VRGQ: Evaluating a Stream of Iterative Graph Queries via Value Reuse

Xiaolin Jiang
xjian049@ucr.edu
Computer Science Department
Univ. of California Riverside

Chengshuo Xu
cxu009@ucr.edu
Computer Science Department
Univ. of California Riverside

Rajiv Gupta
gupta@cs.ucr.edu
Computer Science Department
Univ. of California Riverside

Abstract

While much of the research on graph analytics over large power-law graphs has focused on developing algorithms for evaluating a single global graph query, in practice we may be faced with a stream of queries. We observe that, due to their global nature, vertex specific graph queries present an opportunity for sharing work across queries. To take advantage of this opportunity, we have developed the VRGQ framework that accelerates the evaluation of a stream of queries via coarse-grained *value reuse*. In particular, the results of queries for a small set of source vertices are reused to speedup all future queries. We present a two step algorithm that in its first step initializes the query result based upon value reuse and then in the second step iteratively evaluates the query to convergence. The reused results for a small number of queries are held in a *reuse table*. Our experiments with best reuse configurations on four power law graphs and thousands of graph queries of five kinds yielded average speedups of 143×, 13.2×, 6.89×, 1.43×, and 1.18×.

1 Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs) to uncover insights by analyzing high volumes of connected data. Real world graphs are often large (e.g., Twitter - TT has 2 billion edges and 52.6 million vertices) and iterative graph analytics requires repeated passes over the graph till the algorithm converges to a stable solution. As a result, in practice, iterative graph analytics workloads are highly data- and compute-intensive. Therefore, there has been a great deal of interest in developing scalable and efficient graph analytics systems such as Pregel [10], GraphLab [9], PowerGraph [3], Galois [13], GraphChi [7], Ligra [15], ASPIRE [18, 19] and others.

While the performance of graph analytics has improved greatly due to advances introduced in aforementioned systems, much of this research has focussed on developing highly parallel algorithms for solving a single iterative graph analytic query. For example, SSSP(s) query computes shortest paths from a single source s to all other vertices in the graph. However, in practice the query evaluation system may need to respond to multiple queries for different source vertices. The queries may be generated by a single user or multiple users. In this work we develop a general framework, VRGQ, aimed at evaluating a stream of *vertex queries* received from users for different source vertices of a large graph. For example, for SSSP algorithm, we may be faced with the following stream of queries: SSSP(s_1); SSSP(s_2); ····· SSSP(s_n).

We observe that different queries typically traverse the majority of the graph and thus present an opportunity for *reuse* across multiple queries. For example, the same shortest subpaths may contribute to solutions of many queries and hence *reusable* across them. Our approach for reuse is as follows. Given an input graph and type of vertex query, we *precompute* the results of queries for a small number of source vertices and save them in a table for coarse-grained value reuse to optimize the evaluation of all future queries. Note that an iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When all vertex values become stable, the algorithm terminates. Our proposed *reuse strategy* is designed to update property values of all vertices in a single reuse step such that a good number of vertex values arrive at their final *stable* solutions and thus the active vertex sets of subsequent iterations are greatly reduced.

In the development of VRGQ we consider the following factors. First, because the reuse step incurs significant cost as it updates property values of *all* vertices, to limit its cost reuse is performed only once during the evaluation of a query by both the presented algorithms. Second, to maximize the benefits of reuse, reuse should be performed as early as possible and therefore we develop an algorithm that performs reuse right at the start. In particular, we have developed the 2Step algorithm where the Step 1 of the algorithm safely initializes the values of all vertices with the benefit of the precomputed results of other source vertex queries and then Step 2 simply iterates till the algorithm converges. Experiments with four power law graphs show that 2Step delivers varying amounts

of speedups across queries of different kinds. In particular, with best reuse configurations, for thousands of graph queries of five kinds 2Step yielded average speedups of 143×, 13.2×, 6.89×, 1.43×, and 1.18×.

The remainder of the paper is organized as follows. Section 2 develops the 2Step reuse based query evaluation algorithm that is at the heart of VRGQ. Section 3 presents the evaluation of the 2Step algorithm. Section 4 describes our approach for selecting reuse source vertices whose queries are evaluated to populate the reuse table. Additional related work is discussed in Section 5 and concluding remarks are given in Section 6.

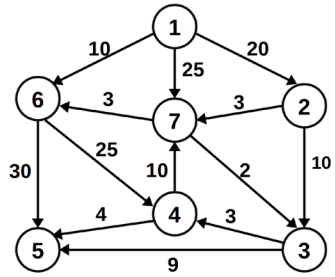
2 VRGQ: Two Step Value Reuse Algorithm

Let us assume that given a source vertex s, a query $Q(s)$ computes the desired property value *val* for every destination vertex d with respect to s, also denoted as $val(d \setminus s)$. A straightforward and intuitive approach for enabling reuse of the computed values is to explicitly transform the graph to express the computed property values so that they become available for reuse to future queries.

The above approach is illustrated by an example shown in Figure 1 using single-source shortest-path or SSSP queries. Figure 1(a) shows a small graph followed by the evaluation of query SSSP(2) using a push-style strictly synchronous algorithm. The shortest path values for all vertices are first initialized to 0 for source vertex 2 and ∞ for all other destination vertices. Following each iteration the updated shortest path values are shown along with list of active vertices, that is, vertices whose values have changed and thus must play a role in further propagation. Note that we do not add vertex 5 to the set of active vertices because it has no outgoing edges. As we can see, in all it takes four iterations for the shortest path values to converge.

Now let us assume that SSSP(7) had been previously evaluated and we would like to reuse its result to accelerate the computation of query SSSP(2). In Figure 1(b) we observe that the results of SSSP(7) indicate that $val(4 \setminus 7)$ is 5 and $val(5 \setminus 7)$ is 9. As shown, this information can be incorporated explicitly into the graph by introducing the two additional edges, or *shortcuts*, one from 7 to 4 and the other from 7 to 5 with weights of 5 and 9 respectively. Next the results of computing SSSP(2) on the transformed are shown. We observe that the shortcuts cause faster propagation and hence convergence is achieved in one less iteration than before. While in the above example there is a reduction in number of total iterations, even if the number of iterations remains the same, reuse can lead to reduction in total amount of work performed due to reduction in number of active vertices.

