Graph Management Systems: A Qualitative Survey

Maurizio Nolé, Carlo Sartiani
DIMIE – Università della Basilicata
Via dell'Ateneo Lucano 10 – Potenza - Italy
e-mail: mau.nole@unibas.it, carlo.sartiani@unibas.it

Abstract

In the recent years many real-world applications have been modeled by graph structures (e.g., social networks, mobile phone networks, web graphs, etc.), and many systems have been developed to manage, query, and analyze these datasets. These systems could be divided into specialized graph database systems and large-scale graph analytics systems. The first ones consider end-to-end data management issues including storage representations, transactions, and query languages, whereas the second ones focus on processing specific tasks over large data graphs. In this paper we provide an overview of several graph database systems and graph processing systems, with the aim of assisting the reader in identifying the best-suited solution for her application scenario.

Keywords*: Big Data, Graph Data Management, Graph Processing Systems, Semistructured Data, Data Analytics*

1. Introduction

In the recent years many real-world applications have been modeled by graph structures (e.g., social networks, mobile phone networks, web graphs, etc.), and many systems have been developed to manage, query, and analyze these datasets; to cope with the ever-increasing size of graph datasets, some systems rely on a centralize architecture, while others adopt a distributed approach, which may further imply the need for specific data partitioning strategies [1] [2] [3] [4].

These systems could be divided into specialized *graph database systems* and *large-scale graph analytics systems*. Graph database systems deal with end-to-end data management issues, and offer typical databse features like ACID transactions and declarative query languages, whereas graph processing systems focus on assisting the user in writing applications for specific tasks over large data graphs. Graph database systems aim at modeling and querying data graphs by overcoming some of the limitations that may arise when using an RDBMS. Indeed, it is possible to store data graphs into a relational system and query them by using SQL and user-defined functions and aggregations. In particular, there are some works that describe how to implement, in an efficient way, graph queries in Relational Databases [5], as well as some tools that provide a specific graphical query, and take care of running queries on the underlying relational engine [5]. However, as shown in [6], relational join engines have been proved to be suboptimal on many graph queries, which may represent a significant issue when dealing with very large graphs.

Graph database systems are specifically designed to store graph data, to support flexible schemas, and to provide specialized query graph traversal languages; moreover, similarly to a traditional DBMS, they provide services such as persistence, transactions, query optimization, etc. Most of these systems, such as Neo4j [7], DEX [8], and HyperGraphDB [9], are efficient single-node systems with limited scalability; furthermore, to manage massive graphs, several distributed systems, such as Horton+ [10], and ThingSpan [11], have been designed. Unfortunately, as no standard query language has been defined for graph databases, each graph database is optimized for a specific set of tasks or queries, and each one implements its own API for querying and manipulating data.

Systems for processing and analyzing massive graphs generally use a distributed environment with more computing and memory resources, and most of them are built on top of shared-nothing architectures. The majority of these systems, e.g., Pregel [12], Giraph [13], GraphLab [14], GPS [15], and Pregel+ [16], adopts a vertex-centric computing approach inspired by the BSP model [17], but there are also other solutions, such as Trinity [18], SociaLite [19], CombBlas [20], and Pegasus [21], that adopt different approaches. All these systems, with the notable exception of SociaLite, have been designed for batch processing of specific algorithms, and do not support high-level query languages.

In this paper we provide an overview of several graph database systems and graph processing systems, with the aim of assisting the reader in identifying the best-suited solution for her application scenario. The rest of the paper is organized as follows. In Section 2 we focus our attention on graph database systems, with particular emphasis on Neo4J and its query language. In Section 3, then, we move to graph processing systems, and review the most important Pregel-inspired systems as well as a few high-level systems like SociaLite and Trinity. In Section 4, finally, we draw our conclusions.

2. Graph Database Management Systems

Graph Database Management Systems (GDBMSs) are mainly designed to support online transaction processing (OLTP) workloads for quick low-latency access to relatively small portions of graph data. GDBMs provide the major services of a traditional DBMS: persistence, transactions, query optimization, etc. These systems, such as Neo4j [7], DEX [22], and HyperGraphDB [23], are mainly centralized systems. Some of them can provide a distributed architecture for high availability and fault tolerance, but their horizontal scalability is limited by data locality issues and the lack of sharding strategies. Indeed, if one wants process large graphs that cannot be stored in main memory, the above systems begin to underperform. Random disk access, the lack of efficient data partitioning methods, the inability to distribute the computation on a cluster become a significant performance and scalability bottleneck. To overcome these problems distributed graph databases have been carefully designed, e.g., Horton+ [10], and ThingSpan [11].

Each of these graph database systems implements methods for querying the graph dataset. Indeed, some systems, such as Dex, implement APIs with special functions for querying graph properties. Neo4j provides Cypher, a graph-oriented query language based on expressions of the form *Start-Match-Where-Return.* ThingSpan allows one to traverse the graph through the implementation of Java classes, but also supports a declarative query language. Finally, RDF stores like AllegroGraph [24] support SPARQL [25], the standard query language for RDF data. None of the above languages provides a formal syntax and semantics, except for SPARQL (Cypher semantics has been formalized only very recently [26].

