of aggregates, *meet* operations, can be evaluated asynchronously. These aggregate functions are *associative*, *commutative*, and *idempotent* binary operations defined on a domain. SociaLite has a working compiler and runtime that automatically translates SociaLite Datalog programs to Java code that runs on a cluster of machines.

- DeALS [Mazuran et al., 2013, Shkapsky et al., 2015, Yang et al., 2015]. The DeALS (*De*ductive *A*pplication *L*anguage *S*ystem) project proposed an alternative approach for monotonic queries. It added monotonic aggregates, i.e., stateful running aggregates that continuously produce partial results, as built-in aggregates for the Datalog language. Therefore, it's programmers' decision to choose between standard aggregates and monotonic aggregates for a particular problem. Mazuran et al. [2013] described the formal semantics, expressive power, and evaluation techniques for this kind of Datalog extensions. However, as far as we know, DeALS is still a single machine system, though it supports multicore processing.
- Myria [Wang et al., 2015]. The Myria project has built a shared-nothing distributed data management system with a data-parallel Datalog query processor that supports synchronous evaluation for general cases as well as asynchronous evaluation for monotonic queries. Interestingly, the experimental results in paper [Wang et al., 2015] show that asynchronous query evaluation does not always lead to the fastest query run times since the benefit of asynchronous execution could be offset by a larger number of intermediate result tuples as well as unnecessary work.
- Yedalog [Chin et al., 2015]. Yedalog is a Datalog implementation developed by Google for exploring knowledge graphs. A Yedalog program can be compiled to three different runtimes: a single machine runtime, an interactive runtime similar to Dremel [Melnik et al., 2010], and a batch runtime (Flume [Chambers et al., 2010]). In addition, Yedalog adopts a flexible data model to support semi-structured data processing. However, the distributed backends of Yedalog are not optimized for expensive recursive queries over very large data sets.

8.2 Dataflow-Based Graph Analytical Systems

Although there have been four decades of researches on recursive queries and their query evaluation techniques, Datalog, and even the recursion support in SQL[4], have not been widely adopted as a programming model by the majority of industry. In our opinion, understandability and debuggability are two major downsides that prevent the prevalence of recursive queries — it is relatively difficult for a programmer to understand a legacy recursive query or to find what is wrong in the query. Recent years, because of the simplicity, "think-like-a-vertex" programming models [Malewicz et al., 2010, Low et al., 2012, Gonzalez et al., 2012] have become de facto application programming interfaces for Big Graph analytics. But as discussed in several recent papers [Bu et al., 2014], [Gonzalez et al., 2014], and [Jindal et al., 2014b], specialized graph processing systems do not yet possess the physical flexibility, scaling properties, and software simplicity that a general-purpose dataflow system can offer.

To bridge the gap, two system projects — Pregelix [Bu et al., 2014] and GraphX [Gonzalez et al., 2014] have tried to build graph-specific programming models on top of general-purpose data-parallel dataflow engines. Both systems replace recursive queries with "modern", user-friendly graph programming models as user interfaces, but internally recast graph-specific optimizations as general-purpose data storage optimizations and query evaluation optimizations in a distributed environment, which have been studied for a long time in the database community. GraphX [Gonzalez et al., 2014] extends Spark [Zaharia et al., 2012] by introducing a small set of specialized graph operators that are sufficient to express existing graph APIs such Pregel [Malewicz et al., 2010] and GAS [Gonzalez et al., 2012]. Pregelix [Bu et al., 2014, Bu, 2015] offers the standard Pregel programming model with global aggregations and graph mutations, as well as an integration with the AsterixDB [Alsubaiee et al., 2014] query language for the ETL support. Regardless of the variance in APIs, both systems implement graph

[4] SQL: `https://en.wikipedia.org/wiki/SQL:1999`

Relation	Schema
Vertex	(vid, halt, value, edges)
Msg	(vid, payload)
GS	(halt, aggregate, superstep)

Table 8.1: Nested relational schema that models the Pregel state.

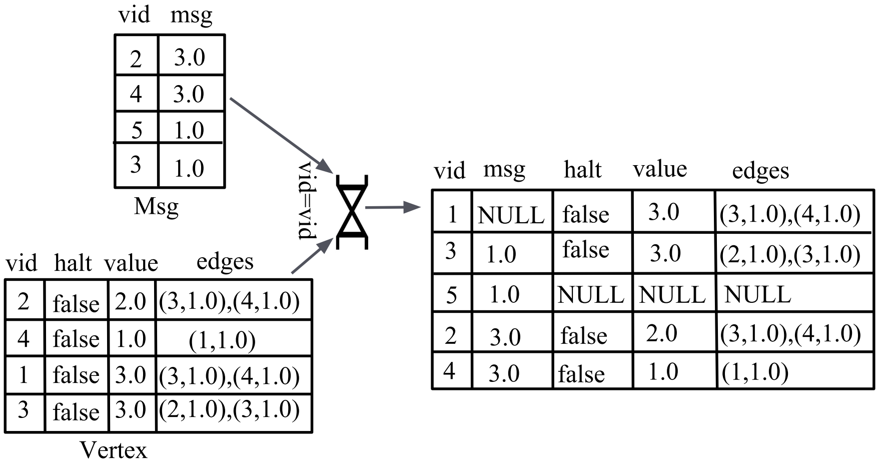
computations as distributed joins (for passing messages) and group-by aggregations (for combining messages).

Let us use Pregelix to walk through the details on how the Pregel think-like-a-vertex programming model can be conceptualized as a set of database algebraic operations. Table 8.1 defines a set of nested relations that Pregelix uses to model the state of a Pregel execution. The input data is modeled as an instance of the `Vertex` relation; each row identifies a single vertex with its halt, value, and edge states. All vertices with a $halt = false$ state are active in the current superstep. The value and edges attributes represent the vertex state and neighbor list, which can each be of a user-defined type. The messages exchanged between vertices in a superstep are modeled by an instance of the `Msg` relation, which associates a destination vertex identifier with a message payload. Finally, the `GS` relation from Table 8.1 models the global state of the Pregel program; here, when $halt = true$ the program terminates[5], *aggregate* is a global state value, and *superstep* tracks the current iteration count.

Figure 8.2 models message passing as a join between the `Msg` and `Vertex` relations. A full-outer-join is used to match messages with vertices corresponding to the Pregel semantics as follows:

- The inner case matches incoming messages with existing destination vertices;
- The left-outer case captures messages sent to vertices that may not exist; in this case, a vertex with the given *vid* is created with other fields set to NULL.

[5] This global halting state depends on the halting states of all vertices as well as the existence of messages.



vid	msg
2	3.0
4	3.0
5	1.0
3	1.0

vid	halt	value	edges
2	false	2.0	(3,1.0),(4,1.0)
4	false	1.0	(1,1.0)
1	false	3.0	(3,1.0),(4,1.0)
3	false	3.0	(2,1.0),(3,1.0)

vid	msg	halt	value	edges
1	NULL	false	3.0	(3,1.0),(4,1.0)
3	1.0	false	3.0	(2,1.0),(3,1.0)
5	1.0	NULL	NULL	NULL
2	3.0	false	2.0	(3,1.0),(4,1.0)
4	3.0	false	1.0	(1,1.0)

Figure 8.2: Implementing message-passing as a logical full-outer join.

UDF	Description
compute	Executed at each active vertex in every superstep.
combine	Aggregation function for messages.
aggregate	Aggregation function for the global state.
resolve	Used to resolve conflicts in graph mutations.

Table 8.2: UDFs used to capture a Pregel program.

- The right-outer case captures vertices that have no messages; in this case, `compute` still needs to be called for such a vertex if it is active.

The output of the full-outer-join will be sent to further operator processing steps that implement the Pregel semantics; some of these downstream operators will involve UDFs that capture the details (e.g., `compute` implementation) of the given Pregel program.

Table 8.2 lists the UDFs that implement a given Pregel program. In a given superstep, each active vertex is processed through a call to the `compute` UDF, which is passed the messages sent to the vertex in the previous superstep. The output of a call to `compute` is a tuple that contains the following fields:

- The possibly updated `Vertex` tuple.

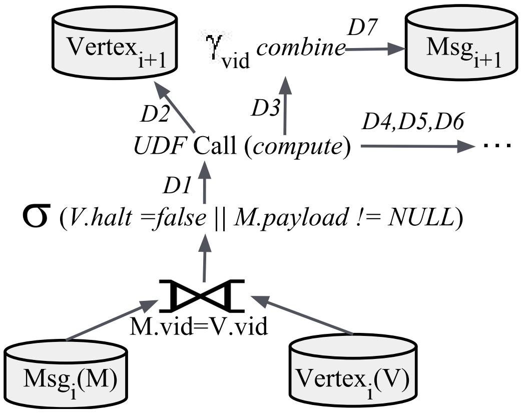
- A list of outbound messages (delivered in the next superstep).
- The global *halt* state contribution, which is *true* when the outbound message list is empty and the *halt* field of the updated vertex is *true*, and *false* otherwise.
- The global *aggregate* state contribution (tuples nested in bag).
- The graph mutations (a nested bag of tuples to insert/delete to/from the `Vertex` relation).

As we will see below, this output is routed to downstream operators that extract (project) one or more of these fields and execute the dataflow of a superstep. For instance, output messages are grouped by the destination vertex id and aggregated by the `combine` UDF. The global aggregate state contributions of all vertices are passed to the `aggregate` function, which produces the global aggregate state value for the subsequent superstep. Finally, the `resolve` UDF accepts all graph mutations—expressed as insertion/deletion tuples against the `Vertex` relation—as input, and it resolves any conflicts before they are applied to the `Vertex` relation.

We now turn to the description of a single logical dataflow plan; we divide it into three figures that each focus on a specific application of the (shared) output of the `compute` function. The relevant dataflows are labeled in each figure. Figure 8.3 defines the input to the `compute` UDF, the output messages, and updated vertices. Flow $D1$ describes the `compute` input for superstep i as being the output of a full-outer-join between `Msg` and `Vertex` (as described by Figure 8.2) followed by a selection predicate that prunes inactive vertices. The `compute` output pertaining to vertex data is projected onto dataflow $D2$, which then updates the `Vertex` relation. In datafow $D3$, the message output is grouped by destination vertex id and aggregated by the `combine` function[6], which produces flow $D7$ that is inserted into the `Msg` relation.

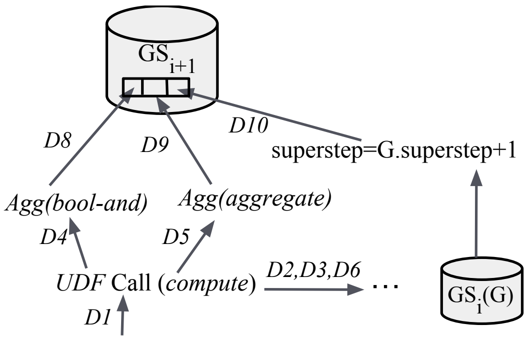
The global state relation `GS` contains a single tuple whose fields comprise the global state. Figure 8.4 describes the flows that revise these fields in each superstep. The halt state and global aggregate fields depend on the output of `compute`, while the superstep counter is simply its previous value plus one. Flow $D4$ applies a boolean aggregate func-

[6] The default `combine` gathers all messages for a given destination into a list.



Flow	Data
D2	Vertex tuples
D3	Msg tuples
D7	Msg tuples after combination

Figure 8.3: The basic logical query plan of a Pregel superstep i which reads the data generated from the last superstep (e.g., $\texttt{Vertex}_i$, $\texttt{Msg}_i$, and $\texttt{GS}_i$) and produces the data (e.g., $\texttt{Vertex}_{i+1}$, $\texttt{Msg}_{i+1}$, and $\texttt{GS}_{i+1}$) for superstep $i+1$. Global aggregation and synchronization are in Figure 8.4, and vertex addition and removal are in Figure 8.5.

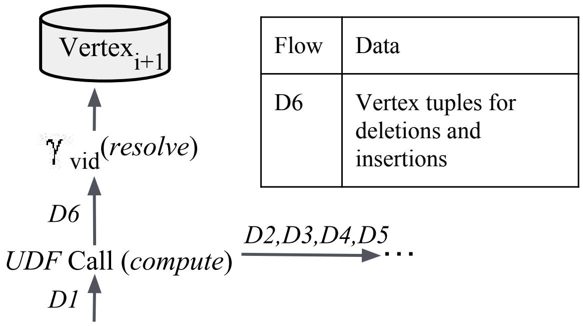


Flow	Data
D4	The global halting state contribution
D5	Values for *aggregate*
D8	The global halt state
D9	The global aggregate value
D10	The increased superstep

Figure 8.4: The plan segment that revises the global state.

tion (logical AND) to the global halting state contribution from each vertex; the output (flow $D8$) indicates the global halt state, which controls the execution of another superstep. Flow $D5$ routes the global aggregate state contributions from all active vertices to the `aggregate` UDF which then produces the global aggregate value (flow $D9$) for the next superstep.

Graph mutations are specified by a `Vertex` tuple with an operation that indicates insertion (adding a new vertex) or deletion (removing a



Flow	Data
D6	Vertex tuples for deletions and insertions

Figure 8.5: The plan segment for vertex addition/removal.

Distributed graph processing system	**Parallel DBMS**
network communication	exchange operator
message passing	join operator
message combination	group-by operator
vertex format	record management
vertex storage	storage management
memory-first computation	buffer cache management
vertex update	storage management
global aggregate	aggregate operator
conflict resolution	group-by operator

Table 8.3: Parallel DBMS primitives for a specialized distributed graph processing system.

vertex)[7]. Flow $D6$ in Figure 8.5 groups these mutation tuples by vertex id and applies the `resolve` function to each group. The output is then applied to the `Vertex` relation.

Table 8.3 summarizes how key components in a distributed graph processing system can be mapped to primitives in a parallel DBMS. By conceptualizing the standard Pregel semantics as a logical query plan, Pregelix turns the problem of how to implement an efficient and flexible distributed graph processing system into how to get an efficient physical query plan that can be evaluated in on a shared-nothing

[7] Pregelix leaves the application-specific vertex deletion semantics in terms of integrity constraints to application programmers.

cluster for the Pregel logical plan. Accordingly, Pregelix uses B-Trees to store vertexes and uses Hyracks [Borkar et al., 2011], an extensible, general purpose dataflow runtime (which is also the runtime for AsterixDB [Alsubaiee et al., 2014]) to scale the evaluation of physical query plans to a cluster of machines. As described in [Bu et al., 2014], such an algebraic conceptualization also allows Pregelix to provide various physical plans that implement the same Pregel semantics to serve different kinds of workloads or clusters.