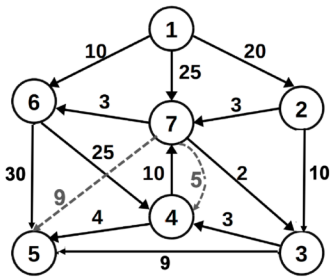
While the above approach is straightforward and applicable to other graph algorithms where query evaluation originates at a source vertex, it also has the following limitations. First, the approach based upon graph transformation may cause the shortcuts introduced to be processed multiple times increasing the overhead of reuse and thus cutting into its benefits. This limitation is illustrated in Figure 1(c) where evaluation of SSSP(1) on the transformed graph causes shortcuts to be



Iter#	1	2	3	4	5	6	7	Active Vertices
0	∞	0	∞	∞	∞	∞	∞	{2}
1	∞	0	10	∞	∞	∞	3	{3,7}
2	∞	0	5	13	19	6	3	{3,4,6}
3	∞	0	5	8	14	6	3	{4}
4	∞	0	5	8	12	6	3	{}

(a) Evaluation of SSSP(2) on the Original Graph.

Source	1	2	3	4	5	6	7
7	∞	∞	2	5	9	3	0



Iter#	1	2	3	4	5	6	7	Active Vertices
0	∞	0	∞	∞	∞	∞	∞	{2}
1	∞	0	10	∞	∞	∞	3	{3,7}
2	∞	0	5	8	12	6	3	{3,4,6}
3	∞	0	5	8	12	6	3	{}

(b) Evaluation of SSSP(2) on the Graph Transformed Via Shortcuts to Reuse Results of Query SSSP(7).

Iter#	1	2	3	4	5	6	7	Active Vertices
0	0	∞	∞	∞	∞	∞	∞	{1}
1	0	20	∞	∞	∞	10	25	{2,6,7}
2	0	20	**27**	**30**	**34**	10	23	{3,4,7}
3	0	20	**25**	**28**	**32**	10	23	{3,4}
4	0	20	25	28	32	10	23	{}

(c) Evaluation of SSSP(1) on Graph Transformed Via Shortcuts.

Figure 1: Value Reuse via Graph Transformation.

processed twice as vertex 7 is activated twice. At first activation $val(7 \setminus 1)$ is 25 which is not stable and during second activation $val(7 \setminus 1)$ is 23 which is the final stable value. Second, introduction of shortcuts increases the number of edges in the graph. Hence there is an increase in the size of memory footprint as well as number of irregular memory accesses.

To overcome the above limitations we developed the 2Step algorithm for evaluating a query $Q(s)$. At the start of the computation, the Step 1 of the algorithm initializes the values of vertices using the results for another query, say $Q(r)$. By performing reuse exactly once right at the start, the benefits of reuse are maximized and its overhead is minimized. Then, in Step 2 the algorithm iterates applying conventional updates to all vertices till the algorithm converges. Instead of transforming the graph by adding shortcuts, in this approach we store results of evaluating a small number of queries in a *reuse table* and select a suitable $Q(r)$ for reuse. By using a reuse table, the footprint of the graph is not increased and its locality of graph accesses is not worsened. Moreover accesses of values from the reuse table exhibit good spatial locality.

The reuse table is designed to contain both the forward and backward results of query $Q(r)$. This ensures that when $Q(s)$ reuses results of query $Q(r)$, it makes use of stable value of $val(r \setminus s)$ which maximizes the stable values produced via reuse. When reuse table contains results of multiple queries for different source vertices, to limit the cost of reuse 2Step selectively reuses results of a small subset of *most promising* source vertices in the reuse table. Thus, 2Step, while controlling the cost of reuse, maximizes production of stable values.

Forward-Backward Reuse Table Our objective is to perform reuse first, i.e. apply reuse updates and then run the original iterative algorithm to completion. Moreover, we want to ensure maximal production of stable values. We observe that both these objectives can be met if we enhance the information contained in the reuse table such that it contains both *forward* and *backward* information for a given reuse vertex r as described below:

- FWDTABLE [**r**] [d] ($\forall\, d \in$ ALLVERTICES) represents the property values (e.g., shortest path) computed for query Q(**r**) on graph *Graph*.
- BWDTABLE [s] [**r**] ($\forall\, s \in$ ALLVERTICES) represents the property values (e.g., shortest path) computed for query Q(**r**) on *edge-reversed* graph $Graph^R$ – which is obtained by reversing the direction of each edge in the original *Graph*.

Now let us see how the precomputed values can be used right away at the start of evaluation of query Q (**v**). Given a vertex d, its value can be initialized using the precomputed results of a reuse source vertex r in the table as follows:

$$\text{REUSEFUNC}\ (d, \text{BWDTABLE}[\mathbf{v}][r], \text{FWDTABLE}[r][d]\ )$$

The REUSEFUNC function for five algorithms is given in Table 1. For example, for SSSP,

$$\text{BWDTABLE}[\mathbf{v}][r] + \text{FWDTABLE}[r][d]$$

is the shortest path from **v** to d via r and thus it is the best estimate we can obtain for $d.value$ by using the results for vertex r in the reuse table. If we reuse results for multiple r vertices in the reuse table then we find the shortest paths via each of these vertices and then take the minimum across all the computed shortest paths to initialize $d.value$. *Because the* BWDTABLE *contains stable values, reuse produces maximal number of stable values upon completion of the reuse step.*

Let us reconsider the computation of SSSP(1) whose evaluation was shown earlier in Figure 1(c). Now let us apply 2Step to this query; however, now the reuse table contains both forward and backward information for vertex 7 as in Figure 2. The evaluation now involves reuse followed by iterations. Following reuse, only two iterations are required for termination. In contrast earlier it took an extra iteration.

