Constructing Bisimulation Summaries on a Multi-Core Graph Processing Framework

Shahan Khatchadourian
University of Toronto
shahan@cs.toronto.edu

Mariano P. Consens
University of Toronto
consens@cs.toronto.edu

ABSTRACT

Bisimulation summaries of graph data have multiple applications, including facilitating graph exploration and enabling query optimization techniques, but efficient, scalable, summary construction is challenging. The literature describes parallel construction algorithms using message-passing, and these have been recently adapted to MapReduce environments. The fixpoint nature of bisimulation is well suited to iterative graph processing, but the existing MapReduce solutions do not drastically decrease per-iteration times as the computation progresses.

In this paper, we focus on leveraging parallel multi-core graph frameworks with the goal of constructing summaries in roughly the same amount of time that it takes to input the data into the framework (for a range of real world data graphs) and output the summary. To achieve our goal we introduce a *singleton optimization* that significantly reduces per-iteration times after only a few iterations. We present experimental results validating that our scalable GraphChi implementation achieves our goal with bisimulation summaries of million to billion edge graphs.

General Terms

Bisimulation, graph summaries, scalable, parallel

1. INTRODUCTION

Organizations generate substantial volumes and varieties of Big Data, and a recent business survey [1] highlights the analysis of Big Data as a key strategy for competitiveness. Often, organizations model semi-structured descriptions and relationships of real-world entities, such as from online social networks [14, 18] and links across the broader World Wide Web [4], as graphs with directed, labeled, multi-relational edges between nodes. For example, the quintessential semantic web dataset of the Linked Open Data (LOD) initiative, DBpedia, contains tens of millions of nodes and hundreds of millions of edges to describe entities such as people and locations.

A user who wishes to explore and query big graphs faces a challenge of querying a large number of nodes and their relationships. To improve query performance, we can build and use a *structural summary* [8], a graph-based index that groups equivalent structures using bisimilarity [10], a notion of equivalence with broad applicability, such as for model checking and information retrieval [21]. Bisimulation summaries provide valuable insight into describing the semi-structure of graph data [12] and existing literature shows that summaries can improve query performance [5, 20].

Related Work. For scalable construction of summaries, there exists an iterative, hash-based algorithm [3] that extends a simple algorithm [11] to iteratively update each node's block identifier using the block identifier of neighbours in parallel. Existing scalable implementations [3, 17, 22] rely on distributed frameworks like MapReduce [6] and message-passing to compute summaries of big graphs. There is literature that points to multi-core systems with large amounts of main-memory [2, 13] as a viable parallel processing architecture; however, there are no summary construction implementations for multi-core systems in the literature. In this work, we use GraphChi [15], a multi-core graph processing framework, to construct dual bisimulation summaries [7, 19] of big graphs with the goal of computing a summary in an amount of time similar to the time required to load the input dataset and then write the summary.

Contributions. First, we describe a summary construction implementation on a parallel graph processing framework (GraphChi). Second, we present a novel and very effective *singleton optimization* that allows us to achieve the goal of drastically reducing per-iteration times after only a few iterations. Third, we give an experimental validation that our GraphChi implementation achieves our goal of constructing summaries in an amount of time similar to the time required to load the dataset and then write the summary. We use in our experiments graphs ranging from millions to billions of edges. Fourth, we compare our GraphChi implementations to Hadoop and give evidence to support our approach on large multi-core systems.

Preliminaries A *dataset* graph $G = (V, E, l, L, m, M)$ satisfies the following properties: (1) V is a finite set of nodes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
GRADES'15, May 31 - June 04 2015, Melbourne, VIC, Australia
Copyright is held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3611-6/15/05...$15.00
DOI: http://dx.doi.org/10.1145/2764947.2764955.

with distinct labels; (2) l is a bijective label function that maps each $v \in V$ to a distinct label $l(v) \in L$, the set of node labels; (3) $E \subseteq V \times V$ is a finite set of labeled, directed edges, where an edge $(v, v') \in E$ between two nodes $v, v' \in V$ represents an edge from a node v to a node v'; and (4) m is a label function that maps each edge $e \in E$ to a label $m(e) \in M$, the set of edge labels.

We construct structural summaries that group nodes of a dataset graph using bisimilarity as the notion of equivalence.

Given a dataset graph $G = (V, E, l, L, m, M)$, we define two nodes $v, v' \in V$ as *forward bisimilar* (which we write as $v \approx_{fw} v'$) iff either of the following conditions hold: (1) if v and v' have no outgoing edges; or (2) if v has an edge $(v, w) \in E$ with label $p \in M$, there exists an edge $(v', w') \in E$ with label p, and $w \approx_{fw} w'$; and vice versa. A closely related counterpart to forward bisimilarity, *backward bisimilarity*, determines node equivalence along incoming edges. We say that two nodes v, v' are *dual bisimilar* (in short, *bisimilar*), which we write as $v \approx v'$, if v and v' are both forward and backward bisimilar.

