Chapter 5
Parallel Processing of Graphs

Bin Shao and Yatao Li

Abstract Graphs play an indispensable role in a wide range of application domains. Graph processing at scale, however, is facing challenges at all levels, ranging from system architectures to programming models. In this chapter, we review the challenges of parallel processing of large graphs, representative graph processing systems, general principles of designing large graph processing systems, and various graph computation paradigms. Graph processing covers a wide range of topics and graphs can be represented in different forms. Different graph representations lead to different computation paradigms and system architectures. From the perspective of graph representation, this chapter also briefly introduces a few alternative forms of graph representation besides adjacency list.

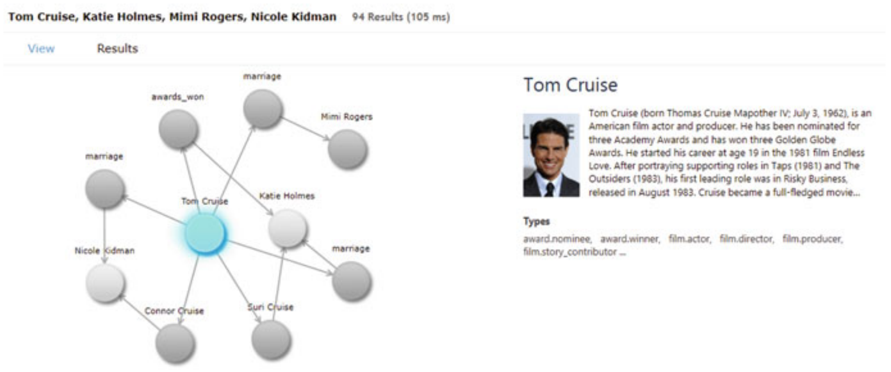
5.1 Overview

Graphs are important to many applications. However, large-scale graph processing is facing challenges at all levels, ranging from system architectures to programming models. There are a large variety of graph applications. We can roughly classify the graph applications into two categories: online query processing, which is usually optimized for low latency; and offline graph analytics, which is usually optimized for high throughput. For instance, deciding instantly whether there is a path between two given people in a social network belongs to the first category, while calculating PageRank for a web graph belongs to the second.

Let us start with a real-life knowledge graph query example. Figure 5.1 gives a real-life relation search example on a big knowledge graph. In a knowledge graph, queries that find the paths linking a set of given graph nodes usually give the relations between these entities. In this example, we find the relations between entities *Tom Cruise*, *Katie Holmes*, *Mimi Rogers*, and *Nicole Kidman*.

B. Shao (✉) · Y. Li
Microsoft Research Asia, Beijing, China
e-mail: binshao@microsoft.com; yatli@microsoft.com

G. Fletcher et al. (eds.), *Graph Data Management*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-319-96193-4_5



Tom Cruise, Katie Holmes, Mimi Rogers, Nicole Kidman 94 Results (105 ms)

View Results

Tom Cruise

Tom Cruise (born Thomas Cruise Mapother IV; July 3, 1962), is an American film actor and producer. He has been nominated for three Academy Awards and has won three Golden Globe Awards. He started his career at age 19 in the 1981 film Endless Love. After portraying supporting roles in Taps (1981) and The Outsiders (1983), his first leading role was in Risky Business, released in August 1983. Cruise became a full-fledged movie...

Types

award.nominee, award.winner, film.actor, film.director, film.producer, film.story_contributor ...

Fig. 5.1 Relation search on a knowledge graph

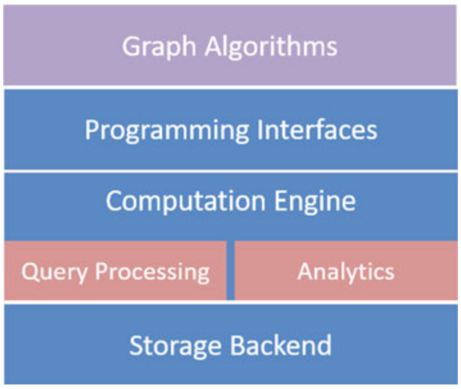


Fig. 5.2 A general graph processing system stack

Many sophisticated real-world applications highly rely on the interplay between offline graph analytics and online query processing. Given two nodes of a graph, the “distance oracle” algorithm designed by Qi et al. (2014) estimates the shortest distance between two given nodes; it is an online algorithm. However, to estimate the distances, the algorithm relies on “landmark” nodes in the graph and an optimal set of landmark nodes are discovered using an offline analytics algorithm.

Generally speaking, the system stack of a graph processing system consists of all or some of the layers shown in Fig. 5.2. At the top, graph algorithms manipulate graphs via programming interfaces provided by a graph processing system. Between the programming interfaces and the storage backend, there usually is a computation engine that executes the graph manipulation instructions dictated by the graph algorithms through programming interfaces.

At the bottom, the storage backend hosts the graph data using a certain graph representation, either in a single machine or over multiple distributed machines. Storage backends have important system design implications. The storage layer largely determines the system optimization goal, as discussed in Kaoudi and Manolescu (2015). For example, systems including SHARD by Rohloff and Schantz (2011), HadoopRDF by Husain et al. (2011), RAPID by Ravindra et al. (2011), and EAGRE by Zhang et al. (2013) use a distributed file system as their storage backends. The systems that directly use a file system as storage backend are usually optimized for throughput due to the relatively high data retrieval latency. In contrast, systems such as H2RDF by Papailiou et al. (2012), AMADA by Aranda-Andújar et al. (2012), and Trinity.RDF by Zeng et al. (2013) are optimized for better response time via the fast random data access capability provided by their key-value store backends.