Source		1	2	3	4	5	6	7
7	FWD	∞	∞	2	5	9	3	0
	BWD	23	3	13	10	∞	35	0

Iter#		1	2	3	4	5	6	7	Active
Init.		0	∞	∞	∞	∞	∞	∞	{1}
Reuse		$d \in \{3,4,5,6,7\}$ BwdTable[7][1] + FwdTable[7][d]							
		0	∞	25	28	32	26	23	{1}
P	1	0	20	25	28	32	10	23	{2,6}
	2	0	20	25	28	32	10	23	{}

Figure 2: Computing SSSP(1) via 2Step Algorithm by Reusing results of SSSP(7).

Push-style 2Step Reuse Algorithm In Algorithm 1 we summarize our 2Step algorithm. As we can see, in Step 1 (lines 4-9) of evaluating the query for a given source vertex s, we exploit the precomputed results for REUSABLEVERTICES whose queries were precomputed and used to populate the reuse table (BWDTABLE, FWDTABLE). The algorithm first carries out the reuse step and sets the values of all destination vertices by reusing n most promising reusable source vertices for which precomputed query results are stored in the reuse table. To find the n most promising vertices we use function PRIORITIZE which looks at values in the BWDTABLE and prioritizes vertices in the increasing or decreasing order (depending upon algorithm characteristic) of BWDTABLE[r][s] values where r is a reusable source vertex whose results were precomputed and stored in the reuse table and s is the source vertex for which the query $Q(s)$ is being evaluated. The details of function REUSE show how it uses REUSEFUNC (lines 32-35) to perform reuse updates of all the vertex values so

Algorithm 1 Backward-Forward Reuse Algorithm.

```
1:  function 2STEP ( Q ( s ), BWDTABLE, FWDTABLE, n )
2:      ▷ Initialize ACTIVE Vertex Set and Vertex Values
3:      ACTIVE ← INITIALIZE ( Q ( s ) )
4:      ▷ Step 1: Reuse
5:      Q ← PRIORITIZE( s )
6:      ▷ Select reuse vertices and perform Reuse
7:      for all r ∈ first n vertices in Q do
8:          REUSE ( s, r )
9:      end for
10:     ▷ Step 2: Iterate
11:     while ACTIVE ≠ ϕ do
12:         ACTIVE ← PROCESS ( ACTIVE )
13:     end while
14: end function
15:
16: function PRIORTIZE ( s )
17:     ▷ Ordering Vertices Used to Populate Reuse Table
18:     Build Priority Queue Q by inserting all
19:     vertices r ∈ REUSABLEVERTICES such that
20:     they are sorted in increasing order of
21:     BWDTABLE[r][s] values.
22:     return Q
23: end function
24:
25: function REUSE ( s, r )
26:     ▷ Only Reuse Valid Values
27:     if BWDTABLE[s][r] ≠ initialValue then
28:         for d ∈ ALLVERTICES do
29:             ▷ Only Reuse Valid Values
30:             if FWDTABLE[r][d] ≠ initialValue then
31:                 ▷ Perform Reuse Update of d
32:                 REUSEFUNC ( d,
33:                             BWDTABLE[s][r],
34:                             FWDTABLE[r][d])
35:                         )
36:             end if
37:         end for
38:     end if
39: end function
40:
41: function PROCESS ( ACTIVE )
42:     NEWACTIVE ← ϕ
43:     for all v ∈ ACTIVE do
44:         for each e ∈ Graph.outEdges(v) do
45:             ▷ Apply Conventional Update to e.dest
46:             changed ← EDGEFUNC (e)
47:             if changed then
48:                 ▷ Update NEWACTIVE Set
49:                 NEWACTIVE ← NEWACTIVE ∪ {e.dest}
50:             end if
51:         end forall
52:     end forall
53:     return NEWACTIVE
54: end function
```

that remainder of the iterative algorithm does not have to start from the initialization values of all vertices but rather better values computed by the reuse step. Once the reuse step has been completed, then the iterative computation is completed (see Step 2, lines 10-13) by applying conventional updates to all vertices using the Process function (lines 41-54).

Table 1: Functions for Reuse Updates for Four Algorithms.

$d.value \leftarrow$ VRFUNC (d, BWDTABLE$[s][r]$, FWDTABLE$[r][d]$)
SSWP(s): $d.value \leftarrow max(d.value, min(\text{BWDTABLE}[s][r], \text{FWDTABLE}[r][d]))$
Viterbi(s): $d.value \leftarrow max(d.value, \text{BWDTABLE}[s][r] * \text{FWDTABLE}[r][d])$
SSSP(s): $d.value \leftarrow min(d.value, \text{BWDTABLE}[s][r] + \text{FWDTABLE}[r][d])$
BFS(s): $d.value \leftarrow min(d.value, \text{BWDTABLE}[s][r] + \text{FWDTABLE}[r][d])$
SSNP(s): $d.value \leftarrow min(d.value, max(\text{BWDTABLE}[s][r], \text{FWDTABLE}[r][d]))$

3 Experimental Evaluation of VRGQ

Next we evaluate VRGQ that is based upon the presented 2Step algorithm and report the speedups achieved, reduction in number of active edges processed, and extent to which stable values are produced.

We implemented our algorithms using Ligra [15] that provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. Graph algorithms used include – Single Source Widest Path (SSWP), Viterbi [8], Single Source Shortest Path (SSSP), Breadth First Search (BFS), and Single Source Narrowest Path (SSNP). Experiments were performed on a 64 core (8 sockets × 8 cores) machine with AMD Opteron 2.3 GHz processor 6376, 512 GB memory, and running CentOS Linux release 7.4.1708.

Table 2: Input graphs used in experiments.

Graphs	#Edges	#Vertices
Twitter (TT) [2]	2.0B	52.6M
Twitter (TTW) [6]	1.5B	41.7M
LiveJournal (LJ) [1]	69M	4.8M
PokeC (PK) [16]	31M	1.6M

We use four directed and edge-weighted power-law input graphs with relatively small diameter that are listed in Table 2 – TT, TTW, LJ, PK. We use the default weight generation tool provided by Ligra. Ligra generated weights range from 1 to the $log(n)+1$ (where, $n = |vertices|$). For the four graphs we tested, $log(n)$ ranged from 20 to 25. We also varied the upper bound of the range to 64 and 128, but the results were similar.