We formalize the notion of a structural summary. A *summary graph* $S = (V, E, l, L, m, M, block)$ of a dataset graph G consists of nodes, which we call *blocks*, and edges between blocks, which we call *block edges*, and also having the fewest number of blocks (also called a *bisimulation contraction*). Each block contains equivalent dataset graph nodes in its *extent* and the function *block(v)* returns the block of V_S that contains a given dataset graph node $v \in V_G$. A summary's blocks form a *partition* of a dataset graph's nodes, a disjoint set of subsets. A *singleton* is a block whose extent contains exactly one node. As well, for every edge $(v, v') \in E_G$ with label p there is a *block edge* such that $(block(v), block(v')) \in E_S$ with label p. Bisimulation summaries have a minimal number of blocks such that each block contains all nodes in the dataset graph that are bisimilar. We call a FW summary one that is constructed using forward bisimulation as the notion of equivalence and, similarly, we call a $FWBW$ summary one constructed using dual bisimulation.

In Section 2, we describe GraphChi and its processing model, and give two summary construction variations. In Section 3, we give an evaluation of our GraphChi algorithms using big data graphs. We conclude in Section 4.

2. ALGORITHM

In this section, we describe GraphChi's processing model, a hash-based summary construction algorithm, and our B and S variations. In the next section, we give evaluations of our variations and comparisons to existing literature.

GraphChi is a multi-core processing framework that supports the Bulk Synchronous Parallel (BSP) [23] processing model, an iterative, node-centric processing model by which nodes in the current iteration execute an *Update* function in parallel that depends on values from the previous iteration. Targetting scalability on 'just a laptop', GraphChi partitions graph that do not fit in main memory in a way that makes it efficient to load and process subgraphs that fit in main memory. GraphChi uses *parallel sliding windows* (PSW), a subgraph partitioning and loading mechanism that involves both main-memory and secondary storage. GraphChi partitions a graph's nodes into equal *intervals*, storing each interval's incoming edges in a *shard* file in sorted order by the source node.

In each iteration, the PSW mechanism processes each interval in turn. For each interval, PSW first loads incoming edges from disk into main-memory then, since other shards sort edges by their source, the PSW mechanism streams the current interval's outgoing edges from disk. GraphChi processes an interval by executing the *Update* method of each node in the interval using parallelism.

Summary construction on GraphChi. Our summary construction algorithm uses the BSP processing model with the parallel, hash-based approach of [3]. The hash-based approach iteratively updates each node's block identifier by computing a hash value from the node's signature, the set of block identifiers from the previous iteration to which the node's neighbours belong to. The main idea is that two bisimilar nodes will have the same signature, the same hash value, and thus have the same block identifier. As well, two nodes that are not bisimilar will not have hash values that collide and will have different block identifiers. A parallel variant of this notion enables executing multiple node update methods at the same time since new block identifiers are computed using values from the previous iteration. However, the PSW mechanism only loads adjacent edges during execution of a node's *Update* method. This means that a node can read and write its own properties, the properties of connected edges, and shared-memory data structures, but cannot read or write the properties of neighbours. GraphChi suggests a BSP implemention that stores each node's previous and current block identifier within its edges. This means that each edge will have four values, two pertaining to the edge source's current and previous block identifiers, and the remaining two of the edge target's current and previous block identifiers, thus increasing duplication and serialization cost. During subsequent iterations, block identifiers are read from one pair of source and target block identifiers, which represent the values from the previous iteration, and then written to the counterpart pair, which represent the values of the current iteration. Then, on the next iteration, block identifiers are read from the second pair, and the first pair will receive updated block identifiers. Anecdotally, although the performance of this BSP implementation is quite good, we choose to implement the model slightly differently.

We implement two variations: variation B avoids storing node properties as edge properties, and variation S which provides an optimization to improve performance. We now describe the two variations.

The algorithm in Figure 1 gives our B variation's program and node update method, *V-Update*. The GraphChi program takes a dataset graph as input and initializes its summary as a single block containing all nodes (L1). In each iteration k, GraphChi executes the *V-Update* on each dataset graph node in parallel (L4); we use the keyword *parallel foreach* to represent how GraphChi processes all of a graph's nodes using parallelism. Our algorithms keeps each node's current and previous iteration block identifiers in two hashmaps A node's update function takes its forward signature from its outgoing edges (V1) and its backward signature from its incoming edges (V2) to compute the (k+1)-dual block identifier (V3). If we exclude line V2 then we compute a FW summary. The algorithm terminates when two consecutive iterations have the same number of distinct block identifiers (L3,L5,L6). The algorithm writes the summary to disk in parallel (L9), skipping duplicate block edges.