In this section, we first introduce the notation and discuss why it is difficult to process large graphs. Then, we present some general principles of designing large-scale graph processing systems after a brief survey of some representative graph processing systems.

5.1.1 Notation

Let us introduce the terminology and notation that will be used throughout this chapter. A graph may refer to a topology-only mathematical concept as defined in Bollobás (1998) or a data set. In the former sense, a graph is a pair of finite sets (V, E) such that the set of edges E is a subset of the set $V \times V$. If each pair of vertices are ordered, we call G a directed graph; otherwise, we call it an undirected graph.

In what follows when we represent a data set as a graph, especially when there are data associated with the vertices, we refer to the vertices as *graph nodes* or *nodes*. Correspondingly, we call adjacent vertices *neighboring nodes*. If the data set only contains graph topology or if we only want to emphasize its graph topology, we call them vertices.

There are two common ways of representing and storing a graph: adjacency list and adjacency matrix. The way of representing a graph determines the way we can process the graph. As most graph query processing and analytics algorithms highly rely on the operator that gets adjacent vertices of a given vertex, adjacency list is usually a preferred way of representing a graph especially when the graph is large. If we use the adjacency matrix representation, we need to scan a whole adjacency matrix row to access the adjacent vertices of given vertex. For a graph with billions of vertices, the costs of scanning matrix rows will be prohibitive. In this chapter, we assume graphs are represented and stored as adjacency lists unless otherwise stated.

5.1.2 Challenges of Large Graph Processing

It is difficult to process large graphs mostly because they have a large number of encoded relations. We summarize the challenges of large graph processing as:

(1) the complex nature of graph; (2) the diversity of graphs; (3) the diversity of graph computations; (4) the scale of graph size.

5.1.2.1 The Complex Nature of Graph

Graphs are inherently complex. The contemporary computer architectures are good at processing linear and simple hierarchical data structures, such as *Lists*, *Stacks*, or *Trees*. When the data scale goes large, the *divide and conquer* computation paradigm still works well for these data structures, even if the data is partitioned over distributed machines.

However, when we are handling graphs, especially large graphs, the situation is changed. Andrew Lumsadine and Douglas Gregor (2007) summarize the characteristics of parallel graph processing as *data-driven computations*, *unstructured problems*, *poor locality*, and *high data access to computation ratio*. The implication is twofold: (1) From the perspective of data access, a graph node's neighboring nodes cannot be accessed without "jumping" in the storage no matter how we represent a graph. In other words, a large amount of random data accesses are required during graph processing. Many modern program optimization techniques rely on data locality and data reuse. Unfortunately, the random data access nature of graph breaks the premise. This usually causes poor system performance as the CPU cache is not effective for most of the time. (2) From the perspective of program structure, it is difficult to extract parallelism because of the unstructured nature of graphs. Partitioning large graphs itself is an NP-hard problem as shown by Garey et al. (1974); this makes it hard to get an efficient *divide and conquer* solution for many large graph processing tasks.

5.1.2.2 The Diversity of Graphs

There are many kinds of graphs, such as scale-free graphs, graphs with community structures, and small-world graphs. A scale-free graph is a graph whose degrees follow a power-law distribution. For graphs with community structure, the graph nodes can easily be grouped into sets of nodes such that each set of nodes are densely connected. For small-world graphs, most nodes can be reached from other nodes by a small number of hops. The performance of graph algorithms may vary a lot on different kinds of graphs.

5.1.2.3 The Diversity of Graph Computations

Furthermore, there are a large variety of graph computations. As discussed earlier, graph computations can be roughly classified into two categories: online query processing and offline graph analytics. Besides common graph query processing and analytics tasks, there are other useful graph operations such as graph generation, graph visualization, and interactive exploration. It is challenging to design a system that can support all these operations on top of the same infrastructure.

5.1.2.4 The Scale of Graph Size

Last but not least, the scale of graph size does matter. Graphs with billions of nodes are common now, for example, the Facebook social network has more than two billion monthly active users.[1] The World Wide Web has more than one trillion unique links. The De brujin graph for genes even has more than one trillion nodes and at least eight trillion edges. The scale of graph size makes many classic graph algorithms from textbooks ineffective.

5.1.3 Representative Graph Processing Systems

Recent years have witnessed an explosive growth of graph processing systems as shown by Aggarwal and Wang (2010). However, many graph algorithms are ad hoc in the sense that each of them assumes that the underlying graph data is organized in a certain way to maximize its performance. In other words, there is not a standard or de facto graph system on which graph algorithms are developed and optimized. In response to this situation, a number of graph systems have been proposed. Some representative systems are summarized in Table 5.1.

Neo4j[2] focuses on supporting online transaction processing (OLTP) of graphs. Neo4j is like a regular database system, with a more expressive and powerful data model. Its computation model does not handle graphs that are partitioned over multiple machines. For large graphs that cannot be stored in the main memory, disk random access becomes the performance bottleneck.

From the perspective of online graph query processing, a few distributed in-memory systems have been designed to meet the challenges faced by disk-based single-machine systems. Representative systems include Trinity by Shao et al. (2013) and Horton by Sarwat et al. (2013). These systems leverage RAM to speed up random data accesses and use a distributed computation engine to process graph queries in parallel.