2.1. Neo4J

Neo4j is an open-source project, written in Java. It works on a network-oriented model with relations as first class objects. Neo4j represents Nodes and Relationships as Java objects, and materializes these objects once at insertion time. Data are stored on disk through an optimized data structure for graph networks, using an *index-free adjacency lists* architecture, where each node has explicit references to its adjacent nodes, and the check for the existence of an edge between two linked nodes does not require the access to an external, global, index.

Neo4j uses the *Property Graph Model*, where the graph is directed and is modeled using nodes, relationships, and properties on nodes and relationships. These components are stored in three separate store files. In order to reduce latency, Neo4j provides two levels of caching: *Filesystem Cache*, and *Object Cache*. The first one divides the store files into pages which are held in main memory (RAM), while the second one maintains nodes and relationships as Java objects in the Heap. Neo4j supports also ACID transactions by implementing a write-ahead log (WAL).

In Neo4j there are several methods for managing and querying graphs: *Core Java API*, *Traversal API*, and the query language *Cypher*. The *Core Java API* allows one to use low-level data structures to manage and query a graph; this solution is powerful and flexible for traversing a graph, but for a complex traversal case the implementation could become quite cumbersome. The *Traversal API* is a framework that allows one to build traversal rules without sacrificing any of the power of graph traversal, in a simple and declarative manner and with minimal performance impact. As the *Core API* and *Traversal API* are difficult to use in complex cases, Neo4j also provides the query language Cypher to query and manipulate an input graph. Moreover, Neo4j is accessible from all popular programming languages (e.g. Java, Python, Ruby, PHP, .NET, etc.), via an HTTP/S REST interface, and via the Blueprint interface part of the TinkerPop software stack. This last interface allows one to use different graph query languages such as Gremlin [27], an imperative language compatible with several graph databases.

Cypher is a pattern-matching query language, and uses a declarative grammar with clauses, similar to SQL. The syntax consists of four different clauses: *Start*, *Match*, *Where*, and *Return. Start* is an optional clause that allows the user to choose the starting nodes of the graph being analyzed by specifying their IDs. *Match* matches graph patterns, allowing one to locate the subgraphs of interest. *Where* filters out data based on some criteria. *Return*, finally, returns the results the user is interested in.

Cypher supports aggregation functions (e.g., *COUNT*, *SUM*, *AVG*), and several functions that can be used to evaluate expressions in a query (e.g., *FOREACH*, *WITH*, *TYPE*, *HAS*, *NODES*). Finally, Cypher allows one also to create, update, and delete nodes, relationships, and their properties (e.g., *CREATE*, *DELETE*, *SET*).

Example 1.

Consider the graph in Figure 1, sketching a social network, and the following Cypher query.

```
START user = node(name='Maria')
MATCH (user) - [:TAG]  - >  photo
RETURN photo;
```

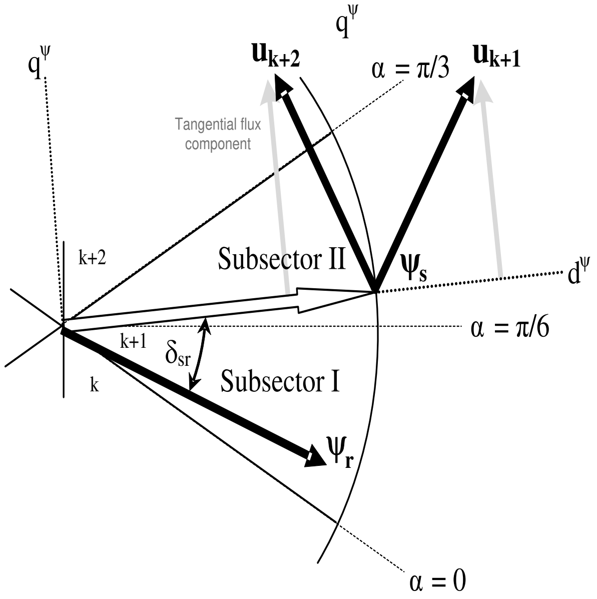

Figure 1. Social network data graph.

The **START** clause on the first row specifies one or more starting points (nodes or relationships) in the graph: here, it looks up for a node whose name is *Maria*. The **MATCH** clause specifies a pattern that is a description of the subgraphs of interest and, here, consists of two nodes connected with relationships, represented in the format *()-[]->()*: nodes are specified using parenthesis, while, relationships are specified using square brackets; nodes and relationships are linked using hyphens. In this query, the **MATCH** clause looks for *photo* nodes that can be reached from *Maria* node by traversing a single edge labeled with *TAG*.

Cypher supports quite complex pattern matching expressions, such as the concatenation of relations, as shown in the following query:

```
START user =node(name='Maria')
MATCH (user) - [:FRIEND]−> ()
                  − [:TAG]− > photo
RETURN photo;
```

This query, starting from a known node user with name *Maria*, searches for all photo nodes that are reachable through the path *FRIEND.TAG*, i.e., all nodes that represent photos tagged by *Maria's* friends.

Just as in SQL, a **WHERE** clause filters the result on the basis of some property of nodes or relationships: in particular for string properties, in addition to the standard equality (=) comparison, one can use regular expressions to filter out specific values, placed between two forward slashes (/), as illustrated in the following query.

```
START user = node(name=’Maria’)
MATCH (user) - [: FRIEND] − > friend
WHERE friend.email = /.*@gmail.com/
RETURN friend;
```

This query looks for all *Maria’s* friends who have an email address containing "gmail.com". While quite powerful, Cypher matching features are far beyond those of regular path query languages like GXPath [28][29][30].