Table 3: Coverage Characteristics of Reuse Table with 5 REUSABLEVERTICES.

	H_{min}	H_{max}	ALL → REUSABLE	#Queries
TT	1	4	81.9% → 81.7%	20K (5K/hop)
TTW	1	4	96.1% → 95.8%	20K (5K/hop)
LJ	2	5	87.2% → 83.2%	20K (5K/hop)
PK	2	5	86.8% → 85.1%	20K (5K/hop)

Table 4: NoReuse: Average execution times in *Seconds* across 20,000 queries.

G	SSWP	Viterbi	SSSP	BFS	SSNP
TT	7.04s	9.28s	8.41s	0.42s	18.03s
TTW	3.10s	3.91s	3.62s	0.28s	11.02s
LJ	0.21s	0.29s	0.22s	0.04s	0.58s
PK	0.07s	0.14s	0.09s	0.02s	0.09s

Table 5: 2Step: Speedups for three Reuse Table configurations: 5 out of 20; 2 out of 10; and 1 out of 5. Overall the 1 out of 5 configuration performs the best in 3 of 5 benchmarks as it minimizes the reuse overhead.

G	Hops	SSWP	Viterbi	SSSP	BFS	SSNP
TT	1	66.5 : 155.9 : 249.0	9.51 : 10.2 : 10.5	1.97 : 1.85 : 1.72	2.12 : 2.20 : 2.14	16.2 : 17.3 : 18.0
	2	52.1 : 125.7 : 266.2	6.61 : 6.82 : 6.96	1.57 : 1.48 : 1.40	1.15 : 1.22 : 1.24	16.1 : 15.3 : 16.1
	3	48.2 : 106.5 : 205.1	3.81 : 3.94 : 3.90	1.49 : 1.42 : 1.35	0.95 : 1.38 : 1.27	16.0 : 14.8 : 15.5
	4	43.9 : 95.9 : 199.9	3.20 : 3.24 : 3.28	1.49 : 1.41 : 1.36	0.95 : 0.95 : 1.01	11.8 : 12.1 : 12.4
TTW	1	82.4 : 94.6 : 157.6	9.61 : 9.48 : 10.9	1.83 : 1.75 : 1.67	1.69 : 2.01 : 2.10	14.4 : 15.3 : 15.7
	2	71.0 : 69.3 : 125.6	6.83 : 7.44 : 7.34	1.39 : 1.33 : 1.30	0.99 : 1.14 : 1.03	14.8 : 14.2 : 15.2
	3	71.0 : 79.0 : 125.3	4.47 : 3.97 : 4.82	1.38 : 1.35 : 1.26	0.87 : 1.00 : 0.94	12.4 : 15.3 : 14.5
	4	47.7 : 51.7 : 111.6	3.01 : 3.76 : 3.86	1.32 : 1.27 : 1.21	0.83 : 0.93 : 0.97	11.8 : 13.9 : 13.0
LJ	2	26.2 : 72.7 : 122.2	8.50 : 10.9 : 10.4	1.14 : 1.13 : 1.13	0.96 : 1.04 : 1.09	14.8 : 17.6 : 18.7
	3	23.2 : 65.9 : 119.6	5.20 : 6.46 : 6.51	1.12 : 1.13 : 1.14	0.88 : 0.90 : 0.97	11.9 : 14.1 : 15.9
	4	25.2 : 57.3 : 105.2	4.53 : 5.71 : 4.80	1.12 : 1.14 : 1.14	0.84 : 0.91 : 0.94	9.34 : 13.3 : 14.2
	5	17.8 : 46.1 : 78.7	4.00 : 4.25 : 4.72	1.17 : 1.15 : 1.11	0.82 : 0.90 : 0.95	6.75 : 8.62 : 9.62
PK	2	38.0 : 89.3 : 138.5	11.5 : 15.6 : 14.5	1.47 : 1.51 : 1.43	1.09 : 1.17 : 1.17	11.3 : 12.6 : 6.23
	3	31.2 : 75.6 : 120.2	6.83 : 8.04 : 8.27	1.37 : 1.29 : 1.32	0.95 : 1.02 : 1.04	4.91 : 5.47 : 5.57
	4	23.0 : 57.1 : 88.0	5.19 : 4.89 : 5.29	1.50 : 1.48 : 1.37	0.90 : 0.97 : 1.00	8.16 : 9.03 : 5.12
	5	20.5 : 52.0 : 78.5	4.05 : 4.52 : 4.19	1.48 : 1.42 : 1.39	0.88 : 0.93 : 0.97	7.40 : 8.03 : 4.28
Average		43.0× : 80.9× : 143.2×	6.05× : 6.82× : 6.89×	1.43× : 1.38× : 1.33×	1.05× : 1.17× : 1.18×	12.0× : 13.2× : 12.8×

For each input graph, we generated 20,000 queries. Table 3 characterizes the coverage of queries and reuse table that are used in our evaluation. The queries used were for source vertices that are H_{min} to H_{max} hops from the reuse source vertices in the reuse table. The reuse table populated with results of only 5 source vertices allows nearly all possible queries to take advantage of its contents for reuse (ALL ranging from 81.9% to 96.1%). The hops considered account for nearly all of these queries (REUSABLE ranging from 81.7% to 95.8%). For each input graph we used 20,000 queries spread across the four different hop values considered. For 80% of vertices, Hops ranged from 1 to 4 or 2 to 5 for four graphs. We randomly picked 5K corresponding to each hop. Thus, the selected queries maximize diversity in terms of number of hops between the source vertex and the reusable vertices.

Finally, we use multiple reuse table configurations of the form *n out of m* in these experiments, where *m* is the number of sources vertices whose full results are stored in the reuse table and *n* is the most promising subset number of these that are actually exploited during reuse. The configurations used are 5 *out of* 20; 2 *out of* 10; and 1 *out of* 5.