On the other end of the spectrum are MapReduce by Dean and Ghemawat (2008), PEGASUS by Kang et al. (2009), Pregel by Malewicz et al. (2010), Giraph, GraphLab by Low et al. (2012), GraphChi by Kyrola et al. (2012), and GraphX by Gonzalez et al. (2014). These systems are usually not optimized for online query processing. Instead, they are optimized for high-throughput analytics on large graphs partitioned over many distributed machines.

[1] http://newsroom.fb.com/company-info/.

[2] http://neo4j.com/.

Table 5.1 Some representative graph processing systems (*SB* means the feature depends on its storage backend)

	Native graphs	Online query	Data sharding	In-memory storage	Transaction support
Neo4j	Yes	Yes	No	No	Yes
Trinity	Yes	Yes	Yes	Yes	Atomicity
Horton	Yes	Yes	Yes	Yes	No
FlockDB[a]	No	Yes	Yes	No	Yes
TinkerGraph[b]	Yes	Yes	No	Yes	No
InfiniteGraph[c]	Yes	Yes	Yes	No	Yes
Cayley[d]	Yes	Yes	SB	SB	Yes
Titan[e]	Yes	Yes	SB	SB	Yes
MapReduce	No	No	Yes	No	No
PEGASUS	No	No	Yes	No	No
Pregel	No	No	Yes	No	No
Giraph[f]	No	No	Yes	No	No
GraphLab	No	No	Yes	No	No
GraphChi	No	No	No	No	No
GraphX	No	No	Yes	No	No

[a]https://github.com/twitter/flockdb
[b]https://github.com/tinkerpop/blueprints/wiki/TinkerGraph
[c]http://www.objectivity.com/products/infinitegraph/
[d]https://github.com/google/cayley
[e]http://thinkaurelius.github.io/titan/
[f]http://giraph.apache.org/

MapReduce-based graph processing depends heavily on inter-processor bandwidth as graph structures are sent over the network iteration after iteration. Pregel and its follow-up systems mitigate this problem by passing computation results instead of graph structures between processors. In Pregel, analytics are expressed using a vertex-centric computation paradigm. Although some well-known graph algorithms such as PageRank and shortest path discovery can be expressed using vertex-centric computation paradigm easily; there are many sophisticated graph computations that cannot be expressed in a succinct and elegant way.

The systems listed in Table 5.1 are compared from five aspects. First, does the graph exist in its native form? When a graph is in its native form, graph algorithms can be expressed in standard, natural ways as discussed by Cohen (2009). Second, does the system support low-latency query processing? Third, does the system support data sharding and distributed parallel graph processing? Fourth, does the system use RAM as the main storage? Fifth, does the system support transactions?

5.2 General Design Principles

We have reviewed a few representative graph processing systems. In this section, we discuss a few general principles of designing a general-purpose, real-time graph processing system.

5.2.1 Addressing the Grand Random Data Access Challenge

As discussed earlier, a graph node's neighboring nodes' cannot be accessed without "jumping" in the storage no matter how we represent a graph. A lot of random accesses on hard disks lead to performance bottlenecks. It is important to keep graphs memory-resident for efficient graph computations, especially for real-time online query processing. In order to create a general-purpose graph processing system that supports both low-latency online query processing and high-throughput offline analytics, the grand challenge of random accesses must be well addressed at the data access layer.

Despite the great progress made in disk technology, it still cannot provide the level of efficient random access required for graph processing. DRAM (dynamic random-access memory) is still the only promising storage medium that can provide a satisfactory level of random access performance with acceptable costs. On the other hand, in-memory approaches usually cannot scale to very large graphs due to the capacity limit of a single machine. We argue that distributed RAM storage is a promising approach to efficient large graph processing.

By addressing the random data access challenge, we can design systems that efficiently support both online graph query processing and offline graph analytics instead of optimizing the systems for certain graph computations. For online queries, it is particularly effective to keep graphs in-memory, as most online query processing algorithms, such as BFS, DFS, and subgraph matching, require a certain degree of graph exploration. On the other hand, offline graph computations are usually conducted in an iterative, batch manner. For iterative computations, keeping data in RAM can boost the performance by an order of magnitude due to the reuse of intermediate results as discussed by Zaharia et al. (2010).

5.2.2 Avoiding Prohibitive Indexes

For offline graph analytics, partitioning the computation task well (if possible) is, to some extent, the silver bullet. As long as we can find an efficient way to partition the graph data, we basically have a solution to efficient offline graph processing. Due to the random data access challenge, general-purpose, efficient disk-based solutions usually do not exist. But under certain constraints, offline graph analytics tasks can have efficient disk-based "divide and conquer" solutions. A good example is the GraphChi system proposed by Kyrola et al. (2012). If a computational problem can be well-partitioned, then the subproblems can be loaded in the main memory and processed in-memory one at a time. However, as discussed by Lumsdaine et al. (2007), many graph problems are inherently irregular and computation partitioning is hard, especially for online queries.

Compared with offline analytics, online queries are much harder to handle due to the following two reasons. First, online queries are sensitive to network latency. It is harder to reduce communication latency than to increase throughput by adding more machines. On the one hand, adding more machines can reduce each machine's workload; on the other hand, having more machines incurs higher communication costs. Second, it is generally difficult to predict the data access patterns of a graph query, thus it is hard to optimize the execution by leveraging I/O optimization techniques such as prefetching.