From a user's perspective, a Pregelix program is almost identical to a Pregel program (e.g., a Giraph program), except that a user can set a few "hint" properties, e.g., message passing strategies (i.e. join algorithms), message combination strategies (i.e., group-by algorithms), and message exchange mechanisms (i.e., data redistribution patterns), to choose a specific physical query plan.

Along the same direction but different from Pregelix and GraphX, the Vertexica project [Jindal et al., 2014b] builds the vertex-centric programming model one level above, i.e., on top of SQL. It has demonstrated that a set of vertex-centric graph algorithms like single source shortest paths, connected components, and PageRank could be translated to SQL queries with user-defined compute functions. It also shows that running the resulting SQL queries on top of the Vertica[8] parallel database offers comparable performance to Giraph [Ching et al., 2015] and GraphLab [Low et al., 2012]. However, Vertexica has not implemented the full-fledged Pregel programming model [Malewicz et al., 2010] regarding to global states, global aggregations, missing destination vertex handling, and no-message vertex handling, all of which together can potentially make the resulting SQL queries harder to read and optimize.

The benefits of building specialized graph programming models on top of general-purpose dataflow systems include:

- **Performance and scaling properties:** A mature general-purpose dataflow engine usually has undergone a lot of performance improvements for a variety of workloads, hence it could be more robust and

[8] Vertica: `http://www.vertica.com`

reliable. For example, Pregelix demonstrated its superiority on out-of-core workloads over Giraph, while GraphX achieved low-cost fault-tolerance by leveraging logical partitioning and lineage from Spark.

- **Physical flexibility:** Several works [Bu et al., 2014], [Jindal et al., 2014b], and [Wang et al., 2015] have found that it is important for a graph processing system to be able to choose among alternative runtime evaluation strategies that would offer a better fit to a particular dataset, algorithm, cluster or desktop.
- **Software simplicity:** The implementation of a specialized graph system spans a full stack of network management, communication protocol, vertex storage, message delivery and combination, memory management, and fault-tolerance; the result is a complex (and hard-to-get-right) runtime system that implements an elegantly simple Pregel-like semantics. For instance, excluding test code and comments, the lines of code of GraphX, Pregelix, and Giraph are 2.5K, 8K, and 32K respectively.
- **Runtime integration:** Dataflow-based graph systems like GraphX and Pregelix can potentially eliminate the need to learn and support multiple systems or write data interchange formats and plumbing to move between systems. Use of SQL makes Vertexica even more attractive in regards to this aspect.

It should be noted that both GraphX and Pregelix are bulk-synchronous graph processing systems — although it is possible, neither of them have provided asynchronous execution alternatives for "monotonic" graph algorithms as discussed in Section 8.1.

The dataflow architecture shown in Figure 8.1 could be (or are being [Jindal et al., 2014a]) adopted by parallel data warehouse vendors (such as Teradata[9], Pivotal[10], or Vertica[11] to build Big Graph processing support by properly reusing their existing software stacks.

[9] Teradata: `http://www.teradata.com`

[10] Pivotal: `http://pivotal.io`

[11] Vertica: `http://www.vertica.com`

8.3 Incremental Graph Processing

A substantial subclass of graph algorithms can be expressed by specifying how changes, i.e., deltas, are propagated from a vertex to its neighborhood. PageRank, single-source shortest path, connected components are such examples. Two recent research projects, REX [Mihaylov et al., 2012] and Maiter [Zhang et al., 2014] have proposed delta-based programming models for expressing this kind of incremental graph computations.

REX developed an SQL extension that supports delta accumulations and allows user-defined delta-handlers for recursions and can compile a program which is a mix of SQL statements and delta-handlers to a runtime query plan that runs synchronously on a data-parallel query execution engine. The key difference between RQL (REX SQL extension) and SQL-99[12] is that SQL-99 recursion accumulates the set of answers, whereas RQL recursion refines the answers with deltas, which fits many graph and machine learning algorithms. Given the same *Edge* relation as described in Section 8.1, the following code is the PageRank program on REX, where *PRAgg.update* reads the current and previous page rank value of a vertex to produce delta values for its neighbors and the SQL WITH clause accumulates delta values for *PR.pr* into the base value for each *srcId* until a fixpoint is reached.

Maiter [Zhang et al., 2014] is a specialized graph system that provides a similar delta-based, accumulative, message passing programming model and implicitly assumes there is a massing passing graph underneath. However, different from REX, Maiter offers data-parallel asynchronous execution for user-defined delta-handlers. To guarantee an accumulative update will yield the same result as its corresponding synchronous iterative update, a user-defined delta handler must meet several conditions:

- Condition A. A vertex can only send messages to a fixed set of "neighbor" vertexes, and every outgoing message from the vertex must be sent to every vertex in the "neighbor" set, which implicitly preclude graph mutations.

[12] SQL: `https://en.wikipedia.org/wiki/SQL:1999`

```
class PRAgg {
  String[] inTypes = {"Integer", "Double"};
  String[] outTypes = {"nbr:Integer", "prdiff:Double"};

  Object[][] update(Object[][] prBucket,
                Object[][] nbrBucket, int nbrId, double pr) {
         double deltaPr = prBucket.get(nbrId)-pr;
     prBucket.put(nbrId, pr);
     if (Math.abs(deltaPr) > 0.01) {
                Object[][] resBag={};
                for (Integer nbr: nbrBucket)
                   resBag.add(nbr, deltaPr/nbrBucket.size());
                return resBag;
        }
  }
}

WITH
Vertex(id) AS SELECT src FROM Edge UNION SELECT dest FROM Edge,
N AS SELECT count(*) FROM Vertex,
PR (srcId, pr) AS /* Base case initializes */
(
     /*PageRank to 1.0/N */
     SELECT id as srcId, 1.0/N AS pr FROM Vertex
)
UNION UNTIL FIXPOINT BY srcId
(    /* Recursive case */
     SELECT nbr, 0.15/N+0.85*sum(prDiff) /* produces deltas */
     FROM ( SELECT PRAgg(srcId, pr).{nbr, prDiff}
            FROM Edge, PR  /* deltas from prev. iteration */
            WHERE Edge.src = PR.srcId
            GROUP BY srcId)
     GROUP BY nbr
)
```

- Condition B. A vertex is aware of the initial states of its “neighbor” vertexes;
- Condition C. The delta-handler function must be associative, commutative and distributive.

These conditions together make asynchronous evaluations viable. Interestingly, although PageRank (Program 8.5) is not a monotonic Datalog query, it can be asynchronously evaluated by leveraging additional constraints — Condition A and B. Condition A and B guarantee no delta update for a vertex can get lost, and the final state of a vertex can be recovered by replaying all the deltas that propagated from the neighborhood. Condition A and B are graph-special properties instead of a general available property of Datalog programs, which means, graph-specific program properties are useful for optimizing the evaluation strategies.

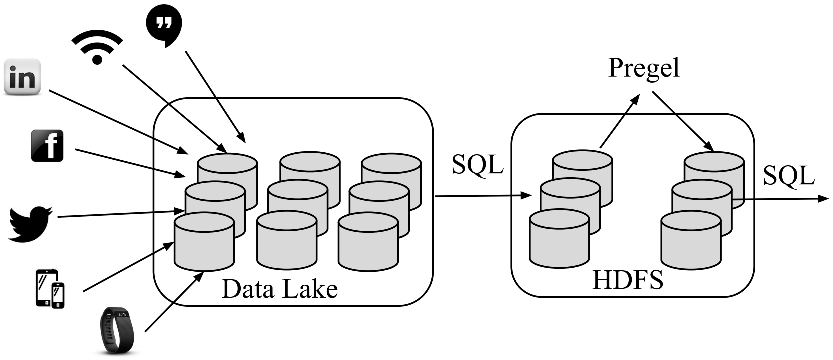


Figure 8.6: The enterprise Big Graph analytics flow.

Stratosphere [Ewen et al., 2012] (now Apache Flink[13]) and Naiad [Murray et al., 2013] are two general-purpose data-parallel dataflow engines that supports asynchronous incremental processing. Different from REX and Maiter, Naiad aims to support yet-another kind of incremental processing, where the delta comes from the source data in addition to the computation itself. A motivating application for Naiad is to compute and update connected components over a dynamically changed Twitter graph in real-time. In spite of the added source of deltas, the internal asynchronous dataflow engine of Naiad is similar to that in Myria [Wang et al., 2015]. Note that the application of the Naiad-style incremental processing is also limited to monotonic queries as we discussed in Section 8.1.

8.4 Integrated Analytical Pipelines

Our previous discussions assume that there is an existing graph for analysis. However, Big Graph analytics are not only about graphs. Many real-world applications do not have a pre-loaded graph dataset for analysis, and instead, graphs are often constructed by querying the data that are dynamically ingested into an enterprise data lake[14], i.e.,

[13] Flink: `http://flink.apache.org`
[14] Data Lake: `https://en.wikipedia.org/wiki/Data_lake`

a storage repository (e.g., HDFS[15]) that holds a vast amount of raw data. Figure 8.6 shows a dataflow for richer forms of graph analytics, i.e., end-to-end analytical pipelines spanning from the raw data to the final mined insights. As the figure indicates, various data from web, mobile, and IoT (internet of things) applications are continuously ingested into a data lake. In the meantime, data scientists run SQL queries using typical SQL-on-Hadoop systems (e.g., Hive [Thusoo et al., 2010], Impala [Kornacker et al., 2015], or Tez[16]) to extract graphs (e.g., in the adjacency list representation) of interests from the raw data, put the intermediate graph data onto HDFS, run distributed graph algorithms using typical graph analytical systems (e.g., Pregel [Malewicz et al., 2010], Giraph [Ching et al., 2015], or PowerGraph [Gonzalez et al., 2012]), and finally run another set of SQL queries over the graph computation results to generate user-readable reports [17]. In this process, a data scientist needs to figure out the physical locations of the intermediate results, manage their life-cycles, determine their formats, and write customized client scripts to glue SQL-on-Hadoop systems and graph analytical systems together through HDFS. All these tedious ETL (extract, transform, load) tasks draw the data scientist's energy from thinking of the analytical task at a logical level.

Several systems have tried to address the issue. GraphX has been integrated with Spark to allow a user to run together a wide range of Spark ecosystem tools, such as SparkSQL[18], DataFrames[19], Spark-R[20] and so on. In Aster Data, a commercial parallel data warehouse, Pregel-like, user-defined graph analytics functions could be invoked from SQL queries [Simmen et al., 2014]. Similar to GraphX and Aster Data, the integration of Pregelix and AsterixDB [Alsubaiee et al., 2014] also offers users a single, logical entry for richer graph analytics, by providing them a "Run Pregel" statement in the query language [Bu, 2015]. The

[15] HDFS: https://hadoop.apache.org/

[16] Tez: https://tez.apache.org/

[17] Hive+Giraph: `https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920`

[18] SparkSQL: `https://spark.apache.org/sql/`

[19] DataFrames: `http://ddf.io/`

[20] Spark-R: `https://spark.apache.org/docs/latest/sparkr.html`

indexing capability of AsterixDB, as well its built-in support for semi-structured data and its ability to work against data that does not fit into main memory, further expends the landscape of well-supported analytics.

Let us take the following example query in Aster Data (from literature [Simmen et al., 2014]) to sketch the concept.

```
1  -- DDLs (just for explaining the schemas)
2  CREATE TABLE Callers (callerId varchar, city varchar)
3   DISTRIBUTE BY HASH (callerId);
4  CREATE TABLE Calls (callerIdFrom varchar,
5   callerIdTo varchar) DISTRIBUTE BY HASH (callerIdFrom);
6
7  -- Query
8  SELECT city, SUM(pagerank) AS sum
9  FROM PageRank(
10     ON Callers AS ‘‘Vertices’’ PARTITION BY callerId
11     ON Calls AS "Edges" PARTITION BY callerIdFrom
12     ON (SELECT COUNT(*) FROM Callers) AS ‘‘Total’’
13     DIMENSION
14       TargetKey (‘callerIdTo’)
15       Accumulate (‘city’)
16       Directed (‘True’)
17       DampFactor (‘0.85’)
18       Threshold (‘1E-8’))
19 GROUP BY city
20 ORDER BY sum DESC
21 LIMIT 10
```

The above SQL query finds the 10 most popular cities in the call network of a telecommunications company. The graph for analysis is built dynamically in the query. Vertexes in the graph are formed from callers while edge adjacency lists are constructed from calls. The PageRank for each caller in the network is computed and aggregated by city. The final result consists of top 10 aggregated scores. Line 9 to 18 invoke a user-defined, Pregel-style, vertex-centric PageRank function with parameters including vertexes, edges, total number of vertexes, the vertex key (‘callerIdTo’), the attached value in each vertex (‘city’), whether the graph is directed, the damping factor, and the convergence threshold. Under the hood of Aster Data, there is a distributed query execution engine as well as distributed graph computation engine and intermediate data is routed back and forth between the two. However, all those low-level details are transparent to data scientists.

Distributed GraphLab [Low et al., 2012] is also adding more and more tabular data processing capabilities into its runtime[21], and Giraph [Ching et al., 2015] has built connectors for Hive [Thusoo et al., 2010] and HBase[22]. However, the two graph processing systems still require users to interact with multiple data processing platforms to achieve a complex analytical task.

8.5 Summary

This section has surveyed literature in the recursive query abstraction, dataflow-based graph analytical systems, incremental graph processing, and integrated analytical pipelines. Here is a brief summary:

- The theoretical results obtained in recursive query processing let us better understand existing Big Graph analytical systems, regarding to expressive powers, evaluation strategies, and potential limitations.
- A large set of graph-specific optimizations can be recast to DBMS-style query optimization and query evaluation techniques (as shown in paper [Bu et al., 2014], [Gonzalez et al., 2014], and [Jindal et al., 2014b]) while others require the knowledge of graph-only properties (i.e., the optimization in Maiter [Zhang et al., 2014]). Dataflow-based architecture is a practical solution for existing DBMS vendors to add their support for graph analytics.
- Building a unified programming model to support batch graph processing, incremental graph processing, and integrated graph analytics in a single system is an interesting direction for further research explorations.