Neo4j is designed as a single-machine database system and limited by the resources of a single machine; hence, the performance of Neo4j becomes significantly worse when the graph exceeds the memory capacity.

Neo4j databases can be distributed across multiple machines. Neo4j makes use of a master-slave replication architecture, using a Paxos-like protocol, and provides support for resilience and fault tolerance in the event of hardware failures, and the ability to scale Neo4j for read-intensive scenarios. A first consequence of the distribution model is that the database consistency property is loosen to *eventual consistency*, while the rest of ACID characteristics stays the same. Finally, Neo4j, uses a technique known as *cache sharding*, that is not the same as *traditional sharding*. In traditional sharding different parts of the data are stored on different instances, often on different physical servers, in order to scale large databases while at the same time maintaining a predictable level of performance as the data grows. The cache sharding of Neo4j, instead, is essentially a routing-based pattern. Indeed, each server always holds the full dataset, but caches a separate part of the graph, simply due to the way requests are routed. The strategy is highly effective for managing a large graph that does not fit in main memory, but do not allows one to scale large databases by dividing it across multiple machines, as in traditional sharding, and the size of database stays limited by the memory of a single machine.

To overcome some of these limitations, very recent versions of Neo4J provide support for the evaluation of Cypher queries on top of Apache Spark.

2.2. Horton+

Horton+ is a distributed system for processing declarative reachability queries over a partitioned graph, hosted in the main memory of a cluster of servers. The system is implemented in C# and consists of a client interface, a query language (parser and compiler), a query optimizer, and a distributed query processor. Horton+ arises from Horton [31], an early version of the system, to whom several features have been added. Horton+ employs a declarative query language that uses regular path queries to express reachability queries over the attributes labeling the graph.

Horton+ use an attributed multi-graph $G = (V, E)$, that consists of a set of nodes V, and a set of edges E. A node represents an entity with a primary key, a categorical type (e.g., person, photo, or event), and a set of attributes (e.g., year, age). An edge represents the relationship between two nodes, with a categorical type (e.g., friend, tag, brother), and a set of attributes (e.g., edge direction and edge weight). The graph can be both directed and undirected. One can use multiple edges to link two nodes, each one representing a different relationship. Moreover, in case of directed graph, each node stores both inbound and outbound edges.

Example 2.

Consider again the graph of Figure 1. In this graph the node types are Person (Anna, Giulia, Rocco, Lucia, Maria), and Photo (Photo1), while the edge types are Friend, Married, Brother, and Tag. The figure shows that *Giulia* is friend of *Anna*, and *Maria* is married with *Rocco*, and *Photo1* node is tagged by *Lucia*, *Maria*, and *Rocco*. The node *Rocco* has an attribute *Age*, while the node *Photo1* has an attribute *Year.* The graph is partitioned across two partitions.

Horton+ query language allows the user to express regular path queries extended with powerful node and edge predicates. The grammar of this language is reported below:

$$
\begin{array}{lll}
Query & ::= & NodePred \\
 & & Query - EdgePred - Query \\
 & & (Query\ OR\ Query) \\
 & & Query(-EdgePred - Query)^* \\
 & & (Query - EdgePred-)^*Query \\
 & & Query(-EdgePred - Query)^+ \\
 & & (Query - EdgePred-)^+Query \\
NodePred & ::= & Id|NodeType|NodeType\{(AttrPred)^+\} \\
 & & (NOT\ NodePred) \\
 & & (NodePred\ AND\ NodePred) \\
 & & (NodePred\ OR\ NodePred) \\
EdgePred & ::= & EdgeType \mid EdgeType\{(AttrPred)^+\} \\
 & & (NOT\ EdgePred) \\
 & & (EdgePred\ AND\ EdgePred) \\
 & & (EdgePred\ OR\ EdgePred) \\
AttrPred & ::= & Operand\ BinaryOperator\ Operand \\
Operand & ::= & AttributeName \mid AttributeValue \\
NodeType & ::= & Node \mid TypeId \\
EdgeType & ::= & Edge \mid TypeId
\end{array}
$$

In this grammar *Query* is a start symbol, and a query starts with a node predicate, possibly followed by a sequence of edge and node predicate pairs. Closures are supported by using the Kleene star $*$ and +. A node predicate can specify a node *id*, a node type (e.g., *Photo*), but it can also match any node (i.e., *Node*). Moreover, a node predicate may contain predicates on node attributes (e.g., *Photo{year = 2015}*), and can be composed (e.g., *Photo OR Video*). Similarly, an edge predicate specifies an edge type (e.g., Tag, Friend, Edge) and can also specify multiple predicates on edge attributes. Finally, one can use < and > symbols to represent edge directions.

Example 3.

Consider again the graph of Figure 1. If we want to find all photos where Lucia and Maria are tagged, we can use the following query:

$$Q_1 =' Lucia' -tag > Photo-tag < -'Maria'$$

The first node predicate specifies a node id ('Lucia'), the second node predicate provides the node type (Photo), and the third node predicate specifies another node id ('Maria'). The two edge predicates specify edge type (Tag).