Many graph computations are I/O intensive; data accesses usually dominate the graph computation costs. The performance of processing a graph query depends on how fast we randomly access the graph. A traditional way of speeding up random data access is to use indexes. Graph indexes are widely employed to speed up online query processing, either by precomputing and materializing the results of common query patterns or by storing redundant information. To capture the structural information of graphs, graph indexes usually require super-linear indexing time and super-linear storage space. For large graphs, for example, graphs with billions of nodes, the super-linear complexity means infeasibility. We will show in the following section that index-free graph processing is a possible and efficient approach to many real-time graph query processing tasks.

5.2.3 Supporting Fine-Grained One-Sided Communications

Most graph computations are data-driven and the communication costs typically contribute a large portion to the overall system costs. Overlapping computations well with the underlying communication is the key to high performance.

MPI is the de facto standard for message passing programming in high-performance computing and pairwise two-sided send/receive communication is the major paradigm provided by MPI.[3] The communication progress is dictated by explicitly invoked MPI primitive calls as shown by Majumder and Rixner (2004). Nearly all modern network communication infrastructures provide asynchronous network events. MPI communication paradigm incurs unnecessary latency because it responds to network events only during *send* and *receive* primitive invocations.

In contrast, active messages introduced by von Eicken et al. (1992) is a communication architecture that can well overlap computation and communication.

[3]Even one-sided primitives are included starting from MPI-2 standard, their usage is still limited.

The communication architecture is desirable for data-driven graph computation, especially for online graph query processing, which is sensitive to network latency.

When we use active messages, a user-space message handler will be invoked upon the arrival of a message. The message handler is pointed by a handler index encoded in the message. Let us use a simple example to illustrate the difference between the two-sided communication paradigm and active messages. Suppose we want to send some messages from one machine to another according to the output values of a random number generator. Using active messages, we can check the random values on the sender side and invoke a *send* operation only if the value matches certain sending condition. Using the pairwise two-sided communication paradigm, we need to invoke as many send/receive calls as the number of generated random values and perform the value checkings on the receiver side.

5.3 Online Query Processing

In this section, we review two online query processing techniques specially designed for distributed large graphs: asynchronous fanout search and index-free query processing.

5.3.1 Asynchronous Fanout Search

Most graph algorithms require a certain kind of graph exploration; breadth-first search (BFS) and depth-first search (DFS) are among the commonest. Here, we use people search in a social network as an example to demonstrate an efficient graph exploration technique called *asynchronous fanout search*. The problem is the following: given a user of a social network, find the people whose first name is "David" among his friends, his friends' friends, and his friends' friends' friends.

It is unlikely that we can index the social network to solve the "David" problem. One option is to index the neighboring nodes for each user, so that given any user, we can use the index to check if there is "David" within his or her 3-hop neighborhood.

Algorithm 1: Asynchronous Fanout search

```
Require: $v$ (a given graph node)
Ensure: all "Davids" within the 3-hop neighboring nodes
1: $N \leftarrow$ the ids of $v$'s neighboring nodes
2: $k \leftarrow$ (the number of machines)
3: $hop \leftarrow 1$
4: Partition $N$ into $k$ parts: $N = N_1 \cup \cdots \cup N_k$
5: parallel-foreach $N_i$ in $N$
6:     async_send message $(N_i, hop)$ to machine $i$
```

Algorithm 2: On receiving message (N_i, hop)

```
1: $S_i \leftarrow$ the graph nodes with ids in $N_i$
2: check if there are "Davids" in $S_i$
3: if $hop < 3$ then
4:     $N' \leftarrow$ ids of the neighboring nodes of the nodes in $S_i$
5:     Partition $N'$ into $k$ parts: $N' = N'_1 \cup \cdots \cup N'_k$
6:     parallel-foreach $N'_i$ in $N'$
7:         async_send message $(N'_i, hop + 1)$ to machine $i$
```

However, the size and the update cost of such an index are prohibitive for a large graph. The second option is to create an index to answer 3-hop reachability queries for any two nodes. This is infeasible either because "David" is a popular name and we cannot check every "David" in the social network to see whether he is within 3 hops to the current user.

We can tackle the "David" problem by leveraging fast memory-based graph explorations. The algorithm simply sends asynchronous "fan-out search" requests recursively to remote machines as shown by Algorithms 1 and 2. Specifically, it partitions v's neighboring nodes into k parts $N_1, N_2, \ldots, N_k$ (line 4 of Algorithm 1), where k is the total number of machines. Then, the "fan-out" search is performed by sending message (N_i, hop) (line 6 of Algorithm 1) to all machines in parallel. On receiving the search requests, machine i searches for "David" in its local data storage (line 2 of Algorithm 2) and sends out the next-hop "fan-out" search requests $(N'_i, hop + 1)$ (line 7 of Algorithm 2).

This simple fanout search works well for randomly partitioned large distributed graphs. As demonstrated by Shao et al. (2013), for a Facebook-like graph, exploring the entire 3-hop neighborhood of any given node in the graph takes less than 100 ms on average.

5.3.2 Index-Free Query Processing

It is usually harder to optimize online query processing because of its limited response time budget. We use the subgraph matching problem as an example to introduce an efficient online query processing paradigm for distributed large graphs.

Subgraph matching is the basic graph operation underpinning many graph applications. Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subset V$ and $E' \subset E$. Graph $G'' = (V'', E'')$ is isomorphic to $G' = (V', E')$ if there is a bijection $f : V' \to V''$ such that $xy \in E'$ iff $f(x)f(y) \in E''$. For a given data graph G and a query graph G', subgraph matching is to retrieve all the subgraphs of G that are isomorphic to the query graph.

