WGB: Towards a Universal Graph Benchmark

Khaled Ammar(✉) and M. Tamer Özsu

Cheriton School of Computer Science, University of Waterloo,
Waterloo, ON, Canada
{khaled.ammar,tamer.ozsu}@uwaterloo.com

Abstract. Graph data are of growing importance in many recent applications. There are many systems proposed in the last decade for graph processing and analysis. Unfortunately, with the exception of RDF stores, every system uses different datasets and queries to assess its scalability and efficiency. This makes it challenging (and sometimes impossible) to conduct a meaningful comparison. Our aim is to close this gap by introducing Waterloo Graph Benchmark (WGB), a benchmark for graph processing systems that offers an efficient generator that creates dynamic graphs with properties similar to real-life ones. WGB includes the basic graph queries which are used for building graph applications.

1 Introduction

Graph data are of growing importance in many recent applications including semantic web (i.e., RDF), bioinformatics, software engineering, e-commerce, finance, social networks, and physical networks. Graphs naturally model complicated structures such as chemical compounds, protein interaction networks, program dependence, product recommendation, fraud detection, social networks, computer networks, and web page connections. The size and complexity of these graphs raise significant data management and data analysis challenges. There has been a considerable amount of research in the last decade in analyzing and processing graph structures in real applications which leads to the development of a number of alternative algorithms, techniques and systems.

The validation of these algorithms/techniques/systems, and their comparison with each other, requires the use of a standard graph data set and a standard set of queries – i.e., a benchmark. Unfortunately, with the exception of RDF stores, every graph analysis system that has been proposed uses different datasets and queries to assess its scalability and efficiency. This makes it challenging (and sometimes impossible) to conduct meaningful comparisons.

In this paper, we propose a universal graph benchmark, called Waterloo Graph Benchmark (WGB), that is suitable for testing (parallel) graph systems, and that includes a wide range of operational, traversal and mining queries. A benchmark consists of two parts: (a) a data generator, and (b) a workload specification. The data generator produces the actual data over which the queries in the workload are executed for performance evaluation. It is generally desirable for the data and the workload to represent real applications. Furthermore,

T. Rabl et al. (Eds.): WBDB 2013, LNCS 8585, pp. 58–72, 2014.
DOI: 10.1007/978-3-319-10596-3_6

a benchmark should be complete, efficient, and usable. WGB features these requirements. It includes a representative query for each class of queries that might be used in graph data processing. We define a simple schema in Sect. 3 and specify multiple queries based on it. These are fundamental and common graph queries that many graph systems already support. WGB can efficiently generate very large datasets using limited resources in a short time, and can be easily tested by various graph systems with simple and straightforward interfaces. Although various data sets have been available for testing graph data management systems, WGB includes a simple and efficient data generator. It is the only graph generator that creates time-evolving graphs, that respects the power-law distribution for vertex degrees, and that provides full freedom in terms of nodes types, structure level, density and graph size.

In the following section, we discuss the existing benchmarks and clarify the motivation behind proposing a new one. Section 3 discusses the three main types of queries in graph systems and presents queries included in WGB that represent each class. Section 4 includes a discussion of the data generator followed by experiments to validate it. Section 6 briefly discusses the existing parallel/distributed graph systems, and Sect. 7 summarizes the paper and our future plans for WGB.

2 Existing Graph Benchmarks

We briefly review in this section the proposed benchmarks for graph analysis, pointing out some of their issues that guide us in the development of WGB. There are also a number of real-life datasets available in the literature such as those published in the Stanford Large Network Dataset Collection[1]. These real data sets are complementary to a benchmark. In general, one can use any available real-life dataset with a benchmark, but these datasets cannot be modified to meet different needs.

2.1 HPC Scalable Graph Analysis Benchmark

The idea behind HPC Scalable Graph Analysis Benchmark[2] is to develop an application with multiple kernels that access a weighted directed graph. These kernels represent multiple graph queries. They require irregular access to the graph so it is not expected that a single data structure for the input graph dataset would be optimal for all kernels. There are four kernels designed to address the following operations: bulk insertion, retrieving highest weight edges, k-hops operations, and betweenness calculation. However, the benchmark does not include many other graph operations such as updates or iterative queries [1].