Speedups Table 5 presents the speedups achieved by 2Step for the three reuse table configurations for all the input graphs and algorithms considered. The baseline running times without reuse (NoReuse) that are used in computing speedups are given in Table 4. For power law graphs, the results for 20,000 queries are separated according to the 5,000 queries each for the four hop values considered. The average speedups obtained for power-law graphs and table configurations are substantial, 43× to 143× for SSWP, 6.05× to 6.89× for Viterbi, 1.33× to 1.43× for SSSP, 1.05× to 1.18× for BFS, and 12× to 13.2× for SSNP.

The larger power-law graphs (TT and TTW) experience higher speedups than smaller power-law graphs (LJ and PK). The number of hops from query source vertex to nearest reuse table source vertex also impacts performance. By and large, the smallest hop distance gives the best speedups while the largest hop distance gives the least speedup. For example, for SSWP on TTW with the smallest reuse table configuration, speedups decrease from 157.6× to 111.6× as hop distance increases from 1 to 4. For SSWP the smallest table configuration gives best speedups, for Viterbi and BFS the speedups for the two smaller table configurations are fairly close and significantly better than for the largest table configuration, and for SSSP mostly the largest table gives best speedups, though by a small margin. Larger reuse tables may enable

Table 6: 2Step: % reduction in processed active edges due to reuse for table configuration of 2 *out of* 10.

G	Hops	SSWP	Viterbi	SSSP	BFS	SSNP
TT	1	99.99	95.28	73.14	29.94	99.99
	2	99.99	99.99	82.80	18.89	99.99
	3	99.99	89.30	66.84	2.21	99.99
	4	99.99	85.60	71.31	3.89	99.99
TTW	1	99.99	94.42	68.76	34.06	99.99
	2	99.99	99.99	80.48	38.09	99.99
	3	99.99	89.67	62.05	2.11	99.99
	4	99.99	91.44	70.93	1.67	99.99
LJ	2	99.99	96.66	58.53	5.86	99.99
	3	99.99	86.89	50.03	3.04	99.99
	4	99.99	94.86	46.84	0.41	99.99
	5	99.99	93.94	49.38	0.34	99.99
PK	2	99.99	98.03	74.59	6.78	99.99
	3	99.99	89.56	70.93	9.71	99.99
	4	99.99	93.21	69.54	1.10	99.99
	5	99.99	91.36	71.88	0.77	99.99
Average		99.99%	93.14%	66.75%	9.93%	99.99%

Table 7: 2Step: Percentage of values that become stable following reuse for table configuration of 2 *out of* 10.

G	Hops	SSWP	Viterbi	SSSP	BFS	SSNP
TT	1	99.99	90.57	41.37	93.04	99.99
	2	99.99	85.81	21.58	60.28	99.99
	3	99.99	74.70	22.64	42.91	99.99
	4	99.99	62.81	28.88	47.31	99.99
TTW	1	99.99	90.79	48.76	94.78	99.99
	2	99.99	89.38	31.81	72.83	99.99
	3	99.99	80.31	31.26	52.97	99.99
	4	99.99	79.10	30.86	47.83	99.99
LJ	2	99.99	91.72	21.08	46.24	99.99
	3	99.99	84.69	16.61	23.65	99.99
	4	99.99	84.72	11.94	16.52	99.99
	5	99.99	79.25	13.39	15.69	99.99
PK	2	99.99	92.99	46.34	66.88	99.99
	3	99.99	84.14	41.52	39.41	99.99
	4	99.99	71.65	33.97	32.76	99.99
	5	99.99	61.77	36.24	27.59	99.99
Average		99.99%	81.52%	29.89%	48.79%	99.99%

more effective reuse but also incur higher reuse overhead. Thus, depending upon the algorithm characteristics, different sized reuse tables deliver the best performance for different benchmarks.

Reduction in Active Edges and Stable Values Produced Next we present additional data for the benchmarks to better understand the large degree of difference in speedups observed. Tables 6 and 7 provides reduction in active edges processed and extent to which reuse step produces stable values. The reason for varying degrees of speedups can be found in algorithms characteristics.

First, performance benefit of reuse is very high for SSWP because in this algorithm no new values are computed – the resulting property value for each vertex is essentially equal to the weight of a selected edge. Thus, reuse often produces stable values as high as 99.99% as shown in Table 7. This unique characteristic of SSWP resulted in high speedups.

The nature of Viterbi and SSSP is quite similar as both compute new values, except that one involves real values and the other integer values. However, on average, Viterbi produces 81.52% stable values while SSSP produces only 29.89% stable values on average across all four power law graphs. Thus, average reduction in active edges is 93.14% for Viterbi which is significantly higher than 66.75% for SSSP across all power law graphs. This explains why Viterbi achieves higher speedups than SSSP via reuse even though the values were not stable following reuse.

Small benefits are expected for BFS as cost of computing a value for a vertex is similar to cost of generating the value via reuse. The competing factors of possibly increased work (when insufficient number of stable values are produced) and better memory behavior during reuse (because edge-lists are not accessed and vertices are visited in the order they are stored) result in small speedups or small slowdowns.

Speedups for Individual Queries So far we have presented the *average* execution times and speedups over 20,000 queries for the 2 *out of* 10 configuration. To demonstrate that the speedups are achieved across nearly *all* queries, we present plots in Figure 3 where for the TT graph, the execution times of 2Step and NoReuse for each query are plotted. The plots are given for the SSWP and BFS algorithms that give maximum and minimum average speedups across all benchmarks. The scatter plots in Figure 3 show that execution times for nearly all of the 20K queries are improved by **VRGQ** (the only exception is Hops=4 queries for BFS).

4 Populating the Reuse Table

To populate the reuse table we need to identify a small subset of vertices in the graph, say N, whose backward and forward query results will be precomputed and stored in the reuse table. Since an input graph typically contains millions of vertices, we need to develop a methodology for selecting the N vertex set. Our selection is aimed at achieving two goals:

- *Maximize Reuse* – Since all vertices are not equally effective in the degree of reuse they support, we will include vertices in N that have high *centrality*. That is, we will give preference to vertices that play an influential role in computation of results of large number of other queries.