Canonical subgraph matching algorithms are usually conducted in the following three steps:

1. Break the data graph into basic units such as edges, paths, or frequent subgraphs.
2. Build indexes for the basic units.
3. Decompose a query into multiple basic unit queries, do subgraph matching for the unit queries, and join their results.

It is much more costly to index graph structure than to index a relational table. For instance, 2-hop reachability indexes usually require $O(n^4)$ construction time. Depending on the structure of the basic unit, the space costs vary. In many cases, they are super-linear. Furthermore, multiway joins are costly too, especially when the data is disk-resident.

To demonstrate the infeasibility of index-based solutions for large graphs, let us review a survey on subgraph matching made by Sun et al. (2012), as shown in Tables 5.2 and 5.3.

Table 5.2 shows the index costs of a few representative subgraph matching algorithms proposed by Ullmann (1976), Cordella et al. (2004), Neumann and Weikum (2010), Atre et al. (2010), Holder et al. (1994), Zhu et al. (2011), Cheng et al. (2008), Zou et al. (2009), He and Singh (2008), Zhao and Han (2010), Zhang et al. (2009), and Sun et al. (2012). To illustrate what the costs listed in Table 5.2 imply for a large graph, Table 5.3 shows their estimated index construction costs and query time for a Facebook-like social network. Although RDF-3X and BitMat have linear indexing complexity, they take more than 20 days to index a Facebook-like large graph, let alone those super-linear indexes. The evident conclusion is that the costly graph indexes are infeasible for large graphs.

To avoid building sophisticated indexes, the STwig method proposed by Sun et al. (2012) and the Trinity.RDF system proposed by Zeng et al. (2013) process subgraph matching queries without using structural graph indexes. This ensures scalability;

Table 5.2 Survey on subgraph matching algorithms by Sun et al. (2012)

Algorithms	Index size	Index time	Update cost	Graph size in experiments
Ullmann, VF2	–	–	–	4484
RDF-3X	O(m)	O(m)	O(d)	33M
BitMat	O(m)	O(m)	O(m)	361M
Subdue	–	Exponential	O(m)	10K
SpiderMine	–	Exponential	O(m)	40K
R-join	O($nm^{1/2}$)	O(n^4)	O(n)	1M
Distance-join	O($nm^{1/2}$)	O(n^4)	O(n)	387K
GraphQL	$O(m + nd^r)$	$O(m + nd^r)$	O(d^r)	320K
Zhao	O(nd^r)	O(nd^r)	O(d^L)	2M
GADDI	$O(nd^L)$	$O(nd^L)$	O(d^L)	10K
STwig	$O(n)$	$O(n)$	$O(1)$	1B

Table 5.3 Index costs and query time of subgraph matching algorithms for a Facebook-like graph

Algorithms	Index size	Index time	Query time
Ullmann, VF2	–	–	>1000
RDF-3X	1T	>20 days	>48
BitMat	2.4T	>20 days	>269
Subdue	–	>67 years	–
SpiderMine	–	>3 years	–
R-join	>175T	$>10^{15}$ years	>200
Distance-join	>175T	$>10^{15}$ years	>4000
GraphQL	>13T(r=2)	>600 years	>2000
Zhao	>12T(r=2)	>600 years	>600
GADDI	$>2 \times 10^5$T (L=4)	$>4 \times 10^5$ years	>400
STwig	6G	33 s	<20

they work well for graphs with billions of nodes, which are not *indexable* in terms of both index space and index time.

To compensate for the performance loss due to the lack of structural indexes, both STwig and Trinity.RDF heavily make use of in-memory graph explorations to replace costly join operations. Given a query, they split it into a set of subqueries that can be efficiently processed via in-memory graph exploration. They only perform join operations when they are absolutely necessary and not avoidable, for example, when there is a cycle in the query graph. This dramatically reduces query processing time, which is usually dominated by join operations.

5.4 Offline Analytics

Graph analytics jobs perform a global computation against a graph. Many of them are conducted in an iterative manner. When the graph is large, the analytics jobs are usually conducted as offline tasks.

In this section, we review the MapReduce computation paradigm and vertex-centric computation paradigm for offline graph analytics. Then, we discuss communication optimization and a lightweight analytics technique called local sampling.

5.4.1 MapReduce Computation Paradigm

MapReduce, as elaborated by Dean and Ghemawat (2008), is a high-latency yet high-throughput data processing platform that is optimized for offline analytics for large partitioned data sets. MapReduce is a very successful programming model for big data processing. However, when used for processing graphs, it suffers from the following problems: First, it is very hard to support real-time online queries. Second, the data model of MapReduce cannot model graphs natively and graph algorithms cannot be expressed intuitively. Third, MapReduce highly relies on data partitioning; however, it is inherently hard to partition graphs.

It is possible to run a graph processing job efficiently on a MapReduce platform if the graph could be well-partitioned. Some well-designed graph algorithms implemented in MapReduce are given by Qin et al. (2014). The computation parallelism a MapReduce system can achieve depends on how well the data can be partitioned. Unfortunately, the partitioning task of a large graph can be very costly, as elaborated by Wang et al. (2014).

5.4.2 Vertex-Centric Computation Paradigm

The vertex-centric graph computation paradigm, which was first advocated by Malewicz et al. (2010), provides a vertex-centric computational abstraction over the BSP model proposed by Valiant (1990). A computation task is expressed in multiple iterative super-steps and each vertex acts as an independent agent. During each super-step, each agent performs some computations, independent of each other. It then waits for all other agents to finish their computations before the next super-step begins.