B variation computation of structural summary

L1. **foreach** $v \in V_G$ **do** $block_0(v)$ = '0' **end** // initialize
L2. $count_{old} \leftarrow 1, count_{new} \leftarrow 0, k \leftarrow 0$
L3. **while** $count_{old} \neq count_{new}$ **do**
L4.   **parallel foreach** $v \in V_G$ **do** V-Update(v) **end**
L5. $count_{old} \leftarrow count_{new}$
L6. $count_{new} \leftarrow |\{block_{k+1}(v) \mid v \in V_G\}|$
L7. $k \leftarrow k + 1$
L8. **end**
L9. $V_S \leftarrow \{block_k(v) \mid v \in V_G\}$,
$E_S \leftarrow \{(block_k(v), block_k(v')) \mid (v, v') \in E_G\}$

Method: V-Update(v)

V1. $fwsig(v) = \{(+m(v, v'), block_k(v')) \mid (v, v') \in E_G, m(v, v') \in M\}$
V2. $bwsig(v) = \{(-m(v', v), block_k(v')) \mid (v', v) \in E_G, m(v', v) \in M\}$
V3. $block_{k+1}(v) = hash(sort(block_k(v) \cup fwsig(v) \cup bwsig(v)))$

Figure 1: B variation algorithm

S variation computation of structural summary

L4′. **parallel foreach** $v \in V_G$ **do** V-Update-Singleton(v)

Method: V-Update-Singleton(v)

S1. **if** $|\{v' \mid v' \in block_k(v)\}| > 1$ **then do**
S2.   V-Update(v)
S3. **else do**
S4.   **disable vertex** v
S5. **end**

Figure 2: S variation algorithm

A node's (k+1)-dual block identifier is the result of a *hash* fuction that takes as input the forward and backward signatures A *sort* function takes the set and returns it as a lexicographically sorted set of signature entries and is used so that there is a canonical input to the hash function. In addition to using canonical input values, we assume that a hash value generated from a signature does not collide with a different signature, thus block identifiers generated from a hash value can be used as a canonical block identifier. Although it is not essential, we choose to reuse a node's block identifier as part of its hash computation as a way to maintain a node's 'history'.

Singleton optimization. We now describe variation S. Since it is common in graph algorithms to skip computations, GraphChi provides a scheduler that maps each node to a boolean value which decides whether to *disable* processing of specific nodes in each iteration. Using the scheduler has a benefit that GraphChi skips loading intervals of disabled nodes and their edges from disk, thereby improving performance by reducing disk access.

Our S variation adds an algorithmic optimization to variation B that reduces the number of nodes the algorithm processes in each iteration by skipping nodes that are in the extent of a singleton block. Notice that summary partitions have the property such that a node that is in a singleton block is never bisimilar with any other node. This means that a singleton node's block identifier will always remain distinct from all other nodes, and that the summary construction algorithm no longer needs to update the node's block identifier. In Figure 2, we give the S variation algorithm, which uses the program of variation B (Figure 1) but having instead the modified line L4′ that executes the custom node update function *V-Update-Singleton*. The algorithm keeps a hashmap with the count of nodes in each block's extent in the previous iteration. Each node executes the *V-Update-Singleton* method and uses the hashmap to check whether the node is in a singleton's extent (S1). If the node is not a singleton, then the method computes an updated block identifier for the node using the same *V-Update* function from the B variation (Figure 1). If a node is a singleton, then the node is not processed in the current iteration, and the keyword *disable* (S4) sets its boolean flag in the scheduler to *false* so as to skip the node in future iterations.

We implement a workaround to a GraphChi bug in order to skip singletons; the bug arises from the way GraphChi iterates over nodes it creates to balance parallelism across cores. We skip singletons as follows. During an iteration k, we count the number of nodes in each block. Then during the $(k + 1)$ iteration, when we execute each node's update method, we check whether the node is contained in the extent of a singleton block. If a node is found to be in a singleton, we do not update its block identifier. As well, during the $(k+1)$ iteration, we identify those nodes that are in singletons, then at the end of the $(k + 1)$ iteration, we disable them in the scheduler. Thus, singletons that exist at the end of an iteration k are skipped from iteration $(k + 2)$ onwards.

The correctness of our S variation follows from the correctness of the hash-based approach [3], which says that, assuming that hash values for non-bisimilar nodes do not collide, two nodes that have different signatures will have different block identifiers. In our approach, since a singleton's identifier will always remain distinct from any other block identifier (no other node can be bisimilar), then we do not have to update the block identifier of the node in its extent and can disable it in the scheduler.

The singleton optimization that we use in this work is novel. In particular, singleton blocks are not captured by the stable/unstable block optimization described in [3][1].

3. EVALUATION

In this section, we give an evaluation of our GraphChi B and S variations using 3 real-world datasets.