[1] http://snap.stanford.edu/data/
[2] www.graphanalysis.org/index.html

2.2 Graph Traversal Benchmark

The authors of the Graph Traversal Benchmark [2] classify graph data management systems in two categories: (1) graph databases such as OrientDB[3], DEX[4], and Neo4j[5], and (2) distributed graph processing frameworks such as Pregel [3] and GraphLab [4]. This benchmark focuses on traversal operations on graph databases. Consequently, the capability of this benchmark is limited, because it does not support graph data queries or graph analysis algorithms. Moreover, it does not consider distributed graph processing frameworks.

2.3 Graph500

Graph500[6] is developed by a group of experts from academia, industry, and research laboratories. The main target is evaluating supercomputing systems for data intensive applications, focusing on graph analysis, to guide the design of hardware architectures and their software systems. Currently there is only one benchmark included in Graph 500, which is the HPC Scalable Graph Analysis Benchmark discussed earlier.

2.4 BigBench

BigBench [5] is a new benchmark that includes structured, semi-structured and unstructured data. The structured schema is borrowed from the Transaction Processing Performance Council's Decision Support benchmark (TPC-DS), the semi-structured part has user clicks on the retailer's website while the unstructured data has the online product reviews. The benchmark also includes a data generator and business queries for its schema. Although this benchmark covers a wide range of queries and data types, these are not graph-structured hence cannot be used as a graph benchmark.

2.5 BigDataBench

BigDataBench [6] is a big data benchmark suite. It is generic and includes various types of dataset: text, graph, table. The graph component uses Kronecker graph model [7] to create synthetic datasets based on real datasets. In this benchmark, the data generator is developed to learn the properties of a real graph and then generate synthetic data with similar characteristics [8]. Two real graph datasets are included: Google web graph, and Facebook social graph. Although Kronecker graph generator is able to create graphs similar to real ones and has a very good mathematical model, it has some disadvantages such as generation of multinomial/lognormal degree distributions instead of power-law

[3] http://www.orientdb.org/
[4] http://www.sparsity-technologies.com/dex
[5] http://www.neo4j.org
[6] http://www.graph500.org/ (accessed on 24th May 2013)

and being not flexible [9]. WGB generator, on the other hand, creates power-law distributions and is very flexible. It is capable of generating directed/undirected, weighted/unweighted, and bipartite/unipartite/multipartite graphs. Our benchmark is a domain specific big data benchmark focusing on graph data only. We need a flexible generator because we aim to assess graph systems using all potential graph datasets that may comply with known real graph properties.

2.6 RDF Benchmarks

For RDF stores, a number benchmarks have been proposed, such as the Berlin SPARQL Benchmark [10], DBpedia SPARQL Benchmark [11], LUBM [12], and SP^{2}Bench SPARQL [13]. These exclusively focus on RDF graphs whose properties are different than graphs seen in other application domains. It is interesting to note that the properties of RDF graphs used in these benchmarks are also different than the properties of real RDF graphs [14]. This has resulted in development of more realistic RDF benchmarks, such as WSDTS [15]. Furthermore, there are differences in the workloads. SPARQL is the standard query language for RDF systems and all benchmarks focus on SPARQL queries. These queries usually have specific structures (e.g., star shape) that are not evident in more general graph queries. Moreover, SPARQL cannot represent some popular graph algorithms such Pagerank.

3 Workload Characterization

There are three main workload categories in graph applications: online queries, updates, and iterative queries. *Online queries* are read only, interactive, usually do not have high complexity, and do not require multiple computation iterations – the answer should be computed quickly. *Updates* are also interactive, but they change the graph's structure by changing its parameters or its topology. Updates also include adding or removing a vertex or an edge. *Iterative queries* are heavy analytical queries, possibly requiring multiple passes over the data, and usually take a long time to complete. They may include some updates but the changes to the graph does not need to occur in real time.