Compared with MapReduce, Pregel exploits finer-grained parallelism at the vertex level. Moreover, Pregel does not move graph partitions over the network; only messages among graph vertices are passed at the end of each iteration. This greatly reduces the network traffic.

Many follow-up works, such as GraphLab by Low et al. (2012), PowerGraph by Gonzalez et al. (2012), Trinity by Shao et al. (2013), and GraphChi by Kyrola et al. (2012), support the vertex-centric computation paradigm for offline graph analytics. Among these systems, GraphChi is specially worth mentioning as it well addresses the "divide and conquer" problem for graph computation under certain constraints. GraphChi can perform efficient disk-based graph computation as long as the computation could be expressed as an asynchronous vertex-centric algorithm. An asynchronous algorithm is one where a vertex can perform a computational task based solely on the partially updated information from its incoming links. This assumption, on the one hand, frees the need of passing messages from the current vertex to all its outgoing links so that it can perform the graph computations block by block. On the other hand, it inherently cannot efficiently support traversal-based graph computations and synchronous graph computations because it cannot freely access the outgoing links of a vertex.

Although quite a few graph computation tasks, including Single Source Shortest Paths, PageRank, and their variants, can be expressed elegantly using the vertex-centric computation paradigm, there are many that cannot be elegantly and intuitively expressed using the vertex-centric paradigm, for example, multilevel graph partitioning.

5.4.3 Communication Optimization

Communication optimization is very important for distributed graph computation. Although a graph is distributed over multiple machines, from the point view of a local machine, vertices of the graph are in two categories: vertices on the local machine, and vertices on any of the remote machines. Figure 5.3 shows a local machine's bipartite view of the entire graph.

Let us take the vertex-centric computation as an example. One naive approach is to run jobs on local vertices without preparing any messages in advance. When a local vertex is scheduled to run a job, we obtain remote messages for the vertex and run the job immediately after they arrive. Since the system usually does not have space to hold all messages, we discard messages after they are used. For instance, in Fig. 5.3, in order to run the job on vertex x, we need messages from vertices u, v, and others. Later on, when y is scheduled to run, we need messages from u and v again. This means a single message needs to be delivered multiple times, which is unacceptable in an environment where network capacity is an extremely valuable resource.

Some graph processing systems, such as the ones built using Parallel Boost Graph Library (PBGL), use *ghost nodes* (local replicas of remote nodes) for message passing as elaborated by Gregor and Lumsdaine (2005). This mechanism works well for well-partitioned graphs. However, it is difficult to create partitions of even size while minimizing the number of edge cuts. A great memory overhead would be incurred for a large graph if it is not well-partitioned. To illustrate the

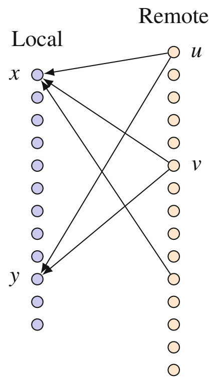


Fig. 5.3 Bipartite view on a local machine

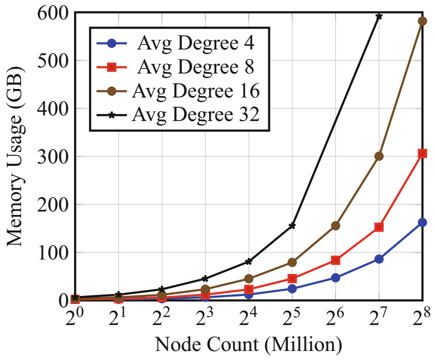


Fig. 5.4 Breadth-first search using PBGL

memory overhead, Fig. 5.4 shows the memory usage for graphs with 1 million to 256 million vertices. It takes nearly 600 GB main memory for the 256-million-node graph when the average degree is 16.

The messages are usually too big to be RAM-resident. We will have a great performance penalty, if we buffer the messages on the disk and perform random accesses on the disk. To address this issue, we can cache the messages in a smarter way. For example, on each machine, we can differentiate remote vertices into two categories. The first category contains hub vertices, that is, vertices having a large degree and connecting to many local vertices. The second category contains the remaining vertices. We buffer messages from the vertices in the first category for the entire duration of one computation iteration. For a scale-free graph, for example, one generated by degree distribution $P(k) \sim ck^{-\gamma}$ with $c = 1.16$ and $\gamma = 2.16$, 20% hub vertices are sending messages to 80% of vertices. Even if we only buffer messages from 10% hub vertices, we have addressed 72.8% of the message needs.

5.4.4 Local Sampling

In a distributed graph processing system, a large graph is partitioned and stored on a number of distributed machines. This leads to the following question: Can we perform graph computations locally on *each machine* and then aggregate their results to derive the final result for the entire graph? Furthermore, can we use probabilistic inferences to derive the result for the entire graph from the result on *a single machine*? This paradigm has the potential to overcome the network communication bottleneck, as it minimizes or even totally eliminates the network communication. The answers to these questions are positive. If a graph is partitioned over ten machines, each machine has full information about 10% of the vertices and 10% of the edges. Also, the edges link to a large amount of the remaining 90% of the vertices. Thus, each machine actually contains a great deal of information about the entire graph.