Now, if we want to find all photos in which a friend of Lucia is tagged, we can run the follow query:

$$Q_2 = Photo - Tag < -Person - Friend -' Lucia'$$

Since the query language is declarative, the system is equipped with a query optimizer. The query optimizer can choose to run a plan among many execution plans. It uses graph statistics and enumeration algorithms to find the lowest-cost solution in a short amount of time, so as to reduce the query execution latency. The query processor receives an input query and returns the matched results. The input query is compiled into a query plan containing one or more deterministic finite automata (DFA) and algebraic graph operators like *select*, *traverse*, and *join*. Each operator has a clearly defined functionality and an efficient implementation, and they are the basic blocks used by the query processor. The *select* operator determines the set of starting nodes; the *traverse* operator receives a set of partial paths and the set of starting nodes, then traverses iteratively the graph to construct a path; the *join* operator receives two sets of matching paths from two query plans, and constructs longer paths by joining paths from these two sets. The query plan can be executed directly, or first optimized and then executed. The query optimizer builds a cost model for each operator and looks for cost-efficient ways to combine them. The graph is partitioned, through an effective graph partitioning algorithm, into disjoint partitions. Each partition is managed by a partition server that is responsible for managing its own subset of graph data. A server is designed as the coordinator, and responsible for query parsing, compilation, and optimization.

2.3. Sparksee

Sparksee (formerly known as DEX) is a commercial graph database system written in C++. Sparksee represents graphs as labeled and attributed multigraphs (*Property graphs*), and it stores graphs using a compressed bitmap-based data structure. Sparksee offers a partial support for ACID transactions, where the isolation and atomicity cannot be always guaranteed. There is also a high availability extension enabling horizontal scaling for large workloads, that uses the Master/Slave model with coordination through Apache Zookeeper. Sparksee provides a native API for Java, C++, .NET, and Python. There is no integrated query language, and the only way to manage and query a graph is through a native API. However, the database server can be remotely accessed via REST methods and it is compliant with the Blueprints interface, allowing one to use the Gremlin query language.

2.4. ThingSpan

ThingSpan is a federated database system able to analyze large-scale graphs. ThingSpan has been developed as a layer on top of Objectivity/DB [32], a distributed object-oriented database server, and extends Objectivity/DB with the ability to exploit Apache Spark and Apache Yarn to distribute and balance the computing load.

ThingSpan inherits from Objectivity/DB many of its features (i.e., scalability, distributed approach, parallel processing, graph partitioning, and full ACID support) and adds APIs designed for graph analytics. The APIs are provided in various languages (Java, C++, C#, Python); moreover, ThingSpan provides a REST interface and Blueprints support, that allows one to use Gremlin. ThingSpan uses a labeled directed multigraph data model, and its library provides two base classes, *BaseVertex* and *BaseEdge*, from which all the instances of vertices and edges should inherit or subclass. Furthermore, it uses a set of specific navigation classes, in order to query the graph, that can use the *predicate query language (PQL)* to specify regular path queries. PQL allows one to looks up the vertices and the edges, in the graph, according to the values of one or more of their attributes, or specify paths based on a pattern or sequence of hops. PQL provides a set of built-in operators that accept zero or more operands and perform arithmetic, relational, logical, path, and other comparison operations.

3. Graph Processing Systems

Graph processing systems are intended to perform off-line computations on very large graphs, and they generally use a distributed environment built on top of a shared-nothing architecture. Such systems focus on graph computations rather than graph querying. All these systems provide APIs in different languages (e.g., Java, C++, .NET), to implement specific graph algorithms, such PageRank, Single-source shortest path (SSSP), etc. Most of the systems in this class, such as Pregel [12], Giraph [13], GPS [15], and Pregel+ [16], are based on a vertex-centric approach introduced by Google in Pregel and inspired by Leslie Valiant's Bulk Synchronous Parallel model (BSP) [17]. Other systems, such as GraphLab [14], and MapGraph [33], are based on the Gather-Apply-Scatter (GAS) model, a variant of the BSP model, where it is also allowed an asynchronous execution. A rather different approach is used by distributed SociaLite [34], that defines a high-level graph query language based on *Datalog*, and allows one to describe graph algorithms with a few Datalog rules that a compiler can translate into a distributed computation. Trinity [18], finally, is a storage infrastructure and computation framework built on top of a cluster of interconnect machines, and can implement any computational paradigm for online queries or vertex centric offline analytics.

3.1. Pregel

Pregel is a distributed programming framework for processing large graphs. It is similar in concept to MapReduce, but Pregel computational model is more suitable for computations working on graphs with scale, in some case, of billions of vertices and trillions of edges. Pregel is implemented in C/C++ and the high-level organization of Pregel programs is inspired by Leslie Valiant's Bulk Synchronous Parallel model (BSP).

In Pregel programs are evaluated through a sequence of iterations, called *supersteps*. During a superstep the framework invokes a user defined function for each vertex, conceptually in parallel, that expresses the logic of a given algorithm. This function specifies the behavior at a single vertex *v* and a single superstep *i*: it can read messages sent to *v* in superstep *i*, send messages to other vertices that will be received at superstep *i+1*, and modify the state of *v* and its outgoing edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifiers are known. In this model, edges have no associated computation.