In this section, we describe these queries assuming a relational database schema *Graph* that has the following three tables:

- $Nodes\{NodeId, \{p_i\}\}$, holds all the nodes in the graph and the properties of each node $\{p_i\}$.
- $Edges\{EdgeId, from, to, \{p_j\}\}$, holds the edges. For each edge, there are two foreign keys to the Nodes table pointing to the source ($from$) and destination (to) of this edge. Similar to nodes, edges may have a list of properties $\{p_j\}$.
- $ShortestPath\{from, to, length, path\}$. This table maintains the length and sequence of the shortest path between any two nodes in the graph. If the two nodes are not connected, i.e., not reachable, then the values of columns $length$ and $path$ are $null$.

It is important to note that we use these tables only to formalize the queries. The actual implementation of the graph data structure is up to each system as long as the result of each query is correct with respect to this schema.

Since there is no single programming language that all graph systems use (analogous to SPARQL for RDF), it is not possible to specify the queries in a language. Thus we specify queries abstractly using SQL over the schema specified above.

3.1 Online Queries

Most graph applications include only online queries such as a simple check of the availability of customers' relationship to a retail shop, or an update for the news feed in social media, etc. This class of queries also include graph matching queries.

Find Matching Nodes (Edges). The objective of this query is to find specific node(s) or edge(s). This is an online query and can be described in an SQL-like notation as follows. Note that X is a list of given parameter values.

```
SELECT NodeId FROM Nodes WHERE { p[i] = X[i] }

SELECT EdgeId FROM Edges WHERE { p[j] = X[j] }
```

The complexity of this class of queries vary based on the data structure(s) used in a system.

Find *k*-hop Neighbours. This query looks for neighbours that can be reached by up to k-hops from a specific node. It could be merged with the previous query to find neighbours who satisfy some conditions. A k-hop query from a given node u is the following:

```
SELECT S.to
FROM ShortestPath S
WHERE  S.from = u  AND S.length < (k+1)
```

Reachability and Shortest Path Query. For any two nodes $u, v \in G = \{V, E\}$, the reachability query from u to v returns true if and only if there is a directed path between them. A reachability query can be specified as:

```
SELECT true
FROM ShortestPath S
WHERE S.from = u AND S.to = v
AND S.length is not null
```

For any two nodes $u, v \in G = \{V, E\}$, the shortest path query returns the shortest path between u and v. The shortest path query between two nodes u and v can be written in SQL as:

```
SELECT path
FROM ShortestPath S
WHERE S.from = u AND S.to = v
AND S.length is not Null
```

There are two straightforward approaches to answer these two queries:

1. Build a breadth- or depth-first search over the graph from node u. This approach costs $O(|V|+|E|)$ to answer a query, where $|V|$ is number of nodes and $|E|$ is number of edges.
2. Build an edge transitive query closure of the graph to answer reachability queries in $O(1)$ time complexity which requires space $O(|V|^2)$. The transitive closure works as an index.

Most of the existing work on this problem try to optimize between these two extremes. There are many proposed approaches to reduce the index size and to minimize the complexity of answering the query [16].

In our experiments, we implemented these queries using breadth-first search, because at this stage we would like to eliminate the benefit of indexes and evaluate simply how a graph system performs traversal. In future, we may extend this to evaluating queries based on possible indexes in the system.

Pattern Matching Query. Given a data graph G and a query graph Q with n vertices (each query can be represented as a graph), any set of n vertices in G is said to match Q, if: (1) all $n \in G$ have the same labels as the corresponding vertices in Q; and (2) for any two adjacent vertices in Q, there is a path between the corresponding vertices in G. This is sometimes referred to as the *reachability pattern matching query*. A variation, that is called *distance pattern matching query*, introduces an additional query parameter δ such that for any two adjacent vertices in Q, the shortest distance between the corresponding vertices in G should be less than or equal to δ. A reachability pattern matching query input and output can be described as follows:

Input: Q specified as a list of edges $(FromLabel_i, ToLabel_i)$
Output: List of matching sub-graphs, each of which consists of a list of edges $(from_i, to_i)$ | Label$(from_i) = FromLabel_i$ and $Label(to_i) = ToLabel_i$ and Reachability$(from_i, to_i) = true$.