Table 1: Dataset graph statistics

Dataset	$\|N\|$	$\|E\|$
lmdb	1,327,120	6,147,717
dbp	48,603,466	317,220,816
Twitter	52,579,678	1,963,263,821

Table 1 shows the number of nodes and edges in the dataset graph of each dataset that we use in our experiments. LinkedMDB [9] describes entities related to movies including actors and directors, and is the smallest dataset graph (*lmdb*) we consider with 1.3M nodes, 6.1M edges, and 222 distinct edge labels. DBpedia describes entities such as places and events, and its dataset graph (*dbp*) has 48 million nodes, 317 million edges, and 1,393 distinct edge labels; we exclude highly structured entities such as geographic coordinates to reduce structuredness. Twitter is the largest dataset that we consider with 52M nodes and almost 2 *billion*

[1]Please see Appendix for a discussion.

Table 2: FWBW summary statistics, with M1 times (in mins) to load, construct (with B or S), and write

Summary	$\lvert N\rvert$	$\lvert E\rvert$	Load	B	S	Write
lmdb	844,877	4,311,098	0.12	8.6	2.8	0.75
dbp	32,274,111	278,182,230	107	775	187	207
Twitter	48,332,025	1,945,307,755		3,118	632	

edges unlabeled follower relationships between users, which is slightly bigger than the Twitter dataset [14] used in existing summary literature.

To compare FWBW summaries, we use *M1*, a single machine that has 8 Xeon X6550 2GHz 8-core CPUs and can process 64 nodes in parallel. M1 has an 80GB allocation of main-memory to the Java 8 JVM that executes unmodified GraphChi 0.2; our B variation uses around 40GB with the largest dataset and our S variation uses around 60GB with the largest dataset.

Table 2 shows the number of blocks and block edges in the FWBW summary of each dataset listed in Table 1. The lmdb summary, which has the greatest reduction in size with respect to the dataset graph (not counting extent edges) amongst the 3 datasets, has 0.84M blocks and 4.3M block edges, which represent 64% and 70% of the number of dataset graph nodes and edges, respectively. The Twitter summary has the least reduction in size with 48M blocks and only 18M fewer block edges than dataset graph edges, which represent over 90% and 99% of the number of dataset nodes and edges, respectively.

Table 2 also shows the M1 times (in mins) to load data into GraphChi, construct the summary using the *B* or *S* GraphChi variation, and write the summary to disk; the times shown are the average of 5 runs excluding the min and max. Constructing FWBW summaries using the S variation is x3.0, x4.1, and x4.9 times faster than the B variation for lmdb, dbp, and Twitter, respectively. Our results also show that the time taken to compute dbp's FWBW summary using the S variation is x1.68 times faster than the time to load the dataset graph and write the summary.

Figures 3 (a) and (b) show lmdb's FWBW summary construction for each iteration of the B and S variations; both variations take 13 iterations. Figure 3 (a) shows per-iteration time (in mins). In the B variation, per-iteration time is 12 seconds in iteration 1, increases steadily to 39s by iteration 5, and then maintains that time. In contrast, the S variation begins similarly but its per-iteration time decreases to around 7s by iteration 7, and then maintains that time. Our results show that the S variation's per-iteration time deviates to 25% of the B variation within the first few iterations.

Figure 3 (b) shows, the total number of blocks in the summary at each iteration, divided into singletons (*SG*) and non-singletons (*Non-SG*); the total is the same as the B variation. The total number of blocks increases to 837K blocks by iteration 5, and increases in small increments thereafter. No single iteration has more than 89K non-singletons and 762K singletons, which means that, after the first few iterations, up to 57.4% of the dataset graph's nodes are skipped in each iteration. As noted above, singletons that exist at the end of an iteration k are not skipped until iteration $k+2$ onwards. We observe that lmdb has 6.8K singletons at the end of iteration 1, which represents only 0.51% of the dataset graph's nodes, and that the S variation is already 15% faster than the B variation in iteration 3, the earliest moment in which the singleton optimization can skip nodes. The S variation has the greatest reduction in per-iteration time in iteration 6 after finding an almost maximal number of singletons in iteration 4. The stark reduction in per-iteration time demonstrates the benefit of skipping singletons as an optimization.

Figures 4 (a) and (b) show dbp's FWBW summary construction for each iteration of the B and S variations; both variations take 17 iterations. Figure 4 (a) shows the per-iteration time (in mins). As with lmdb, the B and S variations take equal number of iterations, iterations in variation B take longer than variation S, and iterations tend to take equal amounts of time. We attribute the difference in time in iteration 1, before skipping any singletons, due to a difference in how the algorithm counts blocks in the first iteration. Figure 4 (b) shows the total number of blocks, divided into singletons and non-singletons. There is a substantial increase in the number of singletons within the first few iterations, with slow growth after iteration 3, at which point the S variation's per-iteration time is 80% less than the B variation. A performance improvement is visible with variation S at iteration 3, the earliest moment in which the algorithm can skip singletons, and the per-iteration time is lowest at iteration 5 onwards after finding an almost maximal number of singletons, which, like lmdb, comprises of 57.3% of the dataset graph's nodes.