[21] SFrame: https://dato.com/products/create/docs/generated/graphlab.SFrame.html

[22] HBase: `http://hbase.apache.org/`

Part III

Miscellaneous Issues

9

More on Single-Machine Systems

In Chapter 5.2, we have reviewed four standalone systems that adopt vertex-centric shared memory abstraction, and support out-of-core execution. In this chapter, we introduce other single-machine systems whose computation models are not purely vertex-centric. Specifically, Section 9.1 introduces two systems that adopt matrix backends for execution but expose a vertex-centric interface to users, and Section 9.2 describes four in-memory systems whose APIs expose high parallelism for execution in a high-end server. We will see more single-machine systems that use new hardware technologies to improve the performance of execution in Chapter 10.

9.1 Vertex-Centric Systems with Matrix Backends

We now review the two single-machine systems GraphMat [Sundaram et al., 2015] and GraphTwist [Zhou et al., 2015], which expose a vertex-centric programming interface to users, but perform matrix-based execution.

9.1.1 GraphMat

GraphMat [Sundaram et al., 2015] is a single-machine in-memory system that takes users' vertex-centric programs and maps them to generalized sparse matrix operations in the backend. The benefits are twofold: (1) users can use the familiar vertex-centric programming model to design graph algorithms, while for execution, (2) GraphMat is able to achieve performance comparable to native, hand-optimized code, by leveraging decades of research on techniques to optimize sparse linear algebra in High Performance Computing.

Vertex-Centric API. GraphMat adopts a BSP programming model similar to that of Giraph. Unlike PEGASUS, GraphMat maintains a boolean array to indicate whether each vertex is active, and in each iteration, this array is scanned to find the active vertices for message generation. Users define the following UDFs: (1) *send_msg*(.), called by a vertex v to read its value and to produce a message, which is then broadcast to all v's neighbors; (2) *process_msg*(.), which reads the message, the value of the edge along which the message arrives, and the data of the destination vertex, to postprocess the message value; (3) *reduce*(.), a commutative function that takes the postprocessed messages for a vertex, and produces a single reduced value; (4) *apply*(.), which updates the vertex value according to the reduced value. In an iteration, these four UDFs are called on the related vertices/edges in order, and only those vertices whose values get updated will become active for the next iteration.

As an example, consider the problem of finding the shortest path distance from a vertex s to every vertex in a graph, where each vertex v maintains a value $a(v)$ indicating the current distance estimation. Here, $v.send\_msg(.)$ assigns $a(v)$ to the message value, $(u, v).process\_msg(.)$ adds the weight of (u, v) to the message, *reduce*(.) computes the minimum message value min, and $v.apply(.)$ updates $a(v)$ as min if $min < a(v)$.

Notably, compared with other vertex-centric systems, the UDF *process_msg*(.) of GraphMat not only allows a user to access the value of the corresponding edge, but also the data of the destination vertex.

Compared with the conventional GAS model, this relaxation is able to cover a larger range of graph algorithms efficiently, such as triangle counting. Specifically, in triangle counting, a vertex u needs to compute the intersection between $\Gamma(u)$ and each neighbor list $\Gamma(v)$ that it receives from a neighbor $v \in \Gamma(u)$, and thus $(v, u).process\_msg(.)$ will access $\Gamma(u)$ in its computation.

GraphMat saves the memory cost by using one message per vertex as opposed to one message per edge, which is possible since $v.send\_msg(.)$ generates the same message to all neighbors. GraphMat can also avoid redundant data copies while generating messages. For example, in the above triangle counting example, the messages sent by v (whose values are the neighbor list $\Gamma(v)$) need not be copied but are rather passed as pointers to the actual data of $\Gamma(v)$. This is possible since GraphMat performs in-memory graph processing in a single machine.

Implementation. GraphMat translates the above vertex-centric program with $process\_msg(.)$ and $reduce(.)$ functions into a generalized sparse matrix-vector multiplication $\vec{v}_{out} \leftarrow A^T \cdot \vec{v}_{in}$, for efficient parallel execution. Specifically, for each **active** vertex v_j, Column j of A^T (denoted by $A^T_{*,j}$) is processed as follows: for each neighbor $v_k \in \Gamma(v_j)$, $(v_j, v_k).process\_msg(.)$ is called to compute a postprocessed message by accessing v_j's value $\vec{v}_{in}[j]$, the edge value $A^T_{k,j}$, and the data of v_k such as $\Gamma(v_k)$. The postprocessed message is then combined to $\vec{v}_{out}[k]$ by calling $reduce(.)$.

In the above algorithm, matrix A^T is represented in a Compressed Sparse Column (CSC) format, and partitioned into many chunks to improve parallelism and load balancing; while a vector (e.g., $\vec{v}_{in}$) is represented with a bitvector for storing valid indices and an array with constant size $|V|$ but only storing values at the valid indices. To improve performance, the bitvector of $\vec{v}_{in}$ is cached and shared among multiple threads that together performs the matrix-vector multiplication.

9.1.2 GraphTwist

GraphTwist [Zhou et al., 2015] is a single-machine system for iterative graph computation, with the following two unique features. Firstly, GraphTwist adopts a multi-level graph partitioning to (1) enable more balanced parallel workloads among computing threads (even for graphs with skewed vertex degree distribution), and to (2) allow the system to choose the right granularity of graph parallel abstractions for different graphs and different applications, so that each partition can be processed in the main memory of a commodity PC. In comparison, earlier systems (e.g., GraphChi) usually support only one level of granularity (e.g., shard). Secondly, GraphTwist adopts a randomized sampling-based approach to significantly reduce the computation workload, while achieving good approximation with bounds on the introduced error. We now introduce GraphTwist as follows.

Graph Partitioning Granularities. GraphTwist regards a directed weighted graph G as a 3D cube with three dimensions: source vertex u, destination vertex v, and the weight of edge (u, v), denoted by w. Note that each element (u, v, w) in the cube corresponds to an edge in G, and thus a partitioning of the cube is also a partitioning of the edges in G. Here, the weight of an edge can be either a numeric value, or a label (a.k.a. categorical value).

GraphTwist supports four levels of partitioning granularities of the cube: slice, strip, dice and vertex cut. We now describe them, where we assume that all source (resp. destination) vertices are ordered by ID and partitioned into a set of ranges $\mathbb{S}$ (resp. $\mathbb{D}$), and that the weights are ordered and partitioned into a set of weight ranges $\mathbb{W}$.

Let S (resp. D, and W) be an arbitrary set in $\mathbb{S}$ (resp. $\mathbb{D}$, and $\mathbb{W}$). A dice (S, D, W) consists of all edges (u, v, w) such that $u \in S$, $v \in D$, and $w \in W$. Intuitively, a dice is the smallest cubic unit in the cube partitioned by $\mathbb{S}$, $\mathbb{D}$, and $\mathbb{W}$. Since different graph applications often use either in-edges or out-edges, to provide efficient access for different graph applications, a dice (S, D, W) is physically organized in two types: (1) in-edge dice, where all edges in the dice are regarded as in-edges to vertices of D, and are stored in the order of source vertices; (2) out-edge dice, where all edges in the dice are regarded as out-edges from vertices of S, and are stored in the order of destination vertices.

A strip represents a larger partition unit than a dice, which groups dices either along $\mathbb{S}$ or along $\mathbb{D}$. Intuitively, a strip can be viewed as a sequence of dices stored physically together. There are two types of strips. An in-edge strip (D, W) contains all in-edge dices (S, D, W), $\forall S \in \mathbb{S}$, and an out-edge strip (S, W) contains all out-edge dices (S, D, W), $\forall D \in \mathbb{D}$.

Slice partitioning is introduced as an effective parallel abstraction to deal with the skewed edge weight distribution. It partitions the 3D cube into slices along dimension $\mathbb{W}$, so that edges with similar weights are clustered into the same slice. There are two types of slices corresponding to each weight range W. An in-edge slice contains all in-edge strips (D, W), $\forall D \in \mathbb{D}$, and an out-edge slice contains all out-edge strips (S, W), $\forall S \in \mathbb{S}$.

Finally, a dice can be further partitioned into vertex cuts, one per vertex. Specifically, since edges in an in-edge dice (S, D, W) are stored in the order of source vertices, it can be split into multiple out-edge cuts $(\{u\}, D, W)$ (or simply (u, D, W)), $\forall u \in S$. Similarly, an out-edge dice (S, D, W) can be split into multiple in-edge cuts (S, v, W), $\forall v \in D$.

Computation Model. Like in GraphMat, users only write iterative graph algorithms with a vertex-centric scatter-gather programming interface. GraphTwist compiles the vertex-centric code, and carries out the iterative computation on each vertex in the proper granularity (i.e., slice by slice, strip by strip, dice by dice, cut by cut). To support fast access to different kinds of partitions, GraphTwist builds a partition-level index and a vertex-to-partition index for an input graph. Specifically, (1) the dice-level index is a dense index that maps a dice ID (S, D, W) to the corresponding chunks on disk where the dice block is physically stored; (2) the strip-level index is a sparse index on top of the dice-level index, which maps a strip ID (D, W) (or (S, W)) to all the dice-level index entries relevant to that strip; (3) the slice-level index is a sparse index on top of all the strip-level index, which maps each weight range W to the strip-level index entries relevant to this slice; (4) the vertex-to-partition index maps each vertex to the set of partitions that contain in-edges or out-edges of this vertex.

GraphTwist partitions the 3D cube in a divide-and-conquer manner: (1) it first partitions the cube into slices; (2) if a slice cannot be processed in main memory, it is further partitioned into strips; (3) if a strip cannot be processed in main memory, it is further partitioned into dices. Therefore, the edges in different partitions are disjoint, and thus different partitions can be processed in parallel. Moreover, different vertices in the same partition can also be processed in parallel.

Each iteration of computation consists of two steps: (1) to calculate partial vertex updates in each subgraph partition in parallel; and (2) to aggregate partial vertex updates from all subgraph partitions to generate complete vertex updates. Note that for a vertex with a high in-degree, its value update may depend on multiple partitions, and such a vertex is called a *critical border vertex*. Since the in-edges of a critical border vertex v belong to multiple partitions, denoted by $P_1, \ldots, P_m$, GraphTwist maintains a partial update list for v in memory (to avoid conflict), with an initial counter of 0. A thread that processes P_i puts the partial update of v to the partial list and increments the counter by 1. If the counter reaches m, the thread then continues to perform the *Gather* process to aggregate all partial updates of v in its partial list, to generate a complete update for v.

Fast Randomized Approximation. GraphTwist executes graph computation by a series of matrix-vector computations, or more generally, matrix-matrix computations. To speed up the iterative computation, GraphTwist supports the pruning of statistically insignificant vertices or edges. To illustrate the idea, one may perform PageRank computation only using the high-weight edges (with proper value adjustments to guarantee unbiased estimation), and the results tend to preserve the ranking order of important vertices.

The first pruning technique is slice pruning (or subgraph-based pruning), which operates on matrix multiplication $A \cdot B$ as follows. Firstly, an importance score is computed for each slice of A and B, as the Frobenius norm[1] of the slice's matrix representation M where $M[u][v] = w$ if (u, v, w) is in the slice and $M[u][v] = 0$ otherwise. Let W_A (resp. W_B) be a slice of A (resp. B), and let its importance score be p_A (resp. p_B). Then, the slice pruning approach computes $A \cdot B$ by sampling (W_A, W_B) pairs with probability proportional to $p_A \cdot p_B$, and

[1] http://mathworld.wolfram.com/FrobeniusNorm.html

computes $W_A \cdot W_B$ for each sample. The results computed from the sample pairs are then used to compute an unbiased estimator of $A \cdot B$.

Note that the computation of $W_A \cdot W_B$ can be further decomposed into a few strip-level multiplications $(S, W_A) \cdot (D, W_B)$, $\forall S \in \mathbb{S}, \forall D \in \mathbb{D}$ for parallel execution. To compute a strip-level multiplication $(S, W_A) \cdot (D, W_B)$ efficiently, GraphTwist decomposes the in-edge strip (S, W_A) (resp. the out-edge strip (D, W_B)) into multiple out-edge cuts (u, D, W), $\forall u \in S$ (resp. in-edge cuts (S, v, W), $\forall v \in D$). Thus, the strip-level multiplication can be efficiently estimated by randomized sampling on vertex cut pairs, in a similar manner as in slice pruning. This method is called cut pruning (or vertex-based pruning).

9.2 In-Memory Systems for Multi-Core Execution

There are also a few systems designed to process a big graph in the main memory of a single high-end server with large RAM space and tens of cores. In this section, we introduce four such systems, Green-Marl [Hong et al., 2012], Ligra [Shun and Blelloch, 2013], GRACE [Xie et al., 2013] and Galois [Nguyen et al., 2013]. Green-Marl, Ligra, and GRACE focus on the programming simplicity of developing parallel graph algorithms, while GRACE and Galois aim at the full utilization of all cores in a machine.

Compared with distributed big graph systems, these systems eliminate the expensive network communication, and are often able to achieve performance comparable to or even better than distributed systems. However, a high-end server is required which can be more expensive than a cluster of commodity machines. The graph size is also limited by the available RAM space, e.g., the largest graph tested on GRACE has less than 300 million edges [Xie et al., 2013]. The biggest problem, however, is the high startup overhead. Specifically, consider the processing of a graph with size 100GB. In a distributed system running with 100 PCs, each PC only needs to load around 1GB data from HDFS. In contrast, a single-machine in-memory system needs to load all the 100GB data from its local disk to main memory before starting processing, with which time a distributed system may have already finished many graph jobs. This is not an issue for an out-of-core single-machine system since the computation itself scans the disk-resident graph.