In the above description we use the Reachability(u, v) to represent a reachability SQL query from u to v. Similarly, we use Label(id) as a shorthand for the following SQL query:

```
SELECT label
FROM Nodes
WHERE NodeId = id
```

3.2 Updates

This category is simple but very important to evaluate a system's readiness for handling time-evolving (dynamic) graphs. Most available graph systems do not support online updates because these queries may invalidate indexing and partitioning decisions. A notable exception is G* [17]. A number of different updates are possible:

- Adding a new edge requires checking if either of its two nodes need to be added.
- The complexity of adding a new node depends on the underlying data structure. Many implementations use adjacency-vertex list. In this case, adding a new node simply means adding a new item to this list.
- Updates to physical data structures. Some systems store indexes to efficiently handle queries such as shortest path and reachability, these indexes should be maintained when a node (edge) is added to the graph.
- Updating and removing a node (edge) requires finding this node (edge) first then updating its information. Similar to adding a node (edge), this will lead to updates for all maintained indexes.

The set of updates can be specified as follows, where X is a list of given values for a node's or edge's parameters, u is a given node, and e is a given edge.

- `INSERT INTO Nodes VALUES (NodeId, { p[i] } )`
- `INSERT INTO Edges VALUES (EdgeId, from, to, { p[j] } )`
- `UPDATE Nodes SET { p[i] = X[i] } WHERE NodeId = u`
- `UPDATE Edges set { p[j] = X[j] } WHERE EdgeId = e`
- `DELETE FROM Nodes WHERE NodeId = u`
- `DELETE FROM Edges WHERE EdgeId = e`

3.3 Iterative Queries

Iterative queries are usually analytical workload over graph datasets (graph analytics). There is a large number of analysis algorithms such as subgraph discovery, finding important nodes, attack detection, diameter computation, topic detection, and triangle counting. There are several alternatives for each of these problems. In our benchmark we include two iterative graph analysis algorithms that represent the main theme of graph mining algorithms, "*iterate on graph data until convergence*". These two algorithms are Pagerank and Clustering.

Pagerank. Pagerank is the prototypical algorithm to check a graph system's performance in performing iterative queries. Almost all proposed graph analysis systems use it to evaluate their performance. In a nutshell, Pagerank finds the most important and influential vertices in a graph by repeating a simple mathematical equation for each node until the ranks of every node converge to a fixed-point.

Clustering. There are two main types of clustering algorithms: *node clustering* ones and *graph clustering* ones [18]. Node clustering algorithms operate on a single graph that they partition into k clusters such that each cluster has similar nodes. Graph clustering algorithms operate on (possibly large) number of graphs and cluster them based on their structure.

Given a connected graph, the main task of a clustering algorithm is to put similar graph nodes in the same group/cluster by minimizing an optimality criterion such as number of edges between clusters. Partitioning a graph into k clusters where $k \leq 2$ is solvable, as it can be mapped to the minimum cut problem. However, it is significantly more difficult, and is NP-hard, to optimally partition a graph when $k > 2$. Local optimum can be achieved by following techniques similar to k-medoid or k-means algorithms [18].

At this stage, we do not consider graph clustering algorithms in WGB. We have two clustering algorithms: the *Connected components* algorithm and the classical *k-means* algorithm. Connected component algorithm is a popular straightforward approach in node clustering where any two nodes belong to a cluster if there is an edge connects them. K-means algorithm picks k random data points as seeds and then assigns every other data point to one of these seeds (clusters). Guided by an objective function, it keeps searching for better seeds and move points between clusters. Its objective function is to minimize the sum of the distances between data points in a cluster [18].