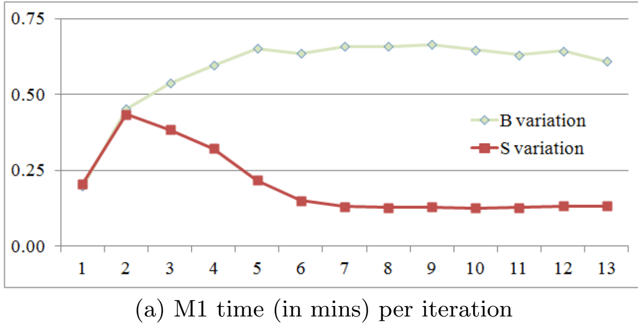
Figures 5 (a) and (b) show Twitter's FWBW summary construction for each iteration of the B and S variations; both variations take 13 iterations. Figure 5 (a) shows the per-iteration time (in mins). The figure shows that both the B and S variations start similarly then in the 5th iteration the B variation quickly takes almost an order of magnitude longer than the S variation, taking only around 4 minutes per iteration. Figure 5 (b) shows the total number of blocks, divided into singletons and non-singletons. There are only 3, 15, and 12K blocks in the first 3 iterations, respectively. No singletons exist in the first two iterations, then after only 2.4K singletons are found at the end of iteration 3, there is a remarkable reduction in per-iteration time to only 4 minutes in iteration 5. In later iterations, even as the number of singletons represent over 90.4% of the dataset graph's nodes, the per-iteration time remains constant.

In this section, we evaluate the construction of summaries for LMDB, DBpedia, and Twitter with our B and S variations. Our results show that our singleton optimization's rapid reduction in per-iteration time results in summary construction performance that is faster than loading the dataset into GraphChi and writing the summary to disk. Next, we compare GraphChi and Hadoop summary construction implementations.

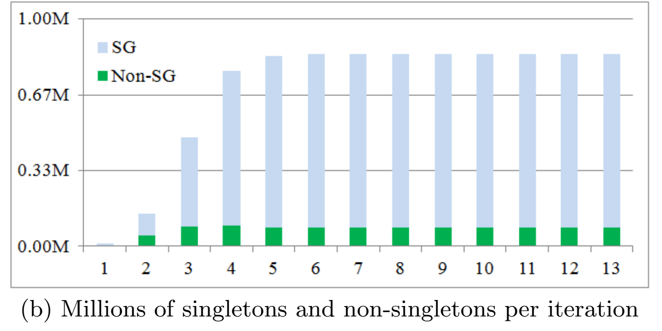
3.1 Hadoop Experimental Setup

In this section, we compare FW summary construction performance of our GraphChi variations with *HF*, our Hadoop 2.20 implementation based on [22]. We use *M2*, a single machine whose hardware consists of 2 Opteron6128 2GHz 8-core CPUs, which can process 16 threads in parallel, and 64GB of RAM. In our M2 experiments, we assign all of the RAM to the JVM but our GraphChi variations use about 40GB and HF uses about 20GB.

We run Hadoop in *local* mode, which uses one JVM on a single machine to execute each iteration of HF as a job consisting of parallel map tasks, but one reduce task. Since we do not require fault-tolerance, we store data as gzip-