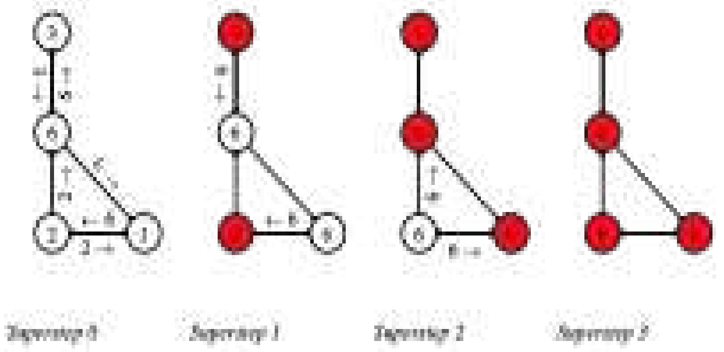

Figure 2: Pregel example.

We illustrate in Figure 2 an example where we use Pregel to compute the maximum value among the vertices. The graph consists of four vertices, each vertex containing a value. The algorithm propagates the largest value to every vertex, and, in each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbors. When no further vertices change in a superstep, the algorithm terminates.

Indeed, in superstep 0, each vertex sends a message with its own value to each connected vertex. In superstep 1, vertices compare their own value with the values contained in incoming messages: if their own value is larger than each received value, they vote to *halt* (red vertices); otherwise, the vertices change their own value with the highest received value. In superstep 2, active vertices send a message with their value as in superstep 1, and execute the comparison. In superstep 3, all vertices vote to halt, and vertex value is the maximum value.

The input to a Pregel computation is a direct graph in which each vertex is uniquely identified by a string vertex identifier. Each vertex is associated with a modifiable, user defined value. Directed edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier. A typical Pregel computation consists of an input phase, when the graph is initialized, followed by a sequence of supersteps separated by a global synchronization point until the algorithm terminates, and finishes with an output phase.

Algorithm termination is based on every vertex voting to halt. In superstep 0, every vertex is in active state; all active vertices participate in the computation of any given superstep. A vertex deactivates itself by voting to halt. This means that the vertex has no further work to do unless triggered externally, and the Pregel framework will not execute that vertex in subsequent supersteps unless it receives a message. If reactivated by a message, a vertex must explicitly deactivate itself again. The algorithm as a whole terminates when all vertices are simultaneously inactive and there is no message in transit. The output of a Pregel program is the set of values explicitly output by the vertices. It is often a directed graph isomorphic to the input, but this is not a necessary property of the system because vertices and edges can be added and removed during computation.

Pregel uses a cluster architecture consisting of thousands of commodity PCs. The graph is divided into partitions, each one consisting of a set of vertices and all of those vertices's outgoing edges, and each partition is assigned to a worker machine. The assignment of a vertex to a partition is decided, generally, through a hash function, but custom assignment functions may be implemented. The system consists of a master and several workers, where the master is responsible for coordinating workers activity, assigns partitions and user's input to workers, instructs each worker to perform a superstep, and after the computation halts, it may instruct each worker to save its portion of the graph. Each worker is responsible for maintaining the state of its section of the graph, executing the user functions on its vertices, and managing the messages.

3.2. Giraph

Giraph is an Apache open source project originated as a counterpart to Pregel, and adds several features beyond the basic Pregel model, including master computation, sharded aggregators, edge-oriented input, and out-of-core computation. A Giraph computation runs in the Map phase of a Hadoop job, hence any existing Hadoop user can immediately benefit from Giraph. Workers use ZooKeeper to select a master that will coordinate computation. The graph is loaded and partitioned across workers. The master then dictates when workers should start computing consecutive supersteps. Once the computation has halted, workers save the output. Checkpoints are initiated at user-defined intervals and are used for automatic application restarts when any worker fails.

Apart from a rich library of predefined graph algorithms, Giraph offers several mechanisms that help implementing new algorithms at scale. First, it is possible to acquire input vertices and edges from any input source, ranging from text files to NoSQL systems. Second, aggregators allow applications to compute a global value from contributing values provided by each vertex, which may reduce the network traffic. Finally, it is possible to decide to store the values and messages on disk, for example on a Hadoop cluster with limited memory but ample disk space, so as to improve the scalability of the system.

A tool for processing graph regular path queries on top of Giraph has been shown in [35][36].

3.3. GPS

GPS (Graph Processing System) [15], is an open-source system developed at Stanford University. GPS has three new features that do not exist in Pregel: global computation, dynamic repartition, and large adjacency list partitioning (LALP). While Pregel can implement *vertex-centric* algorithms only, GPS has an extension that enables efficient implementation of algorithms composed of one or more vertex-centric (parallel) computations, combined with global (sequential) computations, through the special function *master.compute()* called at the beginning of each superstep. Unlike Pregel, GPS can repartition the graph dynamically across compute nodes, on the basis of their message-sending patterns, during the computation, to reduce communication: GPS, indeed, attempts to collocate together vertices that send each other message frequently. Furthermore, in many graph algorithms each vertex sends the same message to all of its neighbors; in this case GPS LALP optimization stores partitioned adjacency list for high-degree vertices across the compute nodes on which the neighbors reside. The input graph is stored in HDFS files in a simple format: each line start with the *id* of a vertex *v*, followed by the *ids* of *v's* outgoing neighbors. The input file may optionally specify values for the vertices and edges. GPS assigns the vertices of G to worker using a simple round robin scheme, but it can use other sophisticated partitioning schemes.

3.4. Pregel+

Pregel+ is implemented in C/C++ and each worker is an MPI process. Pregel+ wrt other Pregel-like systems introduces two technique to reduce the number of messages: vertex mirroring and a request-respond paradigm.