4 Data Generation

We propose a new graph generation tool that creates graphs with real-life properties and allows users to control their behaviour. Our graph generator is based on RTG [19], which expands Millar's work [20]. Millar showed that typing random characters on a keyboard with only one character identified as a separator between words can generate a list of words that follow a power-law distribution. RTG builds on this by connecting each pair of consecutive words by an edge. The final result is a graph whose node degrees follow the power-law distribution. RTG enhances the properties of the generated graphs by introducing some dynamic graph properties such as shrinking diameter and densification [21]. Graph diameter is the longest shortest path between any two vertices in the graph. Shrinking diameter indicates that a graph's diameter decreases as the graph grows over time. Graph density is the ratio between the number of existing edges in a graph and the maximum possible number of edges assuming no multiple edges between any two vertices in the graph. Densification means that a graph's density increases over time.

Although the RTG generator is flexible and can generate graphs with real-life properties, it inherits several required parameters from Miller's process which are not strongly associated with graphs. Based on Millar's work, the generation process needs two main attributes: number of characters (k) and occurrence probability of the separator character (q). Setting these parameters to match desired graphs had not been considered. Our generator addresses this issue by

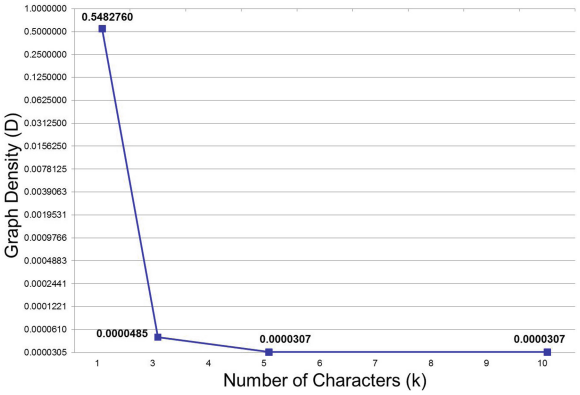

Fig. 1. The relation between the number of characters (k) and the density of the generated graph

only considering parameters directly related to graph properties, and then using approximation models to identify the other ones.

Experimentally we found that $k \in [5, 10]$ is a reasonable choice for generating many massive graphs. Increasing k too much does not have any significant influence on the generated graphs. A very small k generates graphs with high density (see Fig. 1), which is not realistic, and also influences their power-law distribution. A similar result was proven by Millar, where it was shown that changing k from 3 to infinity does not have a significant impact on the generated power-law distribution.

In the following we discuss the relationship between the graph density D and the occurrence probability of the separator character q. The following two equations were already proven [19]. W is the total number of generated edges in the graph, p is the occurrence probability of each character (except the separator character) and is a function of q and k such that $1 = q + k \times p$, N is number of unique vertices and E is number of unique edges:

$$N \propto W^{-log_p k} \tag{1}$$

$$E \approx W^{-log_p k} \times (1 + c' \times log\ W), c' = \frac{q^{-log_p k}}{-log\ p} > 0 \tag{2}$$

Graph density, D, is defined as $D = \frac{E}{N \times N - 1}$ assuming that self-loop edges are not allowed. Based on the above equations, we can write $D = fn(q, k, W)$. Then, we can show that $q = fn(D, k, W)$. However, these functions are very complex and already approximate. Taking the inverse of the D-function to get the q-function is difficult if not impossible. Therefore, we use an approximation technique based on Gaussian models. Based on a large training dataset, we built a Gaussian model to predict q based on D and W. In our experiments, the average error in predicting q was only 10 %.