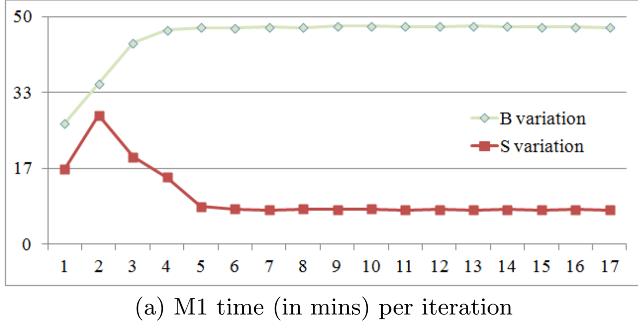


(a) M1 time (in mins) per iteration

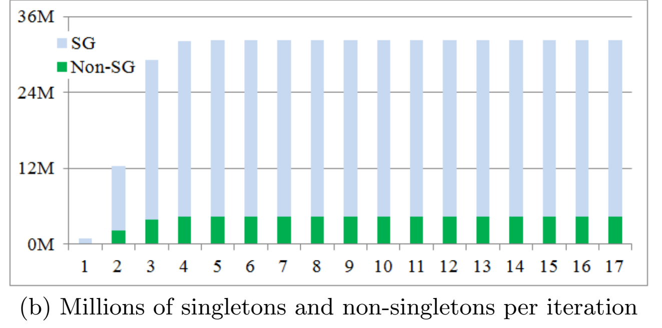


(b) Millions of singletons and non-singletons per iteration

Figure 3: LMDB FWBW summary construction

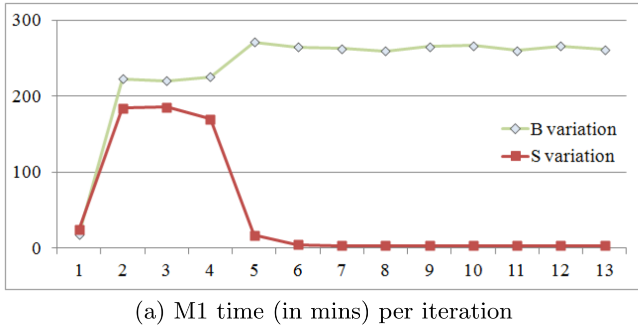


(a) M1 time (in mins) per iteration

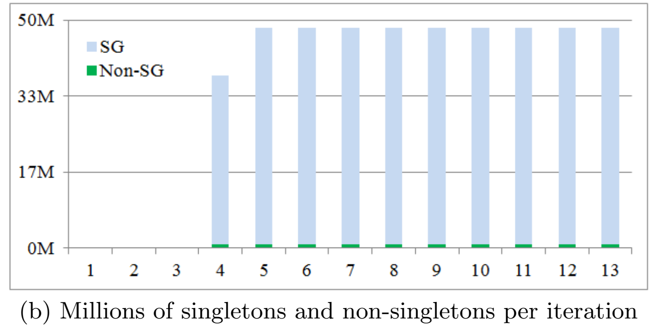


(b) Millions of singletons and non-singletons per iteration

Figure 4: DBpedia FWBW summary construction



(a) M1 time (in mins) per iteration



(b) Millions of singletons and non-singletons per iteration

Figure 5: Twitter FWBW summary construction

Table 3: FW summary statistics, with M2 construction times (in mins)

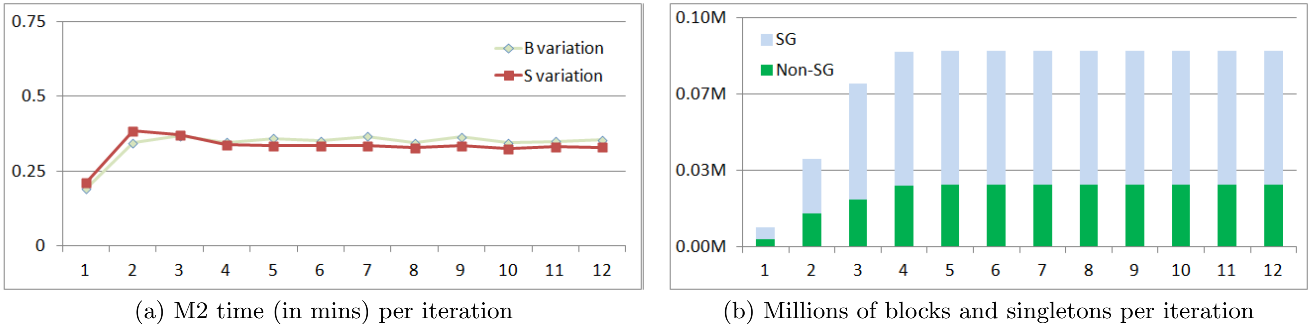
Summary	$\|N\|$	$\|E\|$	B	S	HF
lmdb	85,714	999,934	4.2	4.0	73
dbp	13,429,903	229,490,296	941	331	5,832

compressed files on the local filesystem. In local mode, the reduce task saves its output to a single file; however, map tasks cannot read gzip-compressed files in parallel. To enable parallelism, we save the reducer's output in multiple gzip-compressed files, each of which can then be accessed by one of the parallel map threads in the next iteration.

Table 3 shows FW summary statistics of lmdb and dbp, as well as M2 construction time (in mins) using our B, S, and HF variations. The table shows that each dataset's FW summary has fewer blocks and block edges than its FWBW summary; lmdb's FW summary contains almost 90% fewer blocks and 77% fewer block edges, while dbp's FW summary contains 58% fewer blocks and 18% fewer block edges. Our S variation constructs lmdb's summary only slightly faster than the B variation, but x18 times faster than HF. Our S variation constructs dbp's summary x2.8 times faster than the B variation, while HF, which we run for the first 10 iterations, needs around 9.7 hours per iteration.