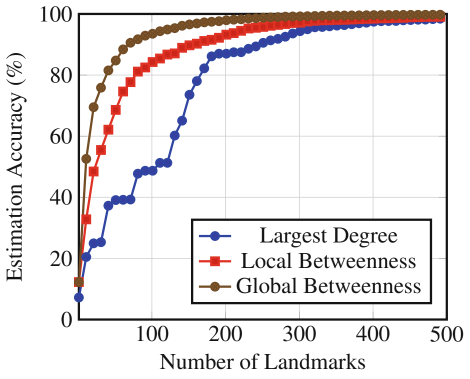


Fig. 5.5 Effectiveness of local sampling in distance oracle

The distance oracle proposed by Qi et al. (2014) demonstrated this graph computation paradigm. Distance oracle finds landmark vertices and uses them to estimate the shortest distances between any two vertices in a large graph. Figure 5.5 shows the effectiveness of three methods for picking landmark vertices. Here, the X axis shows the number of used landmark vertices and the Y axis shows estimation accuracy. The best approach is to use vertices that have the highest global betweenness and the worst approach is to simply use vertices that have the largest degree. The distance oracle approach uses the vertices that have the highest betweenness computed locally. Its accuracy is close to the best approach and its computation costs are dramatically less than that of calculating the highest global betweenness.

5.5 Alternative Graph Representations

In the previous sections, we assume the graph is modeled and stored in the adjacency list form. For a certain task, transforming a graph to other representation forms can help tackle the problem. This section covers three of them: matrix arithmetic, graph embedding, and matroids.

5.5.1 Matrix Arithmetic

A representative system is Pegasus by Kang et al. (2009). Pegasus is an open source large graph mining system. The key idea of Pegasus is to convert graph mining operations into iterative matrix–vector multiplications.

Pegasus uses an n by n matrix m and a vector v of size n to represent graphs. Pegasus defines an operation called *Generalized Iterated Matrix–Vector*

Multiplication (*GIM-V*).

$$M \times v = v', where\ v'_i = \Sigma_{j=1}^{n} m_{i,j} \times v_j$$

Based on it, three primitive graph mining operations are defined. Graph mining problems are solved by customizing the following three operations:

- *combine2*$(m_{i,j}, v_j)$: multiply $m_{i,j}$ and v_j;
- *combineAll*$_i(x_1, \ldots, x_n)$: sum the multiplication results from *combine2* for node i;
- *assign*(v_i, v_{new}): decide how to update v_i with v_{new}.

Many graph mining algorithms, including PageRank, random walk, and connected component, can be expressed elegantly using these three customized primitive operations.

5.5.2 Graph Embedding

Some graph problems can be easily solved after we embed a graph into a high-dimensional space as illustrated by Zhao et al. (2010, 2011) and Qi et al. (2014). This approach is particularly useful for estimating the distances between graph nodes.

Let us use an example given by Zhao et al. (2011) to illustrate the main idea of high-dimensional graph embedding. To compute the distance between two given graph vertices, we can embed a graph into a geometric space so that the distances in the space preserve the shortest distances in the graph. In this way, we can immediately give an approximate shortest distance between two vertices by calculating the Euclidean distance between their coordinates in the high-dimensional geometric space.

5.5.3 Matroids

As illustrated by Oxley (1992), any undirected graph can be represented by a binary matrix that in turn can produce a graphic matroid. Matroids usually use an "*edge-centric*" graph representation. As elaborated by Truemper (1998), instead of representing a graph as (V, E), we consider a graph as a set E of edges and consider graph nodes as certain subsets of E. For example, a graph $(V, E) = (\{a, b, c\}, \{e_1 = ab, e_2 = bc, e_3 = ca\})$ can be represented as follows: $E = \{e_1, e_2, e_3\}$ and $V = \{a = \{e_1, e_3\}, b = \{e_1, e_2\}, c = \{e_2, e_3\}\}$.

Matroids provide a new angle of looking at graphs with a powerful set of tools for solving many graph problems. Even though the interplay between graphs and matroids has been proven fruitful, as elaborated by Oxley (2001), how matroids can be leveraged to help design graph processing systems is still an open problem.

5.6 Summary

The proliferation of large graph applications demands efficient graph processing systems. Parallel graph processing is an active research area. This chapter tried to shed some light on parallel large graph processing from a pragmatic point of view. We discussed the challenges and general principles of designing a general-purpose, large-scale graph processing system. After surveying a few representative systems, we reviewed a few important graph computation paradigms for online query processing and offline analytics. Different graph representations lead to different graph computation paradigms; each of them may be suitable for solving a certain range of problems. At the end of this chapter, we briefly explored a few alternative graph representation forms and their applications.

References

Aggarwal CC, Wang H (eds) (2010) Managing and mining graph data. Advances in database systems, vol 40. Springer, Berlin

Aranda-Andújar A, Bugiotti F, Camacho-Rodríguez J, Colazzo D, Goasdoué F, Kaoudi Z, Manolescu I (2012) Amada: web data repositories in the amazon cloud. In: Proceedings of the 21st ACM international conference on information and knowledge management, CIKM '12. ACM, New York, pp 2749–2751

Atre M, Chaoji V, Zaki MJ, Hendler JA (2010) Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: WWW, pp 41–50

Bollobás B (1998) Modern graph theory. Graduate texts in mathematics, Springer, Berlin

Cheng J, Yu JX, Ding B, Yu PS, Wang H (2008) Fast graph pattern matching. In: ICDE, pp 913–922

Cohen J (2009) Graph twiddling in a mapreduce world. In: Computing in science & engineering, pp 29–41

Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell 26(10):1367–1372

Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51:107–113

Garey MR, Johnson DS, Stockmeyer L (1974) Some simplified np-complete problems. In: Proceedings of the sixth annual ACM symposium on theory of computing, STOC '74. ACM, New York, pp 47–63

Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, pp 17–30

Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) Graphx: graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX conference on operating systems design and implementation, OSDI'14. USENIX Association, Berkeley, pp 599–613

Gregor D, Lumsdaine A (2005) The parallel BGL: a generic library for distributed graph computations. In: Parallel object-oriented scientific computing (POOSC), POOSC '05

He H, Singh AK (2008) Graphs-at-a-time: query language and access methods for graph databases. In: SIGMOD

Holder LB, Cook DJ, Djoko S (1994) Substucture discovery in the subdue system. In: KDD workshop, pp 169–180

Husain M, McGlothlin J, Masud MM, Khan L, Thuraisingham BM (2011) Heuristics-based query processing for large RDF graphs using cloud computing. IEEE Trans Knowl Data Eng 23(9):1312–1327

Kang U, Tsourakakis CE, Faloutsos C (2009) Pegasus: a peta-scale graph mining system implementation and observations. In: Proceedings of the 2009 ninth IEEE international conference on data mining, ICDM '09. IEEE Computer Society, Washington, pp 229–238

Kaoudi Z, Manolescu I (2015) RDF in the clouds: a survey. VLDB J 24(1):67–91

Kyrola A, Blelloch G, Guestrin C (2012) Graphchi: large-scale graph computation on just a pc. In: OSDI, pp 31–46

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

Lumsdaine A, Gregor D, Hendrickson B, Berry JW (2007) Challenges in parallel graph processing. Parallel Process Lett 17(1):5–20

Majumder S, Rixner S (2004) An event-driven architecture for MPI libraries. In: Proceedings of the 2004 Los Alamos computer science institute symposium

Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 international conference on management of data, SIGMOD '10. ACM, New York, pp 135–146

Neumann T, Weikum G (2010) The rdf-3x engine for scalable management of RDF data. VLDB J 19(1):91–113

Oxley J (1992) Matroid theory. Oxford University Press, Oxford

Oxley J (2001) On the interplay between graphs and matroids. In: Surveys in combinatorics 2001. Cambridge University Press, Cambridge

Papailiou N, Konstantinou I, Tsoumakos D, Koziris N (2012) H2rdf: adaptive query processing on RDF data in the cloud. In: Proceedings of the 21st international conference on World Wide Web, WWW '12 Companion. ACM, New York, pp 397–400

Qi Z, Xiao Y, Shao B, Wang H (2014) Distance oracle on billion node graphs. In: VLDB, VLDB Endowment

Qin L, Yu JX, Chang L, Cheng H, Zhang C, Lin X (2014) Scalable big graph processing in mapreduce. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data, SIGMOD '14. ACM, New York, pp 827–838

Ravindra P, Kim H, Anyanwu K (2011) An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In: Proceedings of the 8th extended semantic web conference on the semanic web: research and applications - volume Part II, ESWC'11. Springer, Berlin, pp 46–61

Rohloff K, Schantz RE (2011) Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store. In: Proceedings of the fourth international workshop on data-intensive distributed computing, DIDC '11. ACM, New York, pp 35–44

Sarwat M, Elnikety S, He Y, Mokbel MF (2013) Horton+: a distributed system for processing declarative reachability queries over partitioned graphs. Proc VLDB Endow 6(14):1918–1929

Shao B, Wang H, Li Y (2013) Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data, SIGMOD '13. ACM, New York, pp 505–516

Sun Z, Wang H, Wang H, Shao B, Li J (2012) Efficient subgraph matching on billion node graphs. Proc VLDB Endow 5(9):788–799

Truemper K (1998) Matroid decomposition. Elsevier, Amsterdam

Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM 23(1):31–42

Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33:103–111 von Eicken T, Culler DE, Goldstein SC, Schauser KE (1992) Active messages: a mechanism for integrated communication and computation. In: Proceedings of the 19th annual international symposium on computer architecture, ISCA '92. ACM, New York, pp 256–266

Wang L, Xiao Y, Shao B, Wang H (2014) How to partition a billion-node graph. In: IEEE 30th international conference on data engineering, ICDE 2014, Chicago, March 31–April 4, 2014, pp 568–579

Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: HotCloud'10 proceedings of the 2nd USENIX conference on hot topics in cloud computing. USENIX Association, Berkeley, 18 pp.

Zeng K, Yang J, Wang H, Shao B, Wang Z (2013) A distributed graph engine for web scale RDF data. In: VLDB, VLDB Endowment

Zhang S, Li S, Yang J (2009) Gaddi: distance index based subgraph matching in biological networks. In: EDBT

Zhang X, Chen L, Tong Y, Wang M (2013) Eagre: towards scalable I/O efficient SPARQL query evaluation on the cloud. In: Proceedings of the 2013 IEEE international conference on data engineering (ICDE 2013), ICDE '13. IEEE Computer Society, Washington, pp 565–576

Zhao P, Han J (2010) On graph query optimization in large networks. PVLDB 3(1):340–351

Zhao X, Sala A, Wilson C, Zheng H, Zhao BY (2010) Orion: shortest path estimation for large social graphs. In: WOSN'10

Zhao X, Sala A, Zheng H, Zhao BY (2011) Fast and scalable analysis of massive social graphs. CoRR

Zhu F, Qu Q, Lo D, Yan X, Han J, Yu PS (2011) Mining top-k large structural patterns in a massive network. In: VLDB

Zou L, Chen L, Özsu MT (2009) Distancejoin: pattern match query in a large graph database. PVLDB 2(1):886–897