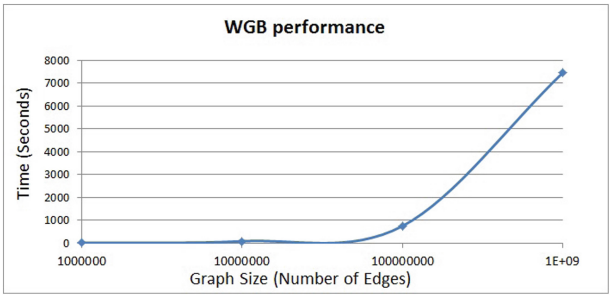

Fig. 2. Generator performance with respect to number of generated edges

Our generator has seven parameters: graph weight as total number of generated edges (W), partite flag (PF) which identifies if the graph should be unipartite or multi-partite, number of node types (NT), desired graph density (D), directed flag (DF) which identifies directed graphs, and the time frame (T) in case of a dynamic graph, which identifies number of timestamps the generated edges are assigned to, and structure preference (S). Note that this generator will generate repeated edges and it is up to the user to use multiple edges to build a weighted graph or ignore the repeated ones. The partite flag indicates whether or not nodes of the same type are allowed to connect to each other. This parameter allows our generator to create graphs commonly known as bipartite but have more than 2 node types. A node type is implemented by preparing a prefix for each type and appending it to the beginning of each node. T was introduced by RTG to create dynamic graphs where each group of edges are assigned to a time stamp.

Structure preference was also introduced in RTG to match the real-life graph property of having community structures, meaning that nodes form groups and possibly groups of groups. These groups are usually created by connecting similar nodes together to form a community. The structure preference, $S \in [0, 1]$, boosts the probability of generating similar nodes to be connected such that the probability of generating a character a in the label of both nodes is $P(a, a) = p - p^2 \times (k - 1) \times S - p \times q \times S$ while the probability of generating 2 different characters a and b is $P(a, b) = p^2 \times S$. Note that k is number of characters in the model, p is probability of choosing a character to be in the label, and q is the probability of generating the separator character.

Our generator is efficient (see Fig. 2) and was successfully used to generate graphs with billions of edges.

5 WGB in Action

It has been shown that the probability of all generated words (nodes) with i characters is higher than the probability of all words with $i + 1$ characters [20]. In other words, if the degree of each node is computed and all generated nodes

are sorted alphabetically based on their labels, then nodes with fewer characters are expected to have a higher degree. There are only k possible words with one character, k^2 possible words with two characters, and k^n possible words with n characters.

This property could be used when generating real-life graphs. For example, given a list of website names, we could generate a sample of the web graph by categorizing these websites into multiple categories based on their expected popularity. Then popular websites are assigned to nodes with one character and less popular websites are assigned to nodes with two characters and so on. The same approach could be used to generate a social network or any other graph given a sample of the most important entities (nodes) in that graph.

The same property could be used to customize the selectivity of online and update queries discussed in Sect. 3 by querying nodes with short labels to get high selectivity or nodes with long labels to aim for low selectivity. This is not relevant for iterative queries because they access the complete graph at least once. The workload queries do not have specific parameters. There is a non-trivial relation between generated data and the selectivity of each query. Given a dataset, the selectivity of a query depends on its parameter in the "where" clause. For example, a 1-hop neighbour query from a node with one character (high degree) has a high selectivity relative to the same query from a node with many characters. Using this approach, the parameters of the proposed workload queries could be customized to offer a diverse range of selectivity.

6 Graph Processing and Analysis Systems

Distributed graph systems in literature could be divided into two main categories: MapReduce-based systems and vertex-centric processing systems. We will analyze systems in both categories using WGB. The systems that we plan to analyse in the first category include Hadoop (http://hadoop.apache.org), Pegasus [22] and Haloop [23]. Those in the second category include Distributed GraphLab [4] and Pregel [3].

6.1 MapReduce-Based Systems

MapReduce [24] is a distributed data processing framework whose fundamental idea is to simplify parallel processing using a distributed computing platform that offers two interfaces: map and reduce. Programmers can implement their own map and reduce functions, while the system is responsible for scheduling and synchronizing the map and reduce tasks. MapReduce model can be used to solve the embarrassingly parallel problems[7], where little or no effort is required to partition a task into a number of parallel but smaller tasks. MapReduce is being used increasingly in applications such as data mining, data analytics and scientific computation. It has indeed been used by Google to implement Pagerank.