Figures 6 (a) and (b) show lmdb's FW summary construction for each iteration of the B and S variations; both variations take 12 iterations. Figure 6 (a) shows per-iteration time (in mins). Both the B and S variations start similarly, increase to 20s by the next iteration, and remain at that duration for the remainder of the construction. Figure 6 (b) shows the total number of blocks in the FW summary at each iteration, divided into singletons and non-singletons. The number of blocks increases to around 85K blocks by iteration 4, with small increments thereafter. The FWBW summary of lmdb has an order of magnitude more singletons than in the FW summary, and our results show that the 59K singletons in lmdb's FW summary provide little discernible per-iteration decrease when using the S variation.

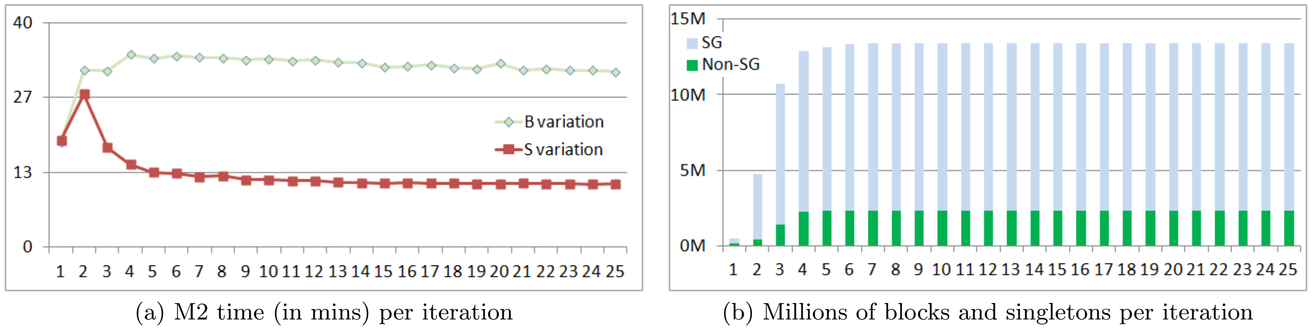
Figures 7 (a) and (b) show dbp's FW summary construction for each iteration of the B and S variations; both variations take 25 iterations. Figure 7 (a) shows per-iteration time (in mins). The per-iteration time of both variations start similarly, then the B variation increases to 32min while the S variation decreases to 12min. Figure 7 (b) shows the



(a) M2 time (in mins) per iteration

(b) Millions of blocks and singletons per iteration

Figure 6: LMDB FW summary construction



(a) M2 time (in mins) per iteration

(b) Millions of blocks and singletons per iteration

Figure 7: DBpedia FW summary construction

total number of blocks in the FW summary in each iteration, divided into the number of singletons and non-singletons. There are around 85K blocks by iteration 4, and it increases in small increments thereafter. Unlike the lmdb FW summary, 22% of the dataset graph's nodes are singletons, which helps to improve the S variation's per-iteration time.

We compare our work with [22], which we call MRB, whose DBpedia dataset is smaller than ours by 27M and 119M nodes and edges, respectively. MRB computes its FW summary, which contains 5 million blocks, in 459 minutes without writing the summary to disk. MRB's advantage is that it does not need to load any data, but has disadvantages in that it duplicates block identifiers as edge properties and processes all nodes in each iteration. Summaries that MRB computes are not as costly as our FWBW summaries (each iteration in a FW summary construction only looks at outgoing edges), and our S variation's iteration time is lower than MRB's. On Amazon EC2, a single multi-core compute instance has more compute power and is x2.5 times more economical than an MRB-equivalent 10-node cluster. Furthermore, our S variation requires much less parallelism, uses 40x less RAM, and has better performance even if a dataset needs more iterations.

Our S variation's per-iteration time to generate Twitter's FWBW structural summary is faster than the FW summary per-iteration time of [16, 17], whose dataset has 500M fewer edges than ours. Furthermore, their system focuses on optimizing the skew of edges across the few large blocks, while we focus on optimizing the many small block at the opposite spectrum of block sizes - where singletons are very common. The singleton optimization may be integrated into Hadoop-based summary constructions by sharing all non-singleton block identifiers with all parallel tasks using Hadoop's *distributed cache* since there are fewer than 5 million non-singletons in the FWBW or FW summaries that we construct.

4. CONCLUSIONS

In this work, we show that GraphChi on multi-core architecture is a viable way to construct summaries of big graphs. The performance of our variations show that skipping singletons provides performance improvement, with evidence to support its use on multi-core systems.

5. REFERENCES

[1] Big data: The next frontier for innovation, competition, and productivity. *McKinsey & Company*, May 2011.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to rethink? *SOCC*, 2013.

[3] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Electr. Notes Theor. Comput. Sci.*, 68(4):523–538, 2002.

[4] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Comput. Netw.*, 33(1-6):309–320, June 2000.