Mirroring is designed to mitigate the problem of imbalanced workload by eliminating bottleneck vertices having a high degree. The main idea is to construct mirrors of each high-degree vertex in different machines, so that messages from a high-degree vertex are forwarded to its neighbors by its mirrors in local machines. The Request-Respond API allows a vertex (source) to request another vertex (target) for a value, and the requested value will be available at the source vertex in the next iteration. All requests from a machine to the same target vertex are merged into one request, to obtain the reduction of the number of messages passed between two machines.

3.5. GraphLab

GraphLab is a high performance, distributed computation framework written in C++. GraphLab 2.2 is last version of GraphLab [37], and includes the features of PowerGraph [38]. The latest version adopts the *Gather, Apply, Scatter (GAS)* model of computation and shared memory abstraction.

A general graph-parallel abstraction consists of a graph- and a vertex-program which is executed in parallel on each vertex. The GAS model is a vertex-centric graph-parallel abstraction similar to BSP, and it represents three conceptual phases of a vertex-program: Gather, Apply and Scatter.

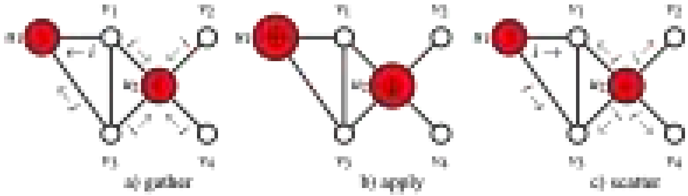

Figure 3: GAS phases.

In the *Gather* phase (Figure 3-a) each active vertex (red vertices in Figure 3-a) runs a vertex-program that accumulates information from adjacent vertices and edges, through a user-defined operation that must be commutative and associative, and can be a numerical sum or the union of the data. In the *Apply* phase (Figure 3-b) the resulting value of this operation is used to update the value of the active vertex. Finally, in the *Scatter* phase (Figure 3-c) the active vertex updates the adjacent vertices and edges, and activates its neighboring vertices.

GrapLab has several differences wrt Pregel. Each vertex in Pregel can receive information only through the messages sent from its neighbors, while, in the Gather phase, the vertex can directly pull data from its neighbors. Moreover, GraphLab provides both synchronous and asynchronous scheduling. In GraphLab synchronous scheduling, as in Pregel, uses communication barriers, but, while in Pregel the vertices that must send last messages must stay active, in GraphLab these vertices do not participate to the computation because their neighbors can pull their last value in Gather phase. In the asynchronous mode, instead, there are not barriers or supersteps: during the Apply phase the changes made to vertices or edges are committed immediately and are usable by any sequent computation phase. Unfortunately, the asynchronous execution, in order to avoid conflicts, uses distributed locking/unlocking protocols.

Finally, in Pregel partitioning does not replicate the vertices and cut edges. In GraphLab, instead, vertex-cut partitioning is used, where each edge is assigned to a unique machine, while the vertices are replicated in the caches of remote machines. In this way, graphs with skewed degree distribution can be partitioned across multiple machines, yielding better balanced workloads; the drawback is extra communication among worker to guarantee the consistency of the vertex value on each replica.

A user must implement a user-defined GAS function for each vertex. Furthermore, in the initialization phase, through a MapReduce job, she must construct, using a partitioning algorithm and the raw graph data, the *atom files* representing the data graph of each partition: in GraphLab each partition is called atom, and an atom index file stores the connectivity structure and the locations of atoms, both stored on the distributed file system.

In a GraphLab cluster one instance of the GraphLab program is executed on each machine. GraphLab processes are symmetric and directly communicate with each other using an asynchronous RPC protocol over TCP/IP. The first process is the master and has the responsibility of assigning the atom files to individual execution engine, reading the atom index file, and then, monitoring the machine.

3.6. MapGraph

MapGraph is a high performance parallel graph programming framework exploiting modern *GPUs*. The framework provides the APIs, based on the Gather-Apply-Scatter (GAS) model as used in GraphLab, to implement a wide range of graph algorithms, hiding the complexity of the GPU architecture.

3.7. SociaLite

SociaLite is a high-level graph query language based on *Datalog*. Datalog is a *declarative logic programming language* used as a query language in deductive databases. It can express in a concise way many graph algorithms, because of its high-level declarative semantics and support for recursion. SociaLite extends Datalog with two main features: *tail-nested tables* and *recursive aggregate functions*. SocialLite, instead of using two dimensional tables as in relational database systems, use a *tail-nested table* which is a generalization of the adjacency list. The last column of the table may contain pointers to two-dimensional tables, whose last columns can themselves expand into other tail-nested tables. Moreover, Sequential SociaLite supports recursive aggregate function, where an aggregate function can depend on itself. SociaLite has a number of pre-defined aggregate functions such as $Sum, $Min and $Max, and allows users to define their aggregate functions in Java.

SocialLite consists of a compiler that accept a SocialLite program and additional Java functions. The compiler parses the code into an abstract syntax tree (AST), performs syntactic and semantic analysis, optimizes the AST, and generates Java source code. The generated code is then compiled by a regular Java compiler into bytecode, which is executed with the SociaLite runtime system.