[7] http://parlab.eecs.berkeley.edu/wiki/media/patterns/map-reduce-pattern.doc

Despite the scalability and simplicity of MapReduce systems, the model has a few shortcomings in processing graph analysis algorithms. First, the data are distributed randomly in multiple partitions at every iteration. Graph analysis algorithms usually execute several iterations before the computation converges. Second, map and reduce functions should be executed in all partitions before the next iteration starts. This introduces a significant delay because there may be one partition that takes too long to finish, and this would delay the processing of all other partitions. Finally, it does not consider the sparsity of a graph. If only one data partition did not converge, the framework will consider all data partitions for another computing iteration instead of focusing on the non-converging data partition only.

Despite these shortcomings, we plan to evaluate the most popular open source MapReduce implementation, Hadoop, as a baseline for system evaluations. Part of the reason is that, all other MapReduce-based systems have compared themselves only to Hadoop. Although, their reported results demonstrate their superiority over Hadoop, each one uses a different version of Hadoop and sometimes even different hardware. Moreover, each system uses a different dataset with different sizes and different properties. Finally, each of these use different workloads. Our objective is to systematically compare these systems against a single baseline.

HaLoop and Pegasus were proposed to overcome some of Hadoop's problems. Pegasus is a matrix multiplication application implemented using Hadoop. The main proposed enhancement is changing how the data are stored and clustered. This enhancement decreases the data size and reduces the number of iterations to reach convergence.

HaLoop is a MapReduce variant developed specifically for iterative computation and aggregation. In addition to the map and reduce functions, HaLoop introduces AddMap and AddReduce functions to express iterative computation. To test the termination condition, the SetFixedPointThreshold, ResultDistance, SetMaxNumOfIterations functions are defined. To distinguish the loop-variant and loop-invariant data, AddStepInput and AddInvariantTable functions are introduced.

To avoid unnecessary scans of invariant data, HaLoop maintains a reducer input cache storing the intermediate results of the invariant table. As a result, at each iteration, the mappers only need to scan the variant data generated from the previous jobs. The reducer pulls the variant data from the mappers, and reads the invariant data locally. To reduce the time to reach a fixed point, the reducer output cache is used to store the output locally in the reducers. At the end of each iteration, the output that has been cached from the previous iteration is compared with the newly generated results to detect whether the fixed point is reached. To assign the reducers to the same machine in different iterations, the MapReduce scheduler is also modified.

6.2 Vertex-Centric Systems

Systems in the vertex-centric category are based on a simple idea: "Think as a vertex". These systems require the programmer to write one compute function that a vertex executes at every iteration. The compute function operates on each vertex and may update the vertex state based on its previous state and the message passed to it from the preceding vertices. As noted earlier, the iterations stop when the computation converges to a fix point.

The first system proposed in this category was Pregel [3] and is used by Google to execute the Pagerank algorithm. In Pregel, data are partitioned across multiple computing nodes, and each node picks a graph vertex in its partition to start executing the compute function. A node may decide not to participate in future iterations (which reduces the load) and can communicate with other vertices through messages. The system is synchronized in the sense that all computing nodes need to complete executing their compute function before the subsequent iteration can start.

The architecture of Distributed GraphLab [4] is similar to Pregel while the processing model is different: a user-defined update function modifies the state of a vertex based on its previous state, the current state of all of its adjacent vertices, and the value of its adjacent edges. An important difference between GraphLab and Pregel is in terms of their synchronization model, which defines how the vertices collaborate in the processing. Pregel only supports Bulk Synchronization Model (BSM) [25] where, after the vertices complete their processing of a iteration, all vertices should reach a global synchronization status, while GraphLab allows three choices: (fully or partially) synchronized and asynchronized.

7 Conclusion and Future Work

In this paper we proposed a new benchmark, WGB. WGB has a data generator built on top of the RTG generator, which is efficient and user friendly. In this generator we solved one of the main issues in the RTG generator by hiding the input parameter that is not related to graph properties. We also introduced more parameters to enable more flexibility. WGB workload specification includes three categories of graph queries: online, update, and iterative queries. As discussed, most graph applications will use combinations of the graph queries we introduced.