9.2.1 Green-Marl

Green-Marl [Hong et al., 2012] is a domain-specific language (DSL) that provides a set of high-level language constructs. These constructs allow developers to describe their graph analysis algorithms intuitively, but meanwhile expose the data-level parallelism inherent in the algorithms for efficient parallel execution. The Green-Marl compiler translates high-level algorithmic description written in Green-Marl into a C++ program (rather than a machine language), for efficient parallel execution in the main memory of a single machine. This approach allows compiler-level optimizations to be used to improve the performance of the generated code. Recently, [Hong et al., 2014] extended this compiler to generate Pregel programs for distributed processing in the GPS system [Salihoglu and Widom, 2013].

The Green-Marl language includes language constructs for implicit parallelism, and meanwhile allows users to explicitly specify parallel execution regions. For example, users may process a collection of data using the *Foreach* statement, which indicates that the order of processing is unimportant. The *Foreach* statement is executed similarly as the parallel-for loop of OpenMP: multiple threads are forked to process the data in parallel, and are then synchronized at a join point before ending the for loop. One may also use the *For* statement of Green-Marl to indicate sequential execution of the data by one single thread (e.g., due to data dependencies).

As a DSL for graph algorithms, the *Foreach* or *For* statement can iterate through the following ranges: (1) all vertices in a graph, (2) all in-neighbors or out-neighbors of a vertex, (3) all up-neighbors or down-neighbors of a vertex, which we explain next. Up-neighbors and down-neighbors are only defined during a BFS from a specific node s: let v be a vertex i hops from s, then an up-neighbor (resp. down-neighbor) of v is an in-neighbor (resp. out-neighbor) of v that is $(i-1)$ (resp. $(i+1)$) hops from s.

Green-Marl also provides two graph traversal schemes: breadth-first search (BFS) and depth-first search (DFS) from a root s, which are specified by the *InBFS*(s) and *InDFS*(s) statements, respectively. DFS implies sequential execution, while BFS implies level-synchronous parallel execution, i.e., vertices with the same distance from s are visited concurrently but execution is synchronized before moving on to the next level. Inside the *InBFS*(s) statement, one may iterate up-neighbors or down-neighbors using for loop. The BFS statement also allows an optional branch of reverse BFS, which is useful for implementing algorithms like computing strongly connected components (which requires a BFS followed by a reverse BFS [Yan et al., 2014b]).

Other useful language constructs include reductions (e.g., SUM, MIN, AND) and deferred assignment. Deferred assignment (as specified by operator <=) is used to support the BSP model, where if users write $a <= f(\vec{b})$, then $\vec{b}$ will always use the old values and the write to a becomes effective only at the end of the binding iteration.

9.2.2 Ligra

Ligra targets at making graph traversal algorithms easy to write, and meanwhile, supporting efficient shared-memory parallel execution that adapts the computation mode to the number of active vertices. Unlike existing vertex-centric systems, Ligra routines operate on a subset of vertices U. More specifically, Ligra supports two routines, *edgeMap*(.) and *vertexMap*(.), which we describe below.

EdgeMap. *edgeMap*(U, F, C) operates on a set of vertices U (considered active) as follows: for each out-edge (u, v) of a vertex $u \in U$, it checks whether the target vertex satisfies C (i.e., whether $C(v) = \mathit{true}$). If so, function F is applied to (u, v), and F also decides whether v should be included into the output vertex set U' (e.g., the active vertices for the next iteration). Here, both functions $C(v)$ and $F(u, v)$ are user-defined, and since F may update the vertex value $a(v)$, users need to use atomic operation for the update to avoid write conflicts.

We now illustrate how to write $C(v)$ and $F(u, v)$ for graph algorithms in the BSP model, where U (resp. U') is the set of active vertices for the current (resp. next) iteration. Consider the BFS algorithm, where an array $parent[]$ is maintained such that $parent[v]$ records the parent of v in the BFS tree, which is initialized as *null*. Each iteration performs *edgeMap*(U, F, C) to activate the new level of vertices U', where $C(v)$ checks whether $parent[v] = null$ (i.e., v has never been visited), and if so, $F(u, v)$ sets $parent[v]$ as u using an atomic compare-and-swap instruction, which checks whether $parent[v] = null$ again since another thread may have updated $parent[v]$ after the last call of $C(v)$.

VertexMap. *vertexMap*(U, F) operates on a set of vertices U as follows: for each vertex $u \in U$, it runs the user-defined function F on u, which also decides whether v should be included into the output vertex set U'. Some graph algorithms require using both *edgeMap* and *vertexMap*, such as PageRank computation which we describe below.

In this algorithm, two arrays $p_{cur}[]$ and $p_{next}[]$ are maintained, where $p_{cur}[v]$ (resp. $p_{next}[v]$) records the PageRank value of v in the current (resp. next) iteration. When an iteration begins, $p_{next}[v] = 0$ for all $v \in V$. Function *edgeMap*(V, F, C) operates on all vertices in V first, where $C(v)$ always returns *true*, and $F(u, v)$ atomically add $(p_{cur}(u)/d_{out}(u))$ to $p_{next}[v]$. Then, *vertexMap*(V, F) operates on every $v \in V$, where $F(v)$ adjusts the value of $p_{next}[v]$ with the damping factor, assigns the value to $p_{cur}[v]$ and reinitializes $p_{next}[v]$ as 0, to be used in next iteration.

Mode Switch Based on Vertex Sparseness. Depending on the size of U (i.e., sparseness of active vertices) during the execution, Ligra switches between a sparse and dense representation of U (which is a set of integer IDs). Accordingly, routine *edgeMap*(.) also switches its execution algorithm based on the representation of U. When U is small, *edgeMap*(.) iterates (in parallel) through every vertex $u \in U$ to process its out-edges (in parallel). But this approach is inefficient when U is large, since a vertex $u \in U$ needs to check every out-neighbor v (e.g., to compute $C(v)$), and a vertex v may be checked for multiple times by its in-neighbors.

Therefore, when U is large, *edgeMap*(.) iterates (in parallel) through every vertex $v \in V$, and if $C(v) = true$, then $F(u, v)$ is executed for each in-neighbor $u \in \Gamma_{in}(v) \cap U$. This method is faster since for each vertex $v \in V$, $C(v)$ is performed for only once. Moreover, for each vertex

v, we can specify $F(u, v)$ to be executed in serial among u to allow early termination. For example, in BFS, for each vertex v, $F(u, v)$ only needs to be executed for one in-neighbor u. Accordingly, $F(.)$ no longer needs to update $a(v)$ atomically. This optimization not only improves the performance of BFS, but also many other graph problems that perform multiple BFSs, such as betweenness centrality and graph radii estimation.

9.2.3 GRACE

Many vertex-centric graph algorithms are computationally light, which exhibit a high "data access to computation" ratio. When such an algorithm is processed in the main memory of a multi-core machine, the performance hits an early "memory wall" when we increase the number of used cores. To achieve reasonable speedup with all the cores in a machine, GRACE [Xie et al., 2013] adopts a block-centric execution model, in which computation is iterated locally over blocks of highly connected nodes to improve the cache hit rate. Blocks are disjoint and are obtained by graph partitioning using METIS.

Meanwhile, GRACE only requires users to write familiar vertex-centric programs, and regards the computation within a block B as a scheduling of vertex-centric computation among the vertices in B.

There are two levels of computations: block-level computation and inner-block computation. In block-level computation, blocks are scheduled for processing by a block-level scheduler, which, for example, can give priority to a block that contains the vertex with the worst residual error. To process a block B, an inner-block scheduler iterates over the vertices inside B (e.g., in a round-robin fashion), and applies the user-defined vertex-centric computing logic to these vertices. The scheduling policy of both the block-level scheduler and the inner-block scheduler can be specified by users, or users may use the pre-defined schedulers.

Unlike in Blogel (see Section 4.1.2), depending on the convergence rate, some blocks may be processed for more times than other blocks; also, a vertex may be processed for multiple times during the processing of its block. The processing of a block B ends either when the values of its vertices converge, or when a user-specified maximum iteration number is reached.

In a block B, if a neighbor u of a vertex $v \in B$ resides in another block B', then u is called as a *boundary vertex* to B. Note that the value of u is read by B (when processing v), and it may also be updated when processing B'. To guarantee serializability while permitting the concurrent processing of both B and B', GRACE implements a simple form of multi-version concurrency control, where B (as well as B') reads the old value of u while B' writes to a replica of u that gets committed when B' finishes its processing.

9.2.4 Galois

Like Green-Marl, the Galois system [Nguyen et al., 2013] let users write graph algorithms using a domain specific language (DSL) so that the system can utilize the exposed parallelism to fully utilize the CPU resources. Galois adopts a data-centric programming model called amorphous data-parallelism (ADP) [Pingali et al., 2011], which enables speculative execution (or optimistic concurrency control) to fully use any extra CPU resources. Specifically, the computation of a vertex v needs to read/write data from/to v's direct neighbors, and if the neighborhoods of two vertices v_a and v_b overlap, then conflicts may happen when v_a and v_b are processed in parallel. In this case, speculative execution allows both vertices to be processed (by two threads), but if a conflict happens, one of the conflicting computations is rolled back.

Application programmers of Galois specify parallelism implicitly by using an unordered-set iterator which iterates over a worklist of active vertices. The worklist is initialized with a set of active vertices before the iterator begins execution, and the execution of an iteration can create new active vertices, which are added to the worklist when that iteration completes execution. To ensure serializability of iterations, programmers must use a library of built-in concurrent data structures for graphs, worklists, etc. When there are no application-specific execution priorities, Galois adopts a machine-topology-aware scheduler to schedule active vertices for processing. Priority scheduling can also be adopted which is layered on top of this scheduler.

To understand the design of the machine-topology-aware scheduler, we first review the CPU architecture in a machine. Specifically, the motherboard contains many CPU sockets, and each socket can accommodate one commodity multi-core CPU (or CPU package). The machine-topology-aware scheduler uses a concurrent bag to hold the set of pending tasks (i.e., active nodes), which supports the concurrent insertion and retrieval of tasks. We now review the structure of a bag which is distributed among all CPUs. Specifically, each core maintains a stack of tasks (implemented as a ring-buffer with a size limit) for pushing/popping tasks. Each CPU also maintains a list of such stacks, called package-level list. When the stack associated with a core becomes full, it is moved to the package-level list of its CPU; while if the stack associated with a core becomes empty, the core fetches a full stack of tasks from the local package-level list if the list is not empty, or otherwise, the core needs to probe the package-level lists of other CPUs.

The priority scheduler of Galois, called *obim*, uses a sequence of bags instead of just one bag, to implement "soft" priorities. Tasks in the same bag have the same priority, while different bags have different priorities. Galois processes the tasks in a higher-priority bag first, before processing any task in a lower-priority bag. The bags are maintained by a global structure called *global map*. To improve locality, each core also maintains a lazy cache of the global map, called a *local map*. When a core (or its associated thread) cannot find a bag of required priority for inserting or retrieving tasks in its local map, it needs to synchronize with the global map. Since the global map is a central data structure that is read and written by all threads, it is represented as an append-only log-based structure, and a thread synchronizes with the global map by replaying the global log from its last synchronized log entry.

9.3 Summary

Both the systems described in Chapter 5.2 and those described in Chapter 9.1 are single-machine systems with vertex-centric API. The difference is that each system described in Chapter 5.2 adopts a backend specially designed for the respective vertex-centric (or edge-centric) API, while GraphMat and GraphTwist map vertex-centric API to matrix backend for execution. It remains an interesting question to answer whether a dedicated vertex-centric backend is more efficient, or a matrix backend is, which may benefit from the years of HPC research on optimizing matrix operations (e.g., better utilization of cache locality). The systems described in Chapter 9.2 demonstrate a few other programming interfaces that could be more flexible (though of lower level) than the vertex-centric interface, and that enables additional optimizations specified to the shared memory environment.

10

Hardware-Accelerated Systems

In this chapter, we introduce those graph analytics systems that accelerate their computation by applying new hardware technologies. Specifically, Section 10.1 introduces systems for out-of-core execution with SSDs (instead of disks) to achieve higher IO throughput and allow better parallelism, while Section 10.2 describes systems that use GPUs to achieve massive parallelism in a shared memory environment. It happens that all the systems reviewed in this chapter are single-machines systems, but hardware acceleration has also been applied to distributed systems recently, as we shall briefly discuss in Section 10.3.

10.1 Out-of-Core SSD-Based Systems

The single-machine out-of-core systems discussed in Chapter 5.2 all focus on iterative graph algorithms where each iteration scans the entire disk-resident graph. As a result, their designs all require that vertex IDs be consecutive integers that can be partitioned into intervals, and the execution of each iteration is essentially a hash-join between vertex values and edge values/updates, where the hash function maps a vertex ID to the corresponding interval.

However, when a graph is stored on SSDs, it is very important to support asynchronous random I/O requests in order to better overlap computation with I/O. This is because, compared with a magnetic disk with only one access arm, an SSD is composed of many flash chips, each with multiple dies that can read/write independently. Multiple asynchronous I/O requests can be submitted to an SSD in parallel, which are pipelined to achieve very high throughput. Computation may continue after an asynchronous I/O request is submitted, and when the transmission finishes, the data can be processed by a callback function.

In this section, we introduce two systems, TurboGraph [Han et al., 2013] and FlashGraph [Zheng et al., 2015], that process a big graph stored on SSDs. These systems support asynchronous random I/O requests to SSDs, and use memory as a cache to pin/unpin data from/to SSDs. TurboGraph focuses more on the efficient processing of light-weight graph queries, while FlashGraph focuses more on the full utilization of the IO bandwidth of an SSD array.

10.1.1 TurboGraph

In TurboGraph [Han et al., 2013], users write their graph algorithms with engine-level graph primitives such as generalized matrix-vector computation, and breadth-first search. These operations are implemented under the *pin-and-slide* execution model of TurboGraph, which leverages the parallelism of multi-core and SSD-IO, and fully overlaps CPU processing with IO processing.

In TurboGraph, vertices are ordered by their IDs and stored along with their adjacency lists. The vertices are stored as a list of pages, and each page stores some vertices (along with their adjacency lists). In the adjacency list of a vertex v, each neighbor u is represented by a pair indicating (1) the page of u, and (2) the order of u among the vertices in the page.