[5] M. P. Consens, V. Fionda, S. Khatchadourian, and G. Pirrò. Rewriting Queries over Summaries of Big Data Graphs. AMW, 2014.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.

[8] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[9] O. Hassanzadeh and M. P. Consens. Linked movie data base. In *LDOW*, 2009.

[10] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Foundations of Computer Science*, pages 453–462, 1995.

[11] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43 – 68, 1990.

[12] S. Khatchadourian and M. P. Consens. ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud. In *ESWC*, pages 272–287, 2010.

[13] A. Kumar, J. Gluck, A. Deshpande, and J. Lin. Hone: "Scaling Down" Hadoop on Shared-Memory Systems. *PVLDB*, 6(12):1354–1357, 2013.

[14] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[16] Y. Luo, G. H. L. Fletcher, J. Hidders, P. De Bra, and Y. Wu. Regularities and dynamics in bisimulation reductions of big graphs. GRADES, 2013.

[17] Y. Luo, Y. Lange, G. H. Fletcher, P. Bra, J. Hidders, and Y. Wu. Bisimulation Reduction of Big Graphs on MapReduce. In *BNCOD*, volume 7968, pages 189–203. 2013.

[18] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM Conference on Internet Measurement*, pages 29–42, 2007.

[19] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[20] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. Synopsys: large graph analytics in the SAP HANA database through summarization. In *GRADES*, 2013.

[21] D. Sangiorgi. On the origins of bisimulation and coinduction. *Trans. Program. Lang. Syst.*, 31(4), 2009.

[22] A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciel-Zablocki. Large-scale bisimulation of RDF graphs. SWIM, 2013.

[23] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

APPENDIX

A. UNSTABLE SINGLETONS

A block is *stable* if it does not have any block edges to a block that was split in the previous iteration (a block is split when two non-bisimilar nodes are found). We can easily construct examples that show a singleton that is stable, a singleton that is unstable, a singleton that is not found until an unstable block is stable, a singleton that is found by splitting an unstable block which remains unstable. We can also verify that singletons are distinct from the notion of stability because the singleton optimization is monotonic, in that once a block is a singleton it always remains a singleton, but stability is not monotonic since a block's state can switch between stable and unstable.

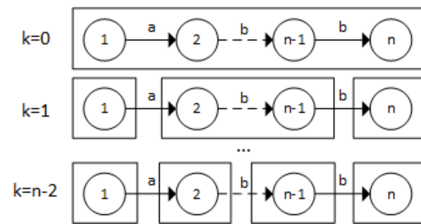


Figure 8: Unstable singleton example

Figure 8 shows an example graph whose FW summary contains a singleton that is unstable and which is found in early iterations (node 1), a singleton that is not found in an unstable block until the block is stable (node 2), and a singleton that is found in an unstable block before the block is stable (node n). The graph has n nodes, with node 1 having an outgoing edge with label *a* to node 2, and a path with edge label *b* from node 2 to node n. Before the iterations start, all nodes are in the same block. In the first iteration, nodes 1 and n are split into their own blocks. Node 1 is in a singleton because it is the only node with an out-edge with label *a*, and node n is in a singleton because it is the only node that does not have any out-edges. As well, $block_1(1)$ is unstable because $(block_1(1), block_0(2)) \in E_S$ but $block_0(2)$ was split, though $block_1(n)$ is stable because it does not have any outgoing edges. After n-2 iterations, we end up with a partition that only contains singletons.

B. UNSTABLE STABILITY

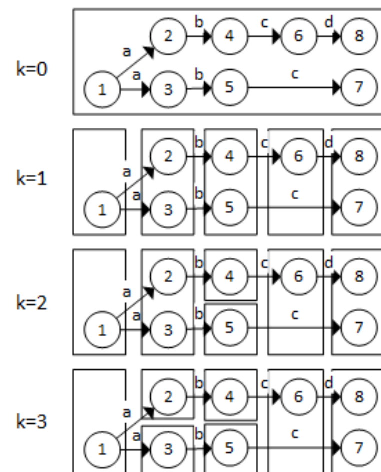


Figure 9: Unstable/stable/unstable example

Figure 9 gives an example where a singleton block can switch back and forth between stability and instability. At the start, all nodes are in the same block. In iteration 1, node 1 is split into a singleton since it is the only node that has out-edges with the label *a*; as well, nodes 2 and 3 form a block, nodes 4 and 5 form a block, node 6 forms a singleton block, and nodes 7 and 8 form a block. Notice that at the end of iteration 1, only $block_1(7)$ is stable since it does not have any outgoing edges. In iteration 2, since $(block_1(6), block_0(7)) \in E_S$ and $block_0(7)$ was split, then $block_1(6)$ is unstable and we need to process it. Since it is has only 1 node, there is nothing to split. In iteration 3, $block_1(4)$ is split such that $block_2(4) \neq block_2(5)$.