In the future, we are planning to use WGB to perform the first systematic experimental study of parallel/distributed graph systems such as Hadoop Map-Reduce [24], Pegasus [22] and HaLoop [23], Distributed GraphLab [4], and Pregel [3]. It is also interesting to investigate how WGB might be integrated with BigBench [5] by generating a graph that represent the relationships between customers, products and retail stores such that the properties of these relationships match real-life graph properties.

Acknowledgments. This research was partially supported by a fellowship from IBM Centre for Advanced Studies (CAS), Toronto.

References

1. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, N., Baleta, P., Larriba-Pay, J.L.: A discussion on the design of graph database benchmarks. In: Proceedings of 2nd TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems, pp. 25–40 (2011)
2. Ciglan, M., Averbuch, A., Hluchy, L.: Benchmarking traversal operations over graph databases. In: Proceedings Workshops of 28th International Conference on Data Engineering, pp. 186–189 (2012)
3. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 135–146 (2010)
4. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
5. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 1197–1208. ACM (2013)
6. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Y., Shi, Y., Zhang, S., et al.: Bigdatabench: A big data benchmark suite from internet services (2014). arXiv preprint arXiv:1401.1406
7. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: an approach to modeling networks. J. Mach. Learn. Res. **11**, 985–1042 (2010)
8. Ming, Z., Luo, C., Gao, W., Han, R., Yang, Q., Wang, L., Zhan, J.: BDGS: a scalable big data generator suite in big data benchmarking (2014). arXiv preprint arXiv:1401.5465
9. Appel, A.P., Faloutsos, C., Junior, C.T.: Graph mining techniques: focusing on discriminating between real and synthetic graphs. Bioinformatics: Concepts, Methodologies, Tools, and Applications, vol. 3, pp. 446–464. Information Resources Management Association, USA (2013)
10. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. J. Seman. Web Inf. Syst. **5**(2), 1–24 (2009)
11. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
12. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Seman.: Sci. Serv. Agents World Wide Web **3**(2), 158–182 (2005)
13. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2l SPARQL performance benchmark. In: Proceedings of 25th International Conferrence on Data Engineering, pp. 222–233 (2009)

14. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 145–156 (2011)
15. Aluç, G., Özsu, M.T., Daudjee, K., Hartig, O.: Chameleon-db: a workload-aware robust RDF data management system, University of Waterloo, Technical report, CS-2013-10(2013)
16. Yu, J., Cheng, J.: Graph reachability queries: a survey. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data. Advances in Database Systems, vol. 40, pp. 181–215. Springer, Heidelberg (2010)
17. Spillane, S.R., Birnbaum, J., Bokser, D., Kemp, D., Labouseur, A., Olsen, P.W., Vijayan, J., Hwang, J.-H., Yoon, J.-W.: A demonstration of the G* graph database system. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), Los Alamitos, CA, USA, pp. 1356–1359. IEEE Computer Society (2013)
18. Aggarwal, C.C., Wang, H.: A survey of clustering algorithms for graph data. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data. Advances in Database Systems, vol. 40. Springer, Heidelberg (2010)
19. Akoglu, L., Faloutsos, C.: RTG: a recursive realistic graph generator using random typing. Data Min. Knowl. Disc. **19**(2), 194–209 (2009)
20. Miller, G.A.: Some effects of intermittent silence. Am. J. Psychol. **70**(2), 311–314 (1957)
21. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: densification and shrinking diameters. ACM Trans. Knowl. Discov. Data, **1**(1), Article 2, pp. 1–41 (2007)
22. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system implementation and observations. In: Proceedings of IEEE International Conference on Data Mining, 2009, pp. 229–238 (2009)
23. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB J. **21**(2), 169–190 (2012)
24. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of 6th USENIX Symposium on Operating System Design and Implementation, pp. 137–149 (2004)
25. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)