TurboGraph maintains an in-memory page table, where each entry only stores the ID of the first vertex in a page. Since the number of entries is the same as the number of pages, the page table is small enough to be kept in main memory. There are two usages of the page table. Firstly, queries like finding 2-hop neighbors of a vertex s can be efficiently supported, since the page of s can be directly located by binary search on the page table. Secondly, given an adjacency list element $(page(u), order(u))$, we can get the start vertex w of $page(u)$ from the page table, and compute u's ID as $(w + order(u))$. For a high-degree vertex v, its adjacency list may not fit into a single page, and thus TurboGraph stores the adjacency list using multiple pages, which are also tracked by the in-memory page table.

Two thread pools are maintained in TurboGraph, one for the execution threads, and the other for the asynchronous I/O callback threads. Meanwhile, a buffer manager is maintained to cache pages that are read from the SSD. We now consider how TurboGraph processes a set of vertices S represented the vertex vector $\vec{v}_{in}$, i.e., to perform $\vec{v}_{out} \leftarrow A^T \cdot \vec{v}_{in}$. In this case, only those columns of A^T that correspond to the adjacency lists of vertices in S need to be accessed. We only present the simple case where $\vec{v}_{in}$ and $\vec{v}_{out}$ fit in main memory, and thus the computation only needs to read adjacency lists (i.e., columns of A^T) from SSD. Otherwise, a block-nested loop (BNL) style algorithm is needed.

Specifically, to process S, TurboGraph first identifies the corresponding pages for these vertices, and pins those pages that are already in the buffer pool. Then, parallel asynchronous I/O requests are submitted to the SSD for those required pages that are not in the buffer pool. Note that TurboGraph does not wait for the completion of those I/O requests, but instead, the execution threads concurrently process vertices of V whose pages are already pinned. A page is unpinned as soon as it is processed (by either an execution thread or a callback thread), so that an execution thread may issue more asynchronous I/O requests to the SSD.

10.1.2 FlashGraph

FlashGraph [Zheng et al., 2015] is a semi-external memory graph engine that stores vertex state in memory and edge lists (i.e., adjacency lists) on SSDs, and its design goal is to achieve performance comparable to an in-memory engine. FlashGraph is built on top of a user-space SSD file system called SAFS (set-associative file system), which supports high-throughput asynchronous I/Os over an array of SSDs. During the computation, FlashGraph only accesses edge lists requested by the user-defined logic from SSDs, and requests to the same page or adjacent pages are merged into one sequential access to improve I/O throughput.

In FlashGraph, the edge lists of vertices are stored as a file on SAFS in the increasing order of vertex ID. An in-memory table is maintained, which stores the positions of the edge lists (in the edge list file) for every vertex whose ID is a multiple of 32. This structure is much more space-efficient than if we store the edge list position for every vertex, and with the help of another in-memory array that records the degree of every vertex, the edge list location of any vertex can be efficiently computed (for random access from SSD). FlashGraph also maintains an array of vertex states, but does not explicitly store the vertex IDs. When the user program needs to access the ID of a vertex, it is computed based on the address of its vertex state in memory and that of the first vertex.

FlashGraph exposes a vertex-centric programming interface to users, where vertices communicate with each other by message passing (and thus there are no race conditions). UDFs are provided for users to define (1) how to process a vertex when its edge list is returned from SARS, (2) how to process a message in memory, and (3) how to postprocess the vertex values (e.g., adjusting them by a damping factor in PageRank computation). FlashGraph requires a vertex to explicitly request its own edge list before accessing it, and this requirement can save I/O bandwidth. For example, if a vertex votes to halt directly in its computation, it does not need to access its edge list from SAFS. There are three possible states for a vertex v: (1) running, which indicates that v is in the middle of computation (e.g., v submitted a request to access its edge list, but the transmission is not finished yet); (2) active, which indicates that v is scheduled for processing; and (3) inactive.

FlashGraph splits V into multiple partitions and assigns a worker thread to each partition. Each worker thread maintains a queue of active vertices within its own partition and executes user-defined vertex programs on them. The threads also send and receive messages on behalf of their vertices and buffer messages to improve performance. More specifically, the vertices are first partitioned into intervals $I_1, I_2, \ldots$, and then assigned to the worker threads in a round-robin fashion. This scheme increases the chance of merging I/O requests. For example, assume that there are only two workers, then it is likely that worker 1 is processing vertices in I_1 when worker 2 is processing vertices in I_2, and then worker 1 is processing vertices in I_3 when worker 2 is processing vertices in I_4. Note that the edge lists of I_1 and I_2 (resp. I_3 and I_4) are stored consecutively in the file. Moreover, each worker thread schedules active vertices for execution in the order of vertex ID, which also increases the chance of merging I/O requests as the edge lists are ordered by vertex ID in the file on SAFS.

10.2 Systems for Execution with GPU(s)

Due to the success of general-purpose computing on graphics processing units (GPGPU), several recent works started to explore the potential of using GPU(s) for vertex-centric graph processing [Zhong and He, 2014, Fu et al., 2014, Khorasani et al., 2014]. These systems aim to provide users with a familiar vertex-centric programming interface, while their execution models are designed to fully explore the massive parallelism of GPU(s). Before introducing the individual systems, we first review the GPU architecture, and identify the important design issues for efficient execution with GPU(s).

GPU Architecture. A GPU is connected to the host CPU via PCI-e (PCI Express) bus. A GPU consists of an array of *streaming multiprocessors* (SMs), where each SM contains multiple *streaming processors* (SPs) (let the number be 32), and executes with the SIMT (single instruction, multiple threads) model. Specifically, when an SM executes an SIMD instruction, it is executed on all 32 threads (run by its 32 SPs). If different threads of an SM need to execute different control flows (e.g., different branches of an if-else block), the processor executes all paths, using masking to disable/enable the relevant threads as appropriate. As a result, a GPU program needs to be carefully designed to avoid path divergence that leads to GPU underutilization.

An SM typically contains multiple warps (let the number be 48), where a warp contains 32 threads to be concurrently executed with the 32 SPs. The warp scheduler of an SM issues one of its 48 warps for execution at a time, and thus an SM can process $48 \times 32 = 1536$ threads. In the more general case, an SM may be able to process multiple warps at a time, achieving not only intra-warp parallelism but also inter-warp parallelism in one SM. Each SM is associated with a private L1 cache and a low latency shared memory (scratchpad memory), while SMs share an L2 cache and the interface to a global memory.

In NVIDIA's CUDA computing architecture, developers write device programs called kernels. A kernel needs to be explicitly configured and invoked to run on a GPU, with many threads running the same kernel program in parallel. A GPU executes one or more kernels (or kernel grids), where each kernel grid consists of an array of thread blocks that execute the same kernel program. Each thread block is also called a cooperative thread array (CTA), which contains multiple warps (let the number be 6, and thus an SM can accommodate 48/6=8 CTAs). The CTAs of a kernel grid are enumerated and distributed to those SMs with available execution capacity. The threads of a CTA execute concurrently on one SM, and when the CTA is processed, it is released from the SM so that the SM can take more CTAs for processing.

Due to the memory hierarchy of GPU, the intra-warp parallelism, and the inter-warp parallelism within an SM (and also a CTA), coalesced (i.e. locality-aware) memory accesses are prefered, and a GPU program should avoid irregular memory accesses.

We now introduce three GPU-based vertex-centric systems, Medusa [Zhong and He, 2014], MapGraph [Fu et al., 2014] and CuSha [Khorasani et al., 2014]. Among them, Medusa adopts the BSP model of Pregel where vertices send messages to each other, while MapGraph and CuSha adopts the GAS model of PowerGraph.

10.2.1 Medusa

Medusa [Zhong and He, 2014] adopts the BSP model of Pregel for computation. However, instead of letting users define a single *compute*(.) function, Medusa provides a fine-grained programming model, called Edge-Message-Vertex (EMV), for users to specify (1) how to process each individual edge, vertex, and message; (2) how to process the edge-list of a vertex, and how to process the message-list received by a vertex; (3) how to apply a combiner to all edge-lists or message-lists. Compared with binding all computation of one vertex to one thread, the fine-grained EMV model avoids path divergence caused by different edge-list size and message-list size of individual vertices.

In a Medusa job, all vertices and edges are processed in each iteration by default, but users may indicate active vertices and edges so that Medusa will not process inactive vertices and edges. There are two ways of terminating a job: (1) users may specify the maximum number of iterations that the job is allowed to run, or (2) a user-defined function may explicitly signal the system to terminate.

Observing that in many graph algorithms, at most k messages are passed along each edge in an iteration, Medusa pre-allocates an array of $k \cdot |E|$ for buffering messages. Let us consider the common case where $k = 1$ for simplicity, then for the set of vertices $v_1, \ldots, v_n$, the first $d_{in}(v_1)$ elements are used for storing messages towards v_1, one for each in-edge of v_1; the next $d_{in}(v_2)$ elements are used for storing messages towards v_2, one for each in-edge of v_2; and so on. Each edge is also stored with the array position for the message sent along that edge. This scheme has two benefits: (1) write positions of the messages that are sent to the same vertex are consecutive, leading to coalesced memory accesses; (2) write positions of different messages do not conflict with each other.

When multiple GPUs are used, Medusa partitions the graph to the GPUs using METIS, to reduce the amount of data transfer on the host-device communication link (i.e., the PCI-e bus). For each cross-partition edge (u, v) where u (resp. v) is in partition P_u (resp. P_v), we assign (u, v) to P_v and replicate vertex u in P_v. In this way, the message from u to v can be directly emitted from the replica of u in P_v, rather than u itself (on another GPU different from the one processing v). Medusa also supports a multi-hop replication scheme to further reduce inter-GPU data transfer, with a tradeoff of having more edges to process.

10.2.2 MapGraph

MapGraph [Fu et al., 2014] targets at high-throughput graph processing with a single GPU. It adopts the GAS programming model as in PowerGraph: (1) the gathering phase computes a generalized sum for vertex v from the data of its adjacent edges and vertices; (2) the applying phase updates the value of v using the generalized sum; (3) the scattering phase distributes messages of v to its adjacent edges and vertices. However, unlike the asynchronous execution of PowerGraph, MapGraph executes in iterations, where vertices in the current frontier are processed and may activate their adjacent vertices to be included into the frontier for the next iteration. This design allows fine-grained processing on vertices and edges with large parallelism and small path divergence.

The topology of a graph is stored in the Compressed Sparse Row (CSR) format with two arrays: (1) *column-index*[], which is a concatenation of the adjacency lists of all vertices, and (2) *row-offset*[], which contains the indices indicating where each adjacency list starts, i.e., *row-offset*[i] is the starting position of the subarray of v_i's neighbors in *column-index*[].

Since the applying phase is embarrassingly parallel, MapGraph focuses on improving the performance of the gathering and scattering phase. The gathering phase adopts the *two-phase decomposition* strategy to achieve good load-balancing for threads within and across CTAs (which also applies to the scattering phase). Specifically, the processing is divided into two sub-phases, one for scheduling and the other for actual computation. The scheduling sub-phase assigns vertices into CTAs, so that the number of adjacent edges in each CTA is the same. Then, in the computation sub-phase, each thread accesses the same number of adjacent vertices and performs the same operation. While the scheduling sub-phase incurs additional overhead, the workload is balanced during computation.

The scattering phase only adopts the two-phase decomposition strategy when the frontier of active vertices is large. When the frontier is small (i.e., computation is sparse), a more efficient *dynamic scheduling* strategy is adopted for scattering. This strategy distributes the workload of vertices to threads of CTAs according to the vertex degree, so that the adjacent edges of a high-degree vertex can be concurrently processed by all threads in a CTA or a warp (depending on the concrete degree value). However, when the computation is dense, the total number of adjacent edges to process can vary largely among CTAs, leading to imbalanced workload among CTAs.

10.2.3 CuSha

Like MapGraph, CuSha [Khorasani et al., 2014] also adopts the GAS programming model. However, instead of organizing a graph in CSR format, it operates on shards similarly defined as in GraphChi (see Section 5.2.1) to achieve fully coalesced memory accesses.

To see why CSR leads to irregular memory accesses, assume that the vertex values are stored in an array *vertex-value*[] where *vertex-value*[i] stores $a(v_i)$. Recall from Section 10.2.2 that the edges (w.l.o.g., let them be in-edges) are stored as an array *column-index*[] that concatenates the adjacency lists of all vertices, i.e., each element *column-index*[j] stores the source vertex u of its corresponding edge, in the form of u's position in *vertex-value*[]. During scattering, edges in *column-index*[] are processed in parallel by GPU, to compute a value for each edge (u, v) from $a(u)$, to be gathered by v. However, consecutive elements in *column-index*[] may point to non-consecutive positions in *vertex-value*[], leading to non-coalesced memory accesses.

Like GraphChi, CuSha partitions vertices into P disjoint intervals (or shards), I_1, …, I_P. Each shard I_i stores the in-edges of all vertices whose IDs fall into I_i, and the edges are ordered by their source. As a result, the edges in shard I_i are stored as a sequence of P windows, $W_{1,i}$, …, $W_{P,i}$, where $W_{j,i}$ consists of those edges pointed from vertices in shard I_j. Each edge (u, v) is also stored with the value of the edge (denoted by $a(u, v)$), and the value of the source (i.e., $a(u)$).

Each shard I_i is processed by one CTA, by the following steps that only perform coalesced memory accesses: (1) For each edge (u, v) in I_i, we assume that the stored $a(u)$ is up-to-date, then the new value of $a(u, v)$ is computed from $a(u)$ and stored with (u, v). (2) Each edge (u, v) in I_i atomically aggregates $a(u, v)$ to the new value of $a(v)$ (note that $v \in I_i$). (3) The new values of vertices in I_i are propagated to every shard I_j, so that the field $a(u)$ of each edge (u, v) in window $W_{i,j}$ of I_j is up-to-date (for use by next iteration).

While guaranteeing coalesced memory accesses, the above method may suffer from imbalanced workload among CTAs. This is because each shard is processed by one CTA, but shards can have very difference sizes due to skewed vertex degree distribution. Moreover, Step (3) may suffer from inter-warp divergence in a CTA due to the difference in window sizes. There may be many small windows, causing most threads in a warp being idle. To solve this problem, CuSha also creates P concatenated windows CW_1, …, CW_P, where each concatenated window (abbr. CW) CW_i concatenates the windows of all shards that I_i needs to propagate its updated vertex values to, i.e., $W_{i,1}$, …, $W_{i,P}$. However, instead of storing each edge of $W_{i,j}$ in CW_i, a pointer to this edge in I_j is stored. In this way, for Step (3), consecutive threads within a CTA may process consecutive CW entries (which may now span across window boundaries) to improve GPU utilization.