- *High Coverage* – Ideally we would like to choose N such that, from all other vertices in the graph, at least some vertices in N can be reached in less than MaxHops. That

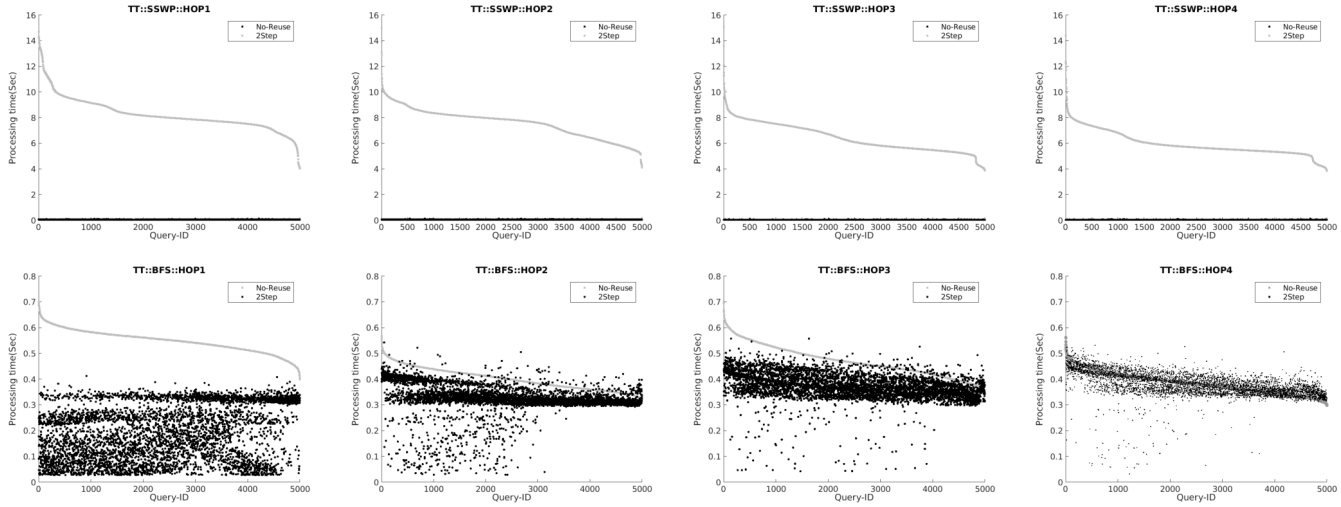


Figure 3: Performance of individual queries – each scatter plot depicts execution times of 2Step and NoReuse for 20,000 queries – black dots correspond to 2Step execution times and gray dots are for NoReuse execution times. The plots show that vast majority of queries benefit from 2Step algorithm.

Table 8: Populating Reuse Table: Overlap in 20 source vertices selected; Runtime costs of using different number of random queries to select 20 vertices; and Runtime overhead of populating the Reuse Table with full results for the 20 vertices.

G		Number of Random Queries					Populating Reuse Table
		1000	**10**	**20**	**40**	**60**	
TT	**Overlap**	20	19	20	20	20	
	Overhead	6070s (3.8%)	101s (3.1%)	175s (5.1%)	333s (6.2%)	393s (4.7%)	699s
TTW	**Overlap**	20	19	18	19	19	
	Overhead	3095s (5.7%)	60s (7.9%)	89s (1.5%)	155s (6.4%)	227s (6.8%)	490s
LJ	**Overlap**	20	20	18	18	20	
	Overhead	164s (14.1%)	3.1s (7.3%)	5s (9.7%)	7.4s (6.7%)	13.6s (4.4%)	43s
PK	**Overlap**	20	17	19	19	19	
	Overhead	81s (8.5%)	1.3s (0.3%)	2.5s (10.3%)	4.2s (10.2%)	5.6s (7.8%)	16s

is, all queries will encounter reusable vertices. Since this is not always possible, we attempt to achieve *high coverage*, i.e. for most queries reuse is possible.

Our three step methodology for selecting vertex set N such that it meets the above goals is presented below.

Step 1: Identify High Centrality Candidates using Random Queries – We evaluate randomly selected queries and during each query evaluation we maintain *impact counts* for all other vertices in the graph. Each time the value of some vertex v, causes an update of its out-neighbor, the impact count of v is incremented. Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between pairs of nodes. Vertices that have a high probability to occur on a randomly chosen shortest path between two randomly chosen vertices have a high betweenness. High *impact counts* are treated as an indicator for high centrality. Thus, extremely expensive centrality is avoided. For each query evaluated, the top N vertices, i.e. vertices with the highest impact counts, are identified. Since this process is repeated for multiple queries, say $q_1, q_2 \cdots q_n$, we obtain multiple candidate sets $\mathsf{N}(q_1), \mathsf{N}(q_2) \cdots \mathsf{N}(q_n)$.

Step 2: Selecting top | N | High Centrality Candidates – Note that the same vertex may appear in multiple sets $\mathsf{N}(q_1)$, $\mathsf{N}(q_2) \cdots \mathsf{N}(q_n)$ obtained in the first step but not necessarily in all the sets. We select the final set N by both considering how frequently a vertex appears in different sets and its corresponding impact counts. This is achieved by summing up the impact counts of all occurrences of a vertex in the above sets and sorting the final sums to identify the top | **N** | vertices.

Step 3: Populating the Table – The top centrality candidates identified in the preceding step are processed one by one to populate the reuse table. By evaluating the query for each vertex v in both the forward direction on the original graph and backward direction on the edge reversed graph, the reuse table is populated with FWDTABLE$[v][*]$ and BWDTABLE$[*][v]$.

Table 9: Improvements if instead of using 5 *out of* 20 we reuse results of all 20 queries. The minimal improvements indicate that reusing results of a few queries is enough.