Sequential Socialite is not suitable for analyzing large scale graphs in a distributed environment. For this reason, distributed SociaLite [34], a version optimized for large scale graphs analysis on distributed machines, has been implemented.

The SociaLite parallel engine consists of a master which interprets the Datalog rules and instructs the slaves to work. Each slave node repeatedly executes the rules upon the arrival of communications from other nodes, and it updates the internal tables or sends messages to remote nodes. Finally, slaves can make a checkpoint of the intermediate work on a fault-tolerant distributed file system to restore it if needed. The parallel SociaLite requires the user to specify how the tables must be sharded across the machines; SociaLite, then, automatically manages the execution across the distributed machines, generates the message passing code, and manages the parallel execution.

3.8. Trinity

Trinity is a general-purpose graph engine over a distributed memory cloud. Trinity is not a system that comes with comprehensive built-in graph computation modules, but it enables the development of such modules and hence empowers a large variety of graph applications from online graph query processing to offline graph analytics. Trinity implements a globally addressable distributed memory storage in the memory of a cluster of commodity machines, and provides a random access abstraction for large graph computation. The memory is essentially a distributed key-value store and consists of a memory storage module and a message passing framework, that provides mechanisms for concurrency control and fault tolerances. Trinity system consists of three components: slaves, proxies, and clients. The Slaves store graph data and perform computation on the data; proxies are optional and may serve as dispatch information from client to slaves and inverse; finally, clients communicate with slaves and proxies, and allow users to interact with cluster.

The memory cloud consists of memory trunks, and each machine hosts multiple memory trunks. To support fault-tolerance data persistence, these memory trunks are also stored in Trinity File System (TFS), a shared distributed file system similar to HDFS. A key-value store is created on top of the memory cloud: the keys are 64-bit globally unique identifiers, and the values are blobs of arbitrary length. To address a key-value pair, Trinity uses a hashing mechanism and maintains a replica of the addressing table on each machine. The addressing table provides a mechanism that allows machines to dynamically join and leave the memory cloud; this mechanism is useful when a machine fails, as the relative trunk is reloaded from TFS to other alive machines.

Trinity also provides a language called TSL (Trinity specification language), that allows one to define a graph schema, communication protocols, and computation paradigms. Through TSL scripts one can define the schema of the data, and eventually integrate the data with data coming from external sources, so that Trinity knows how to manipulate data [39]. Moreover, TSL also allows one to model network communications, such as message passing protocols (e.g. synchronous, asynchronous, etc.) that can be used in vertex based computing and other algorithms. Other examples of schema languages for data graphs are shown in [40][41]. All these languages are based on regular expression types [42][43][44][45][46][47][48] or on record types [49][50].

With the schema and communication protocols defined in TSL, Trinity can implement any computational paradigm for online queries or vertex centric offline analytics, such as in [51], where is proposed Trinity.RDF, a distributed in-memory RDF system, based on Trinity, that is capable of handling web scale RDF data.

4. Conclusions

In this paper we surveyed the most prominent tools for managing and analyzing big graphs. Graph database systems aim at extending traditional database features, like transactions and high level query languages, to graphs, but they usually fail in matching the scalability of graph processing systems; the latter systems, while very scalable, usually require the user to write low-level code to analyze and/or query an input graph, and do not offer any form of declarative access to the data.

References

[1] Bidoit N, Colazzo D, Malla N, et al. Partitioning XML documents for iterative queries. In *Proceedings of 16th International Database Engineering & Applications Symposium,IDEAS '12, Prague, Czech Republic, August 8-10, 2012 - ACM International Conference Proceeding Series*. 2012, pp. 51–60.

[2] Bidoit N, Colazzo D, Malla N, et al. Processing XML queries and updates on map/reduce clusters. In: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013 - ACM International Conference Proceeding Series*. 2013, pp. 745–748.

[3] Bidoit N, Colazzo D, Malla N, et al. Evaluating Queries and Updates on Big XML Documents. *Inf Syst Front* 2018; 20: 63–90.

[4] Bidoit N, Colazzo D, Sartiani C, et al. *Andromeda: A system for processing queries and updates on big XML documents*. 2015. In: *New Trends in Database and Information Systems - ADBIS 2015 - Short Papers and Workshops, Poitiers, France, September 8-11, 2015. Proceedings.*

[5] Jindal A, Rawlani P, Wu E, et al. VERTEXICA: Your Relational Friend for Graph Analytics! *PVLDB* 2014; 7: 1669–1672.

[6] Aberger CR, Tu S, Olukotun K, et al. EmptyHeaded: A relational engine for graph processing. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2016, pp. 431–446.

[7] Neo4j. Available at http://www.neo4j.org/.

[8] Martìnez-Bazan N, Muntés-Mulero V, Gómez-Villamor S, et al. Dex: high-performance exploration on large graphs for information retrieval. In: *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pp. 573–582.

[9] Iordanov B. HyperGraphDB: A Generalized Graph Database. In: *Web-Age Information Management - WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Jiuzhaigou Valley, China, July 15-17, 2010, Revised Selected Papers*, pp. 25–36.

[10] Sarwat M, Elnikety S, He Y, et al. Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs. *PVLDB* 2013; 6: 1918–1929.

[11] ThingSpan. Available at http://www.objectivity.com/products/thingspan/.

[12] Malewicz G, Austern MH, Bik AJC, et al. Pregel: a system for large-scale graph processing. pp. 135–146.