10.3 Summary

The systems we discussed in this chapter use new hardware technologies mainly to improve performance in a shared memory environment on a single machine. This is not a coincidence since these systems target data-intensive jobs, which can substantially benefit from SSDs and GPUs. For a distributed graph system, on the other hand, network communication is often the bottleneck. For example, in PageRank computation, a vertex v computes each out-going message as $pr(v)/d_{out}(v)$ ($pr(v)$ is the current PageRank of v) in negligible time, but the time to transmit this message through the network is much higher.

There has recently been a few distributed systems designed to take advantage of high-speed network hardware technologies. For example, GraM [Wu et al., 2015] uses a specially designed multi-core aware RDMA-based communication stack that preserves parallelism in a balanced way (i.e., NUMA-aware) and allows overlapping of communication and computation, which achieves good performance with a

Mellanox ConnectX-3 InfiniBand NIC with 54Gbps bandwidth. Similarly, Chaos [Roy et al., 2015] scales out the disk-based execution of X-Stream with a cluster where each machine has a 480GB SSD, and where machines are connected by 40 GigE links. As expected, Roy et al. [2015] reported poor performance of Chaos when the normal Gigabit Ethernet is used. Gemini [Zhu et al., 2016] extends the hybrid push-pull computation model from shared-memory to distributed scenarios, and is designed to be NUMA-aware and locality-aware. It is reported to significantly outperforms existing distributed systems with the help of Infiniband EDR network with up to 100Gbps bandwidth.

In general, we remark that (1) while single-machine systems can readily benefit from new computation-intensive hardware, distributed graph systems should exploit advanced network technologies to avoid slow network from throttling the available parallelism; (2) if high-speed network is used in a distributed system, the out-of-core mode should use SSDs instead of hard disks in order to match the bandwidth of the network; and (3) the design of multithreading strategy is important when there are sufficient network bandwidth to utilize the parallelism of multiple cores in each machine.

11

Temporal and Streaming Graph Analytics

The systems described so far are designed to analyze static graphs. However, real world graphs often evolve over time, with vertices and edges continually being added or deleted, and their attribute values being frequently updated. Examples of such graphs 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. Analyzing these temporal graphs is crucial for gaining insights relevant to real-time decision making.

There are two somewhat orthogonal challenges here, that have resulted in two distinct bodies of work. First, there is often interest in doing *temporal analysis* over historical traces of graphs, often called *time-evolving graphs* or *historical graphs*; examples of such analysis tasks include network evolution, historical queries, and many others. Here the main computational challenges include storing very large volumes of historical data compactly, and retrieving the data needed for any specific temporal analysis task or historical query efficiently. There is also a need for better and more user-friendly high-level interfaces for specifying complex analytical tasks. Second, there is a need to do *real-time analytics on the streaming data* as it is being generated; here the scope of the analysis typically only includes the latest snapshot or the snapshots from a recent window. The key challenge here is to be able to deal with the high rate at which the data is often generated. Although these topics have received relatively less attention compared to static graph analysis, both of them have seen a flurry of activity in the recent years that we review in this chapter.

11.1 Overview

We first provide an overview of the existing temporal graph systems. We remark that while systems for incremental iterative data flows (see Section 8.3), such as Naiad [Murray et al., 2013] and Stratosphere [Ewen et al., 2012], can also be used for processing dynamic graphs, they are not specially designed for processing graphs. Also, several recent works have investigated, from a graph database perspective, the problems of storing and retrieving large-scale evolving graphs [Mondal and Deshpande, 2012, Ren et al., 2011]; however they do not consider complex graph analytics, such as influence analysis or community detection algorithms, as the graph engines surveyed in this section do.

Computation Model. In all of the systems that we survey, a temporal graph is viewed as a continuous stream of graph updates. A graph update can be an addition or a deletion of a vertex or an edge, or it can also be an update of an attribute associated with a node or an edge. Of the systems discussed here, Kineograph [Cheng et al., 2012], Chronos [Han et al., 2014b], DeltaGraph [Khurana and Deshpande, 2013], and LLAMA [Macko et al., 2015], all support general graph updates, whereas TIDE [Xie et al., 2015b] focuses on addition of vertices and edges in the context of dynamic interaction graphs, in which new interactions (edges) are continually added over time.

The frequent change of a temporal graph poses a significant challenge to algorithm design, because the overwhelming majority of graph algorithms assume static graph structures. One would have to design special algorithms for each application to accommodate the dynamic aspects of graphs. To support general-purpose computations, most of the temporal graph systems adopt a strategy to separate graph updates from graph computation. More specifically, although updates are continually applied to a temporal graph, graph computation is only performed on a sequence of successive *static* views of the temporal graph. For simplicity, most systems adopt a *discretized-time* approach, so that time domain is set of natural numbers, i.e., $t \in \mathcal{N}$. We use GV_t to denote the static view of a temporal graph G at time t. An analytic function F applied to a temporal graph G at time t is actually applied to GV_t, with the result $F(GV_t)$. As time advances to t', the result is updated to $F(GV_{t'})$ either by computing it from scratch on $GV_{t'}$ or by incrementally updating the result from $F(GV_t)$ to $F(GV_{t'})$.

Kineograph and TIDE both focus on point-in-time analysis that continually delivers the up-to-date results $F(GV_{t_{now}})$, where t_{now} is the current time (which is constantly changing); on the other hand, Chronos, DeltaGraph, and LLAMA are designed for analysis within a time range, i.e., computing results over a series of static views within a time range.

The separation of graph update and graph computation not only allows existing algorithms designed for static graphs to be used for analyzing temporal graphs, but also enables the same familiar computation interface in existing static graph processing systems to be used for temporal graphs. In fact, TIDE employs the message passing-based vertex-centric programming model like in Pregel [Malewicz et al., 2010] for analyzing dynamic graphs, while Kineograph and Chronos use the vertex-centric scatter-gather model (like the GAS model of GraphLab [Low et al., 2012]) in either a pull mode or a push mode. In addition, Chronos can also support the edge-centric model proposed in X-Stream [Roy et al., 2013].

Static Views of Temporal Graphs. The static view of a temporal graph, however, can be defined differently depending on the application requirements. The most straightforward definition is a *snapshot*. A *snapshot* of a temporal graph G at time t, denoted as G_t, is defined as the static graph formed by applying all the graph updates before t. Chronos, DeltaGraph, LLAMA, and Kineograph all adopt this snapshot definition of static view for temporal or streaming graph analysis. However, the snapshot model has two major drawbacks: 1) the ever-increasing size of a snapshot, especially for insertion-heavy graph updates, and 2) the growing reflection of the out-of-date characteristics of the temporal graph, due to the swelling proportion of the stale data in a snapshot. To address these drawbacks, TIDE proposes a novel *probabilistic-edge-decay* (PED) model to generate static views of temporal graphs. More details of this model are provided in Section 11.3.2.

Incremental Computation. To guarantee the timeliness of analysis, systems for dynamic graphs need to continually update results as time advances. The naive way is to recompute on a new static view from scratch, which is obviously expensive. Given the significant overlap of graph structures between two successive static views, is it possible to exploit the results at t to more efficiently generate results at $t+1$? The answer is yes, for iterative algorithms, such as Katz centrality [Katz, 1953] and PageRank [Brin and Page, 1998]. More specifically, the incremental computation can 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. For some single-source shortest path algorithms [Roditty and Zwick, 2011], such incremental computation is also possible if only edge insertions (deletions) are allowed. Unfortunately, some algorithms do not work correctly under this incremental scheme [Eppstein et al., 1999], so that recomputation from scratch is required.

Several of the systems that we survey support incremental computation for dynamic graph analysis, among which Chronos exploits this technique more heavily (for details, see Section 11.2.1).

11.2 Historical Graph Systems

Broadly speaking the focus of this work is on providing the ability to analyze and to reason over the entire history of the changes to a graph. There are many different types of analyses that may be of interest. For example, an analyst may wish to study the evolution of well-studied static graph properties such as centrality measures, density, conductance, etc., over time. Another approach is through the search and discovery of temporal patterns, where the events that constitute the pattern are spread out over time. Comparative analysis, such as juxtaposition of a statistic over time, or perhaps, computing aggregates such as *max* or *mean* over time, possibly gives another style of knowledge discovery into temporal graphs. Most of all, a primitive notion of just being able to access past states of the graphs and performing simple static graph analytics, empowers a data scientist with the capacity to perform analysis in arbitrary and unconventional patterns.

Supporting such a diverse set of temporal analytics and querying over large volumes of historical graph data requires addressing several data management challenges. Specifically, we need techniques for storing the historical information in a compact manner, while allowing a user to retrieve graph snapshots as of any time point in the past or the evolution history of a specific node or a specific neighborhood. Further the data must be stored and queried in a distributed fashion to handle the increasing scale of the data. We must also develop an expressive, high-level, easy-to-use programming framework that will allow users to specify complex temporal graph analysis tasks, while ensuring that the specified tasks can be executed efficiently in a data-parallel fashion across a cluster.

11.2.1 Chronos

Chronos [Han et al., 2014b] targets time-range graph analytics, requiring computation on the sequence of static snapshots of a temporal graph within a time range. An example is analyzing the change of each vertex's PageRank for a given time range. Obviously, the most straightforward approach of applying computation on each snapshot separately is too expensive. Chronos achieves efficiency by exploiting locality of temporal graphs.

In-Memory Graph Layout. There are two kinds of locality for temporal graphs that can be exploited for efficient data layout: *time* locality, where states of a vertex (or an edge) in consecutive snapshots are stored together; and *structure* locality, where states of neighboring vertices in the same snapshot are laid out close to each other. Due to the complex structure of a graph, structure locality is very hard to achieve. Chronos thus favors time locality for graph layout. The selected snapshots for a temporal graph in a time range are stored together in a vertex data array and an edge array. In the vertex data array, data is grouped by the vertices. The data of a vertex in consecutive snapshots are placed together. In the edge array, all the edges are grouped by the source vertices. Inside each group, every edge stores the target vertex ID and a bitmap indicating the snapshots that contain the edge.

Scheduling of Graph Computation. To leverage the time-locality graph layout, Chronos employs the locality-aware batch scheduling (LABS) of graph computation. More specifically, LABS batches the processing of a vertex across all the snapshots, as well as the information propagation to a neighboring vertex for all the snapshots. In [Han et al., 2014b], the authors show that with a simple partition-by-vertex strategy, LABS significantly improves the performance of graph computation in a multi-core parallel setting.

Incremental Computation. Since Chronos targets at time-range graph analysis, it benefits more from incremental computation. Besides the incremental approach discussed at the end of Section 11.1, Chronos proposes two enhancements. First, if the target time-range contains a sequence of N snapshots S_0 to S_{N-1}, it first computes on S_0, and then uses the final states of S_0 as the initial states for S_1 to S_{N-1}, and computes the remaining $N-1$ snapshots in one batch using LABS. In the second enhancement, Chronos pre-computes the intersection (or the union) of the N snapshots, applies graph computation on the intersection (or union) graph first, and then uses the final states of this computation as the initial states for all the snapshots and computes all the snapshots in a batch. The second enhancement allows incremental algorithms designed for edge-insertion only to work with temporal graphs with edge deletion.

On-Disk Graph Layout. Chronos also leverages the time locality to store temporal graphs on disk in a compact way. The layout is organized in snapshot groups. A snapshot group G_{t_1,t_2} contains the state of G in the time range $[t_1, t_2]$, by including a checkpoint of the snapshot of G

at t_1 followed by all the updates made till t_2. The snapshot group is physically stored as edge files and vertex files in time-locality fashion. For example, an edge file begins with an index to each vertex in the snapshot group, followed by segments of vertex data. The segment of a vertex, in turn, first contains a set of edges associated with the vertex at the start time of the snapshot group, followed by all the edge updates to the vertex. A link structure is further introduced to link edge updates related to the same vertex/edge, so that the state of a vertex/edge at a given time t can be efficiently constructed.

11.2.2 DeltaGraph

The original DeltaGraph [Khurana and Deshpande, 2013] system only supported retrieval of individual snapshots of the historical graph as of specific time instances. Here we instead discuss the extension of that work [Khurana and Deshpande, 2016] that allows retrieval of different temporal graph primitives including neighborhood versions, node histories, and graph snapshots, and that features a temporal graph analysis framework built on top of Apache Spark.

Temporal Graph Index. DeltaGraph organizes the historical graph data in a hierarchical data structure, whose lowest level corresponds to the snapshots of the network over time, and whose interior nodes correspond to graphs constructed by "combining" the lower level snapshots in some fashion; the interior nodes are typically not valid snapshots as of any specific time point. Neither the lowest-level graph snapshots nor the graphs corresponding to the interior nodes are actually stored explicitly. Instead, for each edge, a *delta*, i.e., the difference between the two graphs corresponding to its endpoints, is computed, and these deltas are explicitly stored. In addition, the graph corresponding to the root is explicitly stored. Given those, any specific snapshot can be constructed by traversing any path from the root to the node corresponding to the snapshot in the index, and by appropriately combining the information present in the deltas. Use of different "combining" functions leads to a different point in the performance-storage trade-off, with *intersection* being the most natural such function. This index structure especially shines with multi-snapshot retrieval queries which are expected to be common in temporal analysis, as it can share the computation and retrieval of deltas across the multiple snapshots. The index structure is also **extensible**, providing a user the opportunity to define additional indexes to be created and maintained in order to efficiently execute specific queries (e.g., subgraph pattern matching, reachability, etc.) over the historical graph data.

To facilitate distributed storage and parallel retrieval, the deltas themselves are partitioned horizontally by nodes and vertically by attributes of the nodes, and these partitions are stored in a key-value store (specifically, Apache Cassandra). This allows efficient retrieval of not only entire snapshots, but also of individual neighborhoods or temporal histories of individual neighborhoods.