Improvements in	SSWP	Viterbi	SSSP	BFS
Vertices Processed	0%	0%	3%	5%
Stable Values	0%	0%	1%	7%

Table 10: Substantial overlap in high centrality vertices for different benchmarks and graphs for **20** REUSABLE-VERTICES indicates that same reuse vertices can be used across different benchmarks.

	TT	TTW	LJ	PK
SSWP	20	20	20	20
Viterbi	20	19	18	19
SSSP	20	19	17	16
BFS	19	18	19	16

Table 11: Higher Hub populating times in seconds and their comparison with VRGQ populating times.

Graph→	TT	TTW	LJ	PK
Hub=20	1102s 1.58×	732s 1.49×	43s 1.00×	16s 1.00×
Hub=100	5925s 8.50×	3390s 6.92×	211s 4.90×	77s 4.81×

Table 12: Evaluation of Hub and VRGQ applying reuse on 4K BFS queries with size of **20**. The results show that Hub-based reuse largely results in slowdowns.

Speedups		Reuse 1	Reuse 2	Reuse All
Hub=20	**PK**	0.89×	0.75×	0.35×
	TT	0.98×	0.96×	0.75×
Hub=100	**PK**	0.81×	0.74×	0.12 ×
	TT	1.01×	0.95×	0.31 ×
VRGQ	**PK**	1.06×		
	TT	1.39×		

Cost of populating strategy Finally we would like to describe the runtime cost of constructing the reuse table of size 20 – we present this cost when using different number of random queries – 1000, 10, 20, 40 and 60 – in Table 8. Considering the 20 vertices found using 1000 random queries as the best selection, we compared its results with sources vertices selected by using 10, 20, 40 and 60 random queries. The overlap among the selected vertices is very high – when using 60 random queries, the overlap is 20 out of 20 for TT and LJ and it is 19 out of 20 for TTW and PK. Thus, running 60 random queries is more than sufficient for identifying high centrality vertices. The runtime overheads for selecting 20 vertices using 60 random queries range for 5.6 to 393 seconds depending upon the size of the power law graph graph while populating the table with results takes 16 to 699 seconds. Note that while high centrality vertices are being selected, new queries can be processed in parallel. The cost of selecting high centrality vertices varies from 0.44s (for PK) to 18.47s (for TT) for power law graphs for 60 random queries in Table 8. Thus, the overhead of populating the reuse tables is modest as it has to be done only once.

In Table 9 we show results of our experiment justifying the selection of 5 *out of* 20 configuration. If in the reuse step, instead of reusing results of best 5 we reuse results of all 20, little to no reductions in vertices processed and increase in stable values produced are observed as shown in the table. We also increased the size of the reuse table from 20 to 40 source vertices. No improvements in vertices processed or stable values produced were observed for the first three programs and 1% improvement was seen for BFS.

Heuristic effectiveness Our heuristic for finding high centrality vertices is very effective. To verify this we found high centrality vertices using the betweenness centrality algorithm available in Ligra and compared them with ones found by our heuristic. We found that top 5 vertices found by the original algorithm appear among the top 10 vertices found by our heuristic. Thus, there is great degree of overlap, though vertices appear in different order.

Finally, we identified the top 20 vertices in the four power law graphs by considering the centrality with respect to property values of all four algorithms and found that vast majority of vertices are the same for all algorithms. Table 10 shows how many of the 20 reuse sources found for SSWP also appear in the 20 reuse sources found for Viterbi, SSSP and BFS. The high overlap indicates that structure of the graph is the determinative factor much more so than the graph algorithm. Thus, for a given graph, it is possible to find reuse vertices once and use them for all algorithms.

Hub Accelerator vs. VRGQ The Hub Accelerator [5] is used to speedup the evaluation of graph queries. At a high level it is similar to VRGQ as both rely upon precomputations and then use the results of precomputations to speedup subsequent query evaluations. However, VRGQ computes point-to-all (i.e., queries involving single source and all destinations) while the Hub Accelerator computes point-to-point queries (i.e., queries involving a single source and destination pair). We implemented the Hub Accelerator in Ligra and adapted it to compute point-to-all queries by reusing the Hub Accelerator results for all destinations. Next we show that VRGQ is far more effective than Hub Accelerator both because its precomputation is relatively inexpensive and the speedups obtained are higher.

The reason why the Hub Accelerator precomputation is more expensive than our precomputation for populating the reuse table is as follows. For the Hub network to be effective it is typically chosen to have large number of vertices

(e.g., in [5] the authors consider Hub sizes of 5K to 15K vertices) and then shortest paths among all these vertices are precomputed. In addition, the shortest path from each non-Hub vertex to the closest core-Hub vertex must also be precomputed. In contrast, the 2Step algorithm of VRGQ only precomputes results for a handful of high centrality vertices to populate the reuse table. Table 11 compares the precomputation costs of Hub Accelerator precomputation with reuse table precomputation. Table 11 first presents Hub Accelerator precomputation times in seconds and then the factor by which the cost of Hub Accelerator precomputation exceeds that of reuse table precomputation of VRGQ. We present this data for Hub sizes of 20 and 100 for all four graphs. We observe that Hub Accelerator precomputation is several times more expensive that VRGQ precomputation for Hub size of 100 and even for Hub size of 20 for the difference can be substantial for large graphs.

We also compare speedups for evaluating four thousand BFS queries by applying reuse with Hub Accelerator and VRGQ in Table 12. These four thousand queries were chosen from 20K queries used in earlier experiments, we randomly chose 1000 queries each from the 5000 queries for each of the four hop values. With Hub size of 20, the same as the number of reusable-vertices for VRGQ, the Hub Accelerator only experiences slowdowns as shown in Table 12. If we reuse all precomputed results, due to high reuse cost, we get significant slowdown. We therefore tried reuse of only 1 or 2 vertices from the Hub but this too did not produce any speedups although it reduced the slowdowns. When Hub size was increased to 100, for the TT graph, a slight speedup of one percent was finally achieved though the precomputation cost is dramatically increased. In contrast VRGQ achieves average speedup of 1.39× for TT on BFS queries. To be consistent, in VRGQ, we reused only one source vertex selected from 20 queries. For Hub Accelerator [5], each vertex can have multiple *corehubs*. The result in Table 12 shows that if reuse all corehubs we only achieve slowdowns. If we select a specific corehub vertex by distance, then VRGQ and Hub Accelerator will select the same vertex for reuse. Hence there is no advantage of using a selected corehub over VRGQ. Moreover, Hub Accelerator was designed for point-to-point queries and hence its reuse table contains partial results. As a result, the reuse performed is costly and ineffective.