[13] Apache Giraph. Available at http://giraph.apache.org

[14] GrapLab. Available at http://graphlab.org

[15] Salihoglu S, Widom J. GPS: a graph processing system. p. 22.

[16] D.Yan J. Cheng YLWN, Bu Y. *Pregel+*. 2014.

[17] Valiant LG. A Bridging Model for Parallel Computation. *Commun ACM* 1990; 33: 103–111.

[18] Shao B, Wang H, Li Y. Trinity: a distributed graph engine on a memory cloud. pp. 505–516.

[19] Seo J, Guo S, Lam MS. SociaLite: Datalog extensions for efficient social network analysis. In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pp. 278–289.

[20] Buluç A, Gilbert JR. The Combinatorial BLAS: design, implementation, and applications. *IJHPCA* 2011; 25: 496–509.

[21] Kang U, Tsourakakis CE, Faloutsos C. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In: *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*. 2009, pp. 229–238.

[22] Dex. Available at http://www.sparsity-tecnologies.com/dex

[23] HyperGraphDB. Available at http://hypergraphdb.org

[24] Allegrograph. Available at http://www.franz.com/agraph/allegrograph

[25] SPARQL. Available at http:http://www.w3.org/TR/rdf-sparql-query/

[26] Francis N, Green A, Guagliardo P, et al. Cypher: An evolving query language for property graphs. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2018, pp.

1433–1445.

[27] Gremlin Language. Available at https://github.com/tinkerpop/gremlin/wiki

[28] Libkin L, Martens W, Vrgoc D. Querying graph databases with XPath. pp. 129–140.

[29] Nolé M, Sartiani C. A Distributed implementation of GXPath. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016. *CEUR Workshop Proceedings*. 2016.

[30] Colazzo D, Mecca V, Nolé M, et al. PathGraph: querying and exploring big data graphs. BT - Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018. 2018; 29:1-29:4.

[31] Sarwat M, Elnikety S, He Y, et al. Horton: Online Query Execution Engine for Large Distributed Graphs. In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pp. 1289–1292.

[32] Objectivity/DB. Available at http://www.objectivity.com

[33] Fu Z, Thompson BB, Personick M. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In: *Second International Workshop on Graph Data Management Experiences and Systems,GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, p. 2:1--2:6.

[34] Seo J, Park J, Shin J, et al. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *PVLDB* 2013; 6: 1906–1917.

[35] Maurizio N, Sartiani C. Processing regular path queries on Giraph. In: Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference, Athens, Greece, March 28, 2014. *CEUR Workshop Proceedings*. 2014, pp. 37–40.

[36] Nolé M, Sartiani C. Regular path queries on massive graphs. In: Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016.

[37] Low Y, Gonzalez J, Kyrola A, et al. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB* 2012; 5: 716–727.

[38] Gonzalez JE, Low Y, Gu H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 17–30.

[39] Colazzo D, Sartiani C. Detection of corrupted schema mappings in XML data integration systems. *ACM Trans Internet Technol*; 9(4): 14:1-14:53 (2009)

[40] Colazzo D, Sartiani C. Typing query languages for data graphs. In: *Proceedings - International Conference on Data Engineering (ICDE)*. 2014, pp. 28–31.

[41] Colazzo D, Sartiani C. Typing regular path query languages for data graphs. In: *DBPL 2015 - Proceedings of the 15th Symposium on Database Programming Languages*. 2015, pp. 69–78.

[42] Ghelli G, Colazzo D, Sartiani C. Linear time membership in a class of regular expressions with interleaving and counting. In: *International Conference on Information and Knowledge Management (CIKM), Proceedings*. 2008, pp. 389–398.

[43] Colazzo D, Ghelli G, Sartiani C. Efficient inclusion for a class of XML types with interleaving and counting. *Inf Syst* 2009; 34: 643–656.

[44] Colazzo D, Ghelli G, Sartiani C. Efficient asymmetric inclusion between Regular Expression types. In: Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings - *ACM International Conference Proceeding Series*. 2009, pp. 174–182.

[45] Colazzo D, Ghelli G, Pardini L, et al. Linear inclusion for XML regular expression types. In: *International Conference on Information and Knowledge Management (CIKM), Proceedings*. 2009, pp. 137–146.

[46] Colazzo D, Ghelli G, Pardini L, et al. Almost-linear inclusion for XML regular expression types. *ACM Trans Database Syst*; 38(3): 15:1-15:45 (2013)

[47] Colazzo D, Ghelli G, Pardini L, et al. Efficient asymmetric inclusion of regular expressions with

interleaving and counting for XML type-checking. *Theor Comput Sci* 2013; 492: 88–116.

[48] Colazzo D, Ghelli G, Sartiani C. Linear time membership in a class of regular expressions with counting, interleaving, and unordered concatenation. *ACM Trans Database Syst*; 42(4): 24:1-24:44 (2017)

[49] Baazizi M-A, Lahmar HB, Colazzo D, et al. Schema inference for massive JSON datasets. In: *Advances in Database Technology - EDBT*. 2017, pp. 222–233.

[50] Baazizi M-A, Colazzo D, Ghelli G, et al. Counting types for massive JSON datasets. In: DBPL 2017: 9:1-9:12 - *ACM International Conference Proceeding Series*. 2017.

[51] Zeng K, Yang J, Wang H, et al. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 2013; 6: 265–276.