Temporal Graph Analysis Framework. The second, somewhat orthogonal, component of this system is a Apache Spark-based analysis framework to specify temporal graph analysis tasks. This analysis framework is based on an abstraction of a *set of nodes (or subgraphs) evolving over time*. Several operations are supported on top of this abstraction, including selection, timeslicing, and temporal *map* and *reduce* operations. The library is implemented in Python and Java, is built on top of Apache Spark, and also provides integration with GraphX for executing graph algorithms supported by that system.

11.2.3 LLAMA

So far, we have only reviewed distributed temporal-graph systems. There also exist some single-machine systems that support graph analytics on temporal graphs. In this subsection, we introduce a single-machine system called LLAMA [Macko et al., 2015] for storing and analyzing evolving graphs. LLAMA aims at applications that receive a steady stream of graph updates, but need to perform various whole-graph analysis on consistent views. It is worth mentioning that there also exist some single-machine temporal-graph systems for specific types of graph queries. For example, EAGr [Mondal and Deshpande, 2014] is an in-memory system for continuously answering ego-centric aggregate queries on a temporal graph, where a query computes an aggregate in the neighborhood of a vertex over a recent time window.

LLAMA is a single machine system that stores and incrementally updates an evolving graph in multi-version representation, and it supports both in-memory and out-of-core graph analysis on graph snapshots. LLAMA provides a general-purpose programming model, though vertex-centric or edge-centric computations can be implemented on top of it.

In LLAMA, an evolving graph is modeled as a time series of graph snapshots, where each batch of incremental updates produces a new graph snapshot. The graph storage is read-optimized, while the update buffer is write-optimized.

The most important contribution of LLAMA is that, it augments the compact read-only CSR representation to support mutability and persistence. Specifically, a graph is represented by a single vertex table, and multiple edge tables, one per snapshot. The vertex table is organized as a large multi-versioned array (LAMA) that uses a software copy-on-write technique for snapshotting, and the record of each vertex v in the vertex table maintains the necessary information to track v's adjacency list from the edge tables across snapshots.

We now review the LAMA data structure for representing the vertex table. Specifically, the array of records is partitioned into equal-sized data pages, and an indirection array is constructed that contains pointers to the data pages. The indirection array fits in L3 cache. To create a new snapshot, the indirection array is copied, with those references to out-dated pages replaced by those to the newly modified pages. Thus, we do not need to copy unmodified pages across snapshots. LAMA stores 16 consecutive snapshots of the vertex table in each file, so that disk space can be easily reclaimed from deleted snapshots.

The edge table for a snapshot i is organized as a fixed-length array that stores adjacency list fragments consecutively, where each adjacency list fragment contains the edges of a vertex added in snapshot i. An adjacency list fragment of vertex v also stores a continuation record, which points to the next fragment for v, or *null* if there are no more edges. To support edge deletion, each edge table may maintain a deletion vector, which is an array that encodes in which snapshot an edge was deleted.

Properties on vertices and edges may change and should also support snapshotting. Like the vertex table, each type of property is also stored with a LAMA. Different types of properties are stored in separate LAMAs, so that a job may only load the needed property (or properties) for graph analysis.

LLAMA buffers incoming updates in a write-optimized lookup table, which stores the newly-added and deleted edges for each vertex. The buffered updates are only written into a new snapshot, and a graph analytics query only runs on the read-optimized graph storage without checking the table of buffered updates.

11.3 Streaming Graph Systems

In this section, we review some of the work on streaming graph systems.

11.3.1 Kineograph

Kineograph [Cheng et al., 2012] is a dynamic graph system designed to continuously deliver analytics results on static snapshots of a dynamic graph periodically (say every 10 seconds). The system consists of two layers: a *storage layer* that continuously applies updates to a dynamic graph and a *computation layer* that performs graph computation on a graph snapshot.

Storage Layer. In the storage layer, a dynamic graph is stored in a distributed key/value store among a set of *graph nodes*. Each record in the key/value store is a vertex with its sorted list of directed weighted edges and its associated attributes. A separate set of *ingest nodes* serve as the front end of incoming graph updates. Each update received by an ingest node is turned into a transaction consisting of a set of update operations that may span multiple graph nodes. In addition, each transaction received by an ingest node is assigned a continuously increasing sequence number s_i. Kineograph employs an *epoch commit* protocol to produce snapshots of a dynamic graph. A global *progress table* is used to keep track of the process made by each ingest node. When an ingest node i receives the acknowledgment from all relevant graph nodes that update operations for all transactions up to s_i have been received and stored, it updates its corresponding entry in the progress table to s_i. A snapshot is created by periodically taking the vector of sequence numbers, $<s^{(1)}, s^{(2)}, ..., s^{(n)}>$, from the global progress table, where $s^{(i)}$ represents the entry for ingest node i. This vector of sequence numbers serves as a logical time stamp to define the end of an epoch, and ultimately a snapshot. The epoch commit protocol guarantees consistent snapshots without blocking the ingest nodes.

Computation Layer. Once a snapshot is generated, it is passed to the computation layer for processing. Kineograph uses the vertex-based GAS computation model, and supports both *push* and *pull* models for inter-vertex communication. In the push model, a vertex sends partial updates to other vertices after changing its value, whereas in the pull model, a vertex reads the values of its neighbors before updating its own value.

11.3.2 TIDE

TIDE [Xie et al., 2015b] is a distributed system specially designed for analyzing dynamic interaction graphs in which new interactions, represented by edges, are continually added. One of the key features that sets TIDE apart from the other temporal or streaming graph systems is a novel and unique way of generating a static view of a dynamic graph, which is called the *probabilistic edge decay* (PED) model.

As discussed in Section 11.1, all other temporal or streaming graph systems use the snapshot model to generate a static view of a dynamic graph. A key drawback of the snapshot model is the ever-increasing size of the snapshots, especially for insertion-heavy graph updates. 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 new vertices or edges are added to the temporal 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 graph updates 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.

To address the drawbacks of both the snapshot and the sliding-window models, TIDE proposes a *probabilistic-edge-decay* (PED) model, which takes one or more samples of the snapshot at a given time. 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 a controlled trade-off between recency and continuity. In fact, both the snapshot model and the sliding-window model are two special cases of the PED model.

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* $f : \Re_+ \mapsto [0, 1]$. 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 $f(x) = I(x \leq w)$ respectively, where $I(X)$ denotes the indicator function of event X.

In the *PED model*, an analytic function F applied to G at time t is actually applied to N (≥ 1) independent and identically distributed (i.i.d.) sample graphs $G_t^{f,1}, G_t^{f,2}, \ldots, G_t^{f,N}$ to yield i.i.d. results $F(G_t^{f,1}), F(G_t^{f,2}), \ldots, F(G_t^{f,N})$. These results can be used to control the variability introduced by the sampling process. In the simplest cases, the results can be averaged together.

TIDE focuses on the important class of *exponential* decay functions of the form $f(x) = p^x$ for some $0 < p < 1$, where x denotes the age of an edge. Exponential decay of edges is able to capture many application scenarios and has been widely adopted in practice Roth et al. [2010], Yu et al. [2004], Zheng et al. [2011]. Moreover, using the exponential decay functions, the authors in [Xie et al., 2015b] prove that the PED model has a bounded memory requirement as new edges are added over time. This is in contrast to the continually increasing memory requirement under the snapshot model.

Maintaining Sample Graphs. As time advances from t to $t+1$, instead of naively generating the sample graph $G_{t+1}^{f,i}$ from scratch, TIDE employs an incremental approach by subsampling the edge set of $G_t^{f,i}$ using Bernoulli sampling with probability p and combining the subsample with the edges in the arriving edge batch at $(t+1)$. In addition, taking advantage of the overlap between different sample graphs at the same time point, TIDE compactly stores all the N sample graphs together as a single *aggregate graph* $\tilde{G}_t^f = (V, \bigcup_{i=1}^N E_t^{f,i})$, where the edge sets of the sample graphs are simply unioned. The attributes for an edge that appears in multiple sample graphs need only be stored once in the aggregate graph. For each aggregate edge, TIDE keeps track of the sample graph(s) to which the edge belongs.

A straightforward implementation of the incremental sample maintaining process results in an *eager* incremental updating algorithm, where a bit array of size N, denoted as β, is attached to each edge e in the aggregate graph, to indicate the sample graphs to which this edge belongs. At each batch arrival time, this algorithm scans through the bit array and, for each bit that equals 1, the algorithm sets it to 0 with probability $1-p$. Once β contains all 0s, the edge can be removed from the aggregate graph. Albeit simple and straightforward, this algorithm requires storing a bit array of size N for each edge in the aggregate graph.

The lazy incremental updating method avoids materializing the bit arrays based on the observation that the *life span* of edge e in the ith sample graph, denoted by L_e^i, 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. For an edge e that has just been added to the ith sample graph, we can 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 ith sample graph. To avoid materializing the life span of each edge in each sample graph, this algorithm exploits a 64-bit version of the MurmurHash3 random hash function[1] to efficiently and deterministically regenerate the life spans whenever needed, while maintaining their mutual statistical independence.

Bulk Analysis of Sample Graphs. When applying analytics algorithms, TIDE takes advantage of the similarities among sample graphs, and employs a *bulk* execution model on multiple sample graphs to improve efficiency. It first partitions the N sample graphs into one or more *bulk sets* comprising s ($\leq N$) sample graphs. For each bulk set, TIDE combines the s sample graphs into a *partial aggregate* graph, and processes 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 aggregate graph is an array of the states of the corresponding vertex or edge in the s sample graphs.

TIDE allows users to use exactly the same Pregel API for their graph algorithms as if the computation were applied on a single static graph. Underneath, the 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. After one bulk set is complete, TIDE proceeds to the next bulk set until all of the N sample graphs are processed.

11.4 Brief Summary of Other Work

There has been a flurry of work on temporal and stream graph analytics in recent years. In this chapter, we only covered a representative set of systems to emphasize some of the key challenges in this space. Here we briefly review some of the other work in these topics.

Temporal graph analytics is an area of growing interest. Evolution of shortest paths in dynamic graphs has been studied by Huo et al. [Huo

[1] MurmurHash: `sites.google.com/site/murmurhash`

and Tsotras, 2014], and Ren et al. [Ren et al., 2011]. Evolution of community structures in graphs has been of interest as well [Berger-Wolf and Saia, 2006, Greene et al., 2010]. Change in page rank with evolving graphs [Bahmani et al., 2010], and the study of change in centrality of vertices, path lengths of vertex pairs, etc. [Pan and Saramäki, 2011], also lie under the larger umbrella of temporal graph analysis. Ahn et al. [Ahn et al., 2014] provide a taxonomy of analytical tasks over evolving graphs. Barrat et al. [Barrat et al., 2008], provide a good reference for studying several dynamic processes modeled over graphs. Kolaczyk's book on statistical analysis of graphs [Kolaczyk, 2009], serves as a good reference for techniques and applications for graph analysis in general.

Regarding systems for storing and querying historical graph data: G* [Labouseur et al., 2014] stores multiple snapshots compactly by utilizing commonalities. ImmortalGraph [Miao et al., 2015] is an in-memory system for processing dynamic graphs, with the objectives of shared storage and computation for overlapping snapshots. Ghrab et al. [2013] provide a system for network analytics through labeling graph components. Gedik and Bordawekar [2014] describe a block-oriented and cache-enabled system to exploit spatio-temporal locality for solving temporal neighborhood queries. Koloniari and Pitoura [2013] also utilize caching to fetch selective portions of temporal graphs they refer to as partial views.

There are fewer general-purpose systems for real-time analytics over streaming graph data, but this area has seen increased interest in recent years. Several works have looked at the problem of continuous detection of *subgraph pattern matching* queries over streaming graph data. Song et al. [2014] study the problem of event pattern matching over graph streams; they consider queries that have additional timing order constraints (i.e., *happened before relationships* in events) along with the graph structure. Choudhury et al. [2015] investigate a selectivity-driven approach for continuous pattern detection on streaming graphs. Their approach is to do continuous pattern mining by decomposing the main query based on the selectivity of the node attributes, matching the individual components, and finally performing a multi-way join. In a recent work, Gao et al. [2014] propose a vertex-centric approach for continuous pattern matching for dynamic graphs using Apache Giraph. Their approach focuses on decomposing the query graph into a DAG and then using the DAG to define message transition rules for each of the nodes in the Giraph framework. The DAGs could be seen as exploration plans, to be traversed by Giraph, one edge at a time. While their approach is a nice fit for Giraph's programming model, such a framework might not be usable when there exist strict latency requirements. Their approach is more suitable for tree patterns, and may require a very large number of steps to detect structures like cliques and bicliques. Another work by Wang and Chen [2009] used an index-based technique for continuous subgraph pattern matching. For each vertex in the graph, the index, named node-neighbor tree, encodes all the simple paths of length l rooted at the vertex. Designing specification languages for continuous subgraph pattern queries has also received much attention. Two extensions to SPARQL have also been proposed in recent work for specifying continuous queries over streaming RDF data [Barbieri et al., 2010, Anicic et al., 2011].

There is also much work on streaming algorithms for specific problems like *counting triangles*, PageRank computation, sketching, etc. (e.g., *counting triangles* [Jowhari and Ghodsi, 2005, Becchetti et al., 2008], PageRank computation [Das Sarma et al., 2008], sketching [Zhao et al., 2011, Aggarwal et al., 2010], etc.), on theoretical models and approximation algorithms [Feigenbaum et al., 2004, 2005, Demetrescu et al., 2009]. Mondal et al. [Mondal and Deshpande, 2014] consider the problem of executing a large number of *ego-centric* aggregate queries over streaming graph data, and present a series of techniques for sharing computation across those. Given the increasing prevalence of graph-structured data, we expect to see much more work on real-time graph analytics in the near future.

11.5 Summary

In this chapter, we briefly surveyed the work on enabling temporal and real-time analytics over time-evolving, dynamic graphs. Although some patterns are beginning to emerge, the existing works on these topics still exhibit significant and fundamental differences, both in the high-level interfaces and in the low-level data structures and execution paradigms. For static graph analytics, the benefits of the vertex-centric programming model, and its applicability to a wide range of graph analysis tasks, were immediately evident, and this allowed the initial research to coalesce around that model. There is no such consensus yet for temporal or real-time graph analytics, although that may primarily be a result of the limited amount of work (with very different starting points) on this topic to date. As static graph analytics systems mature, we expect an increasing amount of work on extending those systems to handle temporal analytics. In terms of programming models, similar expressivity vs efficiency trade-offs are likely to emerge as with static graph analytics; in other words, it should be easier to support a vertex-centric programming model than the more expressive programming models discussed in previous chapters (in fact, several of the systems we surveyed in this chapter already support the vertex-centric programming model). Overall, this remains a rich area for future work in graph analytics.

