BEAD: Batched Evaluation of Iterative Graph Queries with Evolving Analytics Demands

Abbas Mazloumi, Chengshuo Xu, Zhijia Zhao and Rajiv Gupta
Computer Science & Engineering, Univ. of California Riverside

***Abstract*—Simultaneous evaluating a batch of iterative graph queries on a distributed system enables amortization of high communication and computation costs across multiple queries. As demonstrated by our prior work on MultiLyra [BigData'19], batched graph query processing can deliver significant speedups and scale up to batch sizes of hundreds of queries.**

In this paper, we greatly expand the applicable scenarios for batching by developing BEAD, a system that supports Batching in the presence of Evolving Analytics Demands. First, BEAD allows the graph data set to evolve (grow) over time, more vertices (e.g., users) and edges (e.g., interactions) are added. In addition, as the graph data set evolves, BEAD also allows the user to add more queries of interests to the query batch to accommodate new user demands. The key to the superior efficiency offered by BEAD lies in a series of incremental evaluation techniques that leverage the results of prior request to "fast-foward" the evaluation of the current request.

We performed experiments comparing batching in BEAD with batching in MultiLyra for multiple input graphs and algorithms. Experiments demonstrate that BEAD's batched evaluation of 256 queries, following graph changes that add up to 100K edges to a billion edge Twitter graph and also query changes of up to 32 new queries, outperforms MultiLyra's batched evaluation by factors of up to 26.16× and 5.66× respectively.

***Index Terms*—Distributed Graph Processing, Query Batching, Evolving Graphs, Evolving Query Batch.**

I. Introduction

Graph analytics is employed in many domains (e.g., social networks [9], web graphs, etc.) to gain insights from large data graphs. Since real-world graphs are often large, there has been a growing interest in developing scalable graph processing systems that exploit parallelism on scalable distributed platforms (e.g., Pregel [11], GraphLab [10], GraphX [6], PowerGraph [5], PowerLyra [3], ASPIRE [17], [18]).

In a recent paper [12], we presented MultiLyra, a scalable graph system that simultaneously evaluates a batch of hundreds of iterative graph queries. It achieves high performance via aggregated communication strategies and query status tracking policies, both of which amortize the communication and computation overhead across queries. The idea of batching is motivated by the practical scenario [20], where online shopping platforms frequently compute a set of *interesting graph queries* involving the most important shoppers over a large input graph that represents the online shopping behaviors of all the customers. To meet the needs, Yan and others proposed Quegel [20], which allows for overlapped execution of a small set of queries. By contrast, our previously proposed system MultiLyra expanded the batching capability to simultaneously evaluating hundreds of iterative queries.

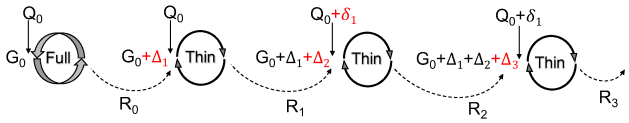
Despite the promises of batched graph query processing, the existing systems target a *static* scenario where the input graph is fixed and the queries of interests are pre-defined. However, in many real-world scenarios, both the graph and the batch of queries may evolve. For example, in the scenario of online shopping, as the system is being used, new customers may join continuously and the graph representing the online shopping activities will also continue to grow. Consequently, there is a need to regularly reevaluate the batch of interesting queries as the graph grows in size and changes in its structure. In addition, as the set of customers increases, a need arises to also grow the set of interesting queries to account for newly identified important customers. In other words, in the real-world situation of continuous activity, the system is faced with Evolving Analytics Demands. While using MultiLyra [12] or Quegel [20] for such evolving demands are possible by *fully* reevaluating the updated batch of queries on the updated graph, they may incur significant latency that keeps growing as the graph expands and more interesting queries are identified.

In this paper, we present BEAD – a Batched graph query processing system for Evolving Analytics Demands. Unlike MultiLyra and Quegel, BEAD leverages a set of incremental evaluation techniques to minimize the cost of reevaluation in the presence of graph growth and/or query additions.

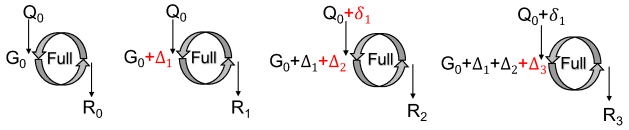
Let $Eval(G, Q) \rightarrow R$ denote the evaluation of a batch of queries Q on graph G with results R, a basic functionality of MultiLyra. BEAD generalizes the capabilities of MultiLyra by efficiently handling evolving analytics demands. Assume the initial graph is G_0, the initial query batch is Q_0, and their evaluation $Eval(G_0, Q_0)$ yields results R_0. Then, the graph grows with a set of additions Δ, which may include new edges and/or vertices. Instead of fully reevaluating Q_0 on the new graph $G_0 + \Delta$ as in MultiLyra, BEAD exploits prior results R_0 to incrementally evaluate $G_0 + \Delta$, denoted as $Inc(G_0{+}\Delta, Q_0, R_0)$. In addition to the growing graph $G_0{+}\Delta$, new queries of interests δ are also added to the query batch occasionally, that is, $Q_0 + \delta$. Even in this scenario, BEAD can still manage to leverage prior results R_0 to streamline the full evaluation $Eval(G_0 + \Delta, Q_0 + \delta)$ to an *incremental* evaluation $Inc(G_0 + \Delta, Q_0 + \delta, R_0)$, without compromising the correctness of the results.

Figure 1 illustrates how a sequence of requests are handled by BEAD and how they can be evaluated, though inefficiently, using MultiLyra. In this example, after the initial evaluation of queries on the original graph, first reevaluation is required due to changes in graph (Δ_1), then due to changes in both graph

978-1-7281-6251-5/20/$31.00 ©2020 IEEE



(a) BEAD: Efficient Incremental Evaluations.



(b) MultiLyra: Independent Full Evaluations.

Fig. 1. BEAD vs. MultiLyra: Evaluating sequence (G_0, Q_0), $(G_0+\Delta_1, Q_0)$, $(G_0+\Delta_1+\Delta_2, Q_0+\delta_1)$, $(G_0+\Delta_1+\Delta_2+\Delta_3, Q_0+\delta_1)$, for results R_0, R_1, R_2, and R_3.

and query batch (Δ_2 and δ_1), and finally due to changes in the graph (Δ_3). In Figure 1(a), BEAD efficiently evaluates the queries by taking advantage of the query results from the prior evaluation, while in Figure 1(b) MultiLyra evaluates queries on the corresponding graphs independently from scratch.

Furthermore, in the presence of new user request while the old is still being processed (i.e., user interruption), instead of waiting for the previous request to finish, BEAD permits *anytime evaluation* of the new request $