From the above results we conclude that the limitation of the Hub Accelerator approach is that it cannot deliver speedups for point-to-all queries. When small Hub sizes are used only slowdows are observed and when large Hub sizes are used the precomputation cost becomes very high. In contrast, VRGQ gives speedups with small number of reuseable-vertices while incurring low precomputation cost. Finally our 2Step algorithm requires minimal change to the original algorithms as it simply initializes the vertex values using reuse-table (not default initialization values) and then standard iterative algorithm is run.

5 Other Related Work on Optimizing Evaluation of Multiple Graph Queries

VRGQ reuses results of a few queries for achieving speedups in the evaluation of numerous other queries. We have adapted this concept and applied it in other scenarios as well. SimGQ [21] reuses the results of dynamically identified *sharing queries* to speed up the simultaneous evaluation of a batch of queries. Tripoline [4] uses this idea to create a generalized streaming graph model where continuously updated results of a small number of standing queries are used to rapidly evaluate arbitrary user queries.

There are two recent works, Quegel [23] and PnP [20], that evaluate a stream of graph queries. However, both these works are aimed at evaluating point-to-point queries (e.g., shortest path from a single source to a single destination). Quegel derives improved throughput by evaluating queries in a pipelined fashion and taking advantage of the Hub [5] precomputation. PnP [20] is similar to other graph frameworks in that speedups are achieved by evaluating a single query faster using new dynamic optimizations.

There are other recent works [11, 14, 17, 22–24] that evaluate multiple queries simultaneously. Congra [14] handles each query independently using a separate process. Thus, it is unable to take advantage of value reuse opportunity across queries. CGraph [24] and Seraph [22] are out-of-core systems that evaluate multiple queries. However, their benefits are achieved via sharing of the graph as opposed to the result values. In [17] a specialized system that processes multiple BFS queries is presented. Finally, MultiLyra [11] and BEAD [12] simultaneously evaluate queries on distributed systems.

6 Conclusion

We have developed VRGQ that incorporates the 2Step iterative algorithm for evaluating a sequence of graph queries on individual vertices. The key feature of this framework is that during evaluation of any query, coarse-grained reuse of previously computed results for selected vertices is performed to accelerate query evaluation. The VRGQ system is different from related works in a key way. It takes advantages of results computed for a very small number of queries to optimize the execution of all future queries via coarse-grained reuse.

Acknowledgments

This work is supported by National Science Foundation Grants CCF-2002554, CCF-2028714, and CCF-1813173 to the University of California Riverside.

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD), pages 44-54, 2006.

[2] M. Cha, H. Haddadi, F. Benevenuto, and P.K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *AAAI Conference on Web and Social Media*, 10(10-17):30, 2010.

[3] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 17-30, 2012.

[4] X. Jiang, C. Xu, X. Yin, Z. Zhao, and R. Gupta. Tripoline: Generalized incremental graph processing via graph triangle inequality. In *European Conference on Computer Systems* (EuroSys), pages 1-16, April 2021.

[5] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. CoRR, abs/1305.0507, 2013.

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

[7] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi : Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 31-46, 2012.

[8] J. Lember, D. Gasbarra, A. Koloydenko, and K. Kuljus. Estimation of Viterbi path in bayesian hidden markov models. In *METRON*, 77(2):137-169, 2019.

[9] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB Endowment*, 5(8):716-727, 2012.

[10] G. Malewicz, M.H. Austern, A.J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data* (SIGMOD), pages 135-146, 2010.

[11] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable distributed evaluation of batches of iterative graph queries. In *IEEE International Conference on Big Data* (BigData), pages 349-358, 2019.

[12] A. Mazloumi, C. Xu, Z. Zhao, and R. Gupta. BEAD: Batched evaluation of iterative graph-queries with evolving analytics demands. In *IEEE International Conference on Big Data* (BigData), pages 461-468, 2020.

[13] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles* (SOSP), pages 456-471, 2013.

[14] P. Pan and C. Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *IEEE International Conference on Computer Design* (ICCD), pages 217-224, 2017.

[15] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 135-146, 2013.

[16] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pages 1-6, 2012.

[17] M. Then, M. Kaufmann, F. Chirigati, T-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H.T. Vo. The more the merrier: Efficient multi-source graph traversal. In *Proceedings of the VLDB Endowment*, 8(4):449-460, 2015.

[18] K. Vora, S-C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *SIGPLAN International Conf. on Object Oriented Programming Systems, Languages and Applications* (OOPSLA), pages 861-878, 2014.

[19] K. Vora, C. Tian, R. Gupta, and Z. Hu. CoRAL: Confined recovery in distributed asynchronous graph processing. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 223-236, 2017.

[20] C. Xu, K. Vora, and R. Gupta. PnP: Pruning and prediction for point-to-point iterative graph analytics. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 587-600, 2019.

[21] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. SimGQ: Simultaneously evaluating iterative graph queries. In *IEEE International Conference on High Performance Computing, Data, and Analytics* (HiPC), pages 1-10, 2020.

[22] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai Seraph: an efficient, low-cost system for concurrent graph processing. In *ACM Symposium on High-Performance Parallel and Distributed Computing* (HPDC), pages 227-238, 2014.

[23] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng and W. Ng. A general-purpose query-centric framework for querying big graphs. In *Proceedings of the VLDB Endowment*, 9(7):564-575, 2016.

[24] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu. Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing. In *USENIX Annual Technical Conference* (USENIX ATC), pages 441-452, 2018.