12

Conclusions and Future Directions

In this survey, we have reviewed the landscape of big graph analytics platforms, summarizing a large number of systems that have been developed over the last decade, introduced their key ideas and features, and discussed their weaknesses. As we discussed early on, the graph analytics tasks that are of interest exhibit significant diversity in the types of computations they need to do, leading to the different systems adopting a range of different designs that we attempted to categorize in this survey.

Over the last decade, many lessons have been learnt in developing big graph systems, and insights have been developed about the various inherent tradeoffs, especially in the distributed context. For example, in a distributed graph-parallel platform, the expensive cost of vertex migration may outweigh the performance improvement resulting from a balanced computation workload (see Section 3.4); while maintaining data as raw Java objects may lead to a high memory consumption and garbage collection cost, and thus it may be important to perform serialization and to support out-of-core execution. Also, some optimization techniques for Pregel-like systems may not guarantee result exactness for all Pregel algorithms: GiraphUC only computes approximate results for PageRank computation (see Section 4.2.2), while the message online computing model of MOCgraph (see Section 3.3) and the DAIC model of Maiter (see Section 4.2.1) are only applicable to Pregel algorithms where message combining applies. As another example, while the execution model of X-Stream avoids edge sorting, each iteration needs to stream all edges (see Section 5.2.2) and it is thus not suitable for graph algorithms that run a lot of iterations but only a small portion of vertices perform computation in each iteration (e.g., BFS). Similarly, the more complex programming models that we discussed in the latter parts of the survey may make it easier or more intuitive to write some tasks, but often exhibit significant performance or environment limitations. Being aware of the pros and cons of existing big graph analytics platforms, data scientists may select the right platform for their particular graph problems, while system practitioners may apply the proper techniques in their systems and avoid ineffective approaches.

In terms of future research in this field, we identify several directions where we believe more research is required.

Firstly, it is important to understand the expressiveness of a graph-parallel computation paradigm, such as whether the model supports pointer-jumping algorithms, and whether the model supports graph mining problems. For instance, many existing works focus too much on the GAS model of PowerGraph, or Pregel algorithms where message combining is applicable. In these algorithms, messages can be directly aggregated to vertex value without being buffered, and the narrower algorithm domain naturally generates more opportunities for optimization. However, it is also important to explore the opportunity of optimizing graph algorithms that are more general, such as those previously studied in the PRAM (parallel random-access machine) model, where vertices may communicate with non-neighbors (e.g., pointer jumping) but there are still performance guarantees. Similarly, a large class of graph algorithms cannot be easily and/or efficiently handled using the vertex-centric frameworks that most works primarily use. This includes graph mining problems whose outputs can be overlapping subgraphs, as well as a range of other tasks where the scope of computation goes beyond the neighborhood of a vertex (Section 7.1). Although we have reviewed two very recent subgraph-centric systems, NScale (Section 7.2) and Arabesque (Section 7.3), these solutions are still not satisfactory. For example, they need to construct subgraphs first for later processing, and subgraph construction often involves saving (resp. loading) the partially constructed subgraphs to (resp. from) HDFS. A better solution could be to handle a portion of subgraphs each time, by pulling remote vertices without the need of saving subgraphs back to HDFS, while still fully utilizing the message bandwidth by batched message transmission. Most of the other systems that support more complex graph programming paradigms are designed for multi-core shared-memory environments (Section 9.2), and it is not clear how one may support those in a distributed setting.

Secondly, existing experimental studies on big graph systems [Lu et al., 2014, Han et al., 2014a, Satish et al., 2014, Guo et al., 2014] mainly compare the performance of in-memory vertex-centric systems. We have reviewed a lot of other systems, with different optimization techniques, programming models (e.g., matrix-centric, DSL), execution environments (e.g., single-machine or distributed, in-memory or out-of-core), and hardwares (e.g., SSD, GPU), and therefore, an interesting direction is to conduct experimental comparisons along these richer aspects, to gain more insight into the performance, strengths, and features of existing big graph systems, and to guide the choice of the right graph platforms and hardwares for specific applications and graph sizes.

Thirdly, despite the plurality of graph systems, majority of them support processing static graphs only. In reality, however, many graph applications today need to handle changes to the graphs over time. In Chapter 11, we surveyed the few emerging systems that support the analysis of temporal and streaming graphs. Still, many challenges remain in this sub-field. For example, most temporal and streaming graph systems cannot efficiently handle frequent vertex and edge deletions, as deletions often break the nice properties that enable incremental computation in these systems. Certainly, more work can be done in this area to support efficient analysis of dynamic graphs.

Fourthly, most graph analysis systems assume that the graph is already generated in the requisite format for ingesting into the system. In practice, however, usually the graphs must first be extracted from non-graph data stores using a pre-processing step that generates the lists of nodes and edges [Xirogiannopoulos et al., 2015]. In many cases, graph analysis may form one component of a deep analysis pipeline, that also involves non-graph analytics operations [Dave et al., 2016]; in such cases also, we may need to convert the data among different representations. The costs of such pre-processing or conversion steps can be significant, and in some cases, the cost of extracting graphs may dominate the actual computation that follows (e.g., if edges are generated by computing similarities between node attributes).

Lastly, with the popularity of new hardwares such as Infiniband, SSD and GPU, the architecture of existing big graph analytics platforms may need to be changed due to the different speeds of the network, storage and computing resources. Also, as the memory size is getting larger and larger, the CPU cache misses will become an increasingly more important factor. Although we have reviewed some novel systems designed to exploit the high potential of new hardware technologies in Sections 5.2.2, 10.1 and 10.2, we expect that more hardware-aware designs of big graph systems will emerge in the future.

References

Charu C. Aggarwal, Yao Li, Philip S. Yu, and Ruoming Jin. On dense pattern mining in graph streams. *VLDB*, 2010.

Jae-wook Ahn, Catherine Plaisant, and Ben Shneiderman. A task taxonomy for network evolution analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, 2011.

Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.

Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *VLDB*, 2010.

Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.

François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *VLDB*, pages 1–15, 1986.

Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT*, 2010.

Alain Barrat, Marc Barthelemy, and Alessandro Vespignani. *Dynamical processes on complex networks*. Cambridge University Press Cambridge, 2008.

Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, pages 16–24, 2008.

Catriel Beeri, Shamim A. Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language (LDL1). In *PODS*, pages 21–37, 1987.

Tanya Y Berger-Wolf and Jared Saia. A framework for analysis of dynamic social networks. In *SIGKDD*, 2006.

Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7):553–564, 2014.

Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425 – 1436, 2016.

Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World-Wide Web Conference (WWW)*, pages 107–117, 1998.

Yingyi Bu. *On Software Infrastructure for Scalable Graph Analytics*. PhD thesis, Computer Science Department, University of California, Irvine, August 2015.

Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C. S. Lui, and Cheng He. VENUS: vertex-centric streamlined graph computation on a single PC. In *ICDE*, pages 1131–1142, 2015.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Josef Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL*, pages 63–78, 2015.

Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *EDBT*, 2015.

Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. In *PODS*, 2008.

Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. GraphFrames: An integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 2:1–2:8, 2016.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Trans. Algorithms*, 2009.

David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending datalog for modern AI. In *Datalog*, pages 181–220, 2010.

Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.

E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. ISSN 0360-0300.

David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Dynamic Graph Algorithms*. CRC Press, 1999.

Shimon Even. *Graph Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.

Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *ICALP*, 2004.

Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: the value of space. In *SODA*, 2005.

Zhisong Fu, Bryan B. Thompson, and Michael Personick. Mapgraph: A high level API for fast development of high performance graph analytics on gpus. In *GRADES*, pages 2:1–2:6, 2014.

Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, pages 556–567, 2014.

B. Gedik and R. Bordawekar. Disk-based management of interaction graphs. *TKDE*, 2014.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

A. Ghrab, S. Skhiri, S. Jouili, and E. Zimányi. An analytics-aware conceptual model for evolving graphs. In *Data Warehousing and Knowledge Discovery*. Springer, 2013.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

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*, pages 599–613, 2014.

Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *ASONAM*, 2010.

Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *IPDPS*, pages 395–404, 2014.

Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. WTF: the who to follow service at twitter. In *WWW*, pages 505–514, 2013.

Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of Pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014a.

Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, pages 1:1–1:14, 2014b.

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, 2012.

Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *CGO*, page 208, 2014.

Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.

W. Huo and V. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, 2014.

Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. epic: an extensible and scalable system for processing big data. *PVLDB*, 7(7): 541–552, 2014.

Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph analytics using the Vertica relational database. *CoRR*, abs/1412.5263, 2014a.

Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. VERTEXICA: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014b.

Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In Lusheng Wang, editor, *Computing and Combinatorics*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005.

U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: a scalable and general graph management system. In *SIGKDD*, pages 1091–1099, 2011.

George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC*, pages 239–252, 2014.

Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.

Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, pages 65–76, 2016.

Eric D Kolaczyk. *Statistical analysis of network data*. Springer, 2009.

G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *GRADES workshop*, 2013.

M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

A. Labouseur, J. Birnbaum, Jr. Olsen, P., S. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 2014.

Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

Jure Leskovec and Julian J. Mcauley. Learning to discover social circles in ego networks. In *NIPS*, pages 548–556, 2012.

Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *MLG*, pages 78–85. ACM, 2010. ISBN 978-1-4503-0214-2.

Guimei Liu and Limsoon Wong. Effective pruning techniques for mining quasi-cliques. In *ECML/PKDD Part II*, pages 33–49, 2008.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

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):716–727, 2012.

Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *ICDE*, pages 363–374, 2015.

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 Conference*, pages 135–146, 2010.

Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *VLDB J.*, 22(4):471–493, 2013.

Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25, 2015.

Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost. In *HotOS*. USENIX Association, 2015.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM TOS*, July 2015. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=242176`.

Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, pages 145–156, 2012.

Jayanta Mondal and Amol Deshpande. Eagr: supporting continuous egocentric aggregate queries over large dynamic graphs. In *SIGMOD*, pages 1335–1346, 2014.

Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 2011.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

Abdul Quamar and Amol Deshpande. NScaleSpark: Subgraph-centric graph analytics on Apache Spark. In *Proceedings of the SIGMOD Workshop on Network Data Analytics (NDA)*, pages 5:1–5:8, 2016.

Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB Journal*, 25(2): 125–150, 2016.

Louise Quick, Paul Wilkinson, and David Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.

Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126, 1992.

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*, pages 233–242, 2010.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*, pages 979–990, 2014.

Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "All roads lead to rome": optimistic recovery for distributed iterative data processing. In *CIKM*, pages 1919–1928, 2013.

Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013a.

Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6 (14):1906–1917, 2013b.

Zechao Shang and Jeffrey Xu Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, pages 553–564, 2013.

Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.

Yingxia Shao, Bin Cui, and Lin Ma. PAGE: A partition aware engine for parallel graph computation. *IEEE Trans. Knowl. Data Eng.*, 27(2):518–530, 2015.

Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. Fast failure recovery in distributed graph processing systems. *PVLDB*, 8(4):437–448, 2014.

Yossi Shiloach and Uzi Vishkin. An O(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878, 2015.

Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, pages 135–146, 2013.

David E. Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *PVLDB*, 7(13):1405–1416, 2014.

Yogesh Simmhan, Alok Gautam Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*, pages 451–462, 2014.

Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *VLDB*, 2014.

Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, pages 1222–1230, 2012.

Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

Aubrey Tatarowicz, Carlo Curino, Evan P. C. Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, pages 102–113, 2012.

Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

Yuanyuan Tian, Shirish Tatikonda, and Berthold Reinwald. Scalable and numerically stable descriptive statistics in systemml. In *ICDE*, pages 1351–1359, 2012.

Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From ”think like a vertex” to “think like a graph”. *PVLDB*, 7(3):193–204, 2013.

Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, pages 5:1–5:16, 2013.

Changliang Wang and Lei Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *DSN*, pages 562–573, 2014.

Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.

Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *PPoPP*, pages 194–204, 2015a.

Wenlei Xie, Guozhang Wang, David Bindel, Alan J. Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6 (14):2014–2025, 2013.

Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *ICDE*, pages 1143–1154, 2015b.

Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen: Exploring interesting graphs in relational data. *PVLDB*, 8(12), 2015.

Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14): 1981–1992, 2014a.

Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014b.

Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. Big graph analytics systems. In *SIGMOD*, pages 2241–2243, 2016a.

Da Yan, James Cheng, M. Tamer Özsu, Fan Yang, Yi Lu, John C. S. Lui, Qizhen Zhang, and Wilfred Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, 2016b.

Da Yan, James Cheng, and Fan Yang. Lightweight fault tolerance in large-scale distributed graph processing. *CoRR*, abs/1601.06496, 2016c.

Da Yan, Yuzhen Huang, James Cheng, and Huanhuan Wu. Efficient processing of very large graphs in a small cluster. *CoRR*, abs/1601.05590, 2016d.

Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Parallel bottom-up evaluation of logic programs: Deals on shared-memory multicore machines. In *ICLP*, 2015.

Philip S. Yu, Xin Li, and Bing Liu. On the temporal dimension of search. In *WWW Alt*, pages 448–449, 2004.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

Honglei Zhang, Jenni Raitoharju, Serkan Kiranyaz, and Moncef Gabbouj. Limited random walk algorithm for big graph data clustering. *CoRR*, abs/1606.06450, 2016.

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.*, 25(8):2091–2100, 2014.

Peixiang Zhao, Charu C. Aggarwal, and Min Wang. gSketch: on query estimation in graph streams. *VLDB*, 2011.

Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

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*, pages 283–291, 2011.

Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, 2014.

Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. Mocgraph: Scalable distributed graph processing using message online computing. *PVLDB*, 8 (4):377–388, 2014.

Yang Zhou, Ling Liu, Kisung Lee, and Qi Zhang. Graphtwist: Fast iterative graph computation with two-tier optimizations. *PVLDB*, 8(11):1262–1273, 2015.

Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.

Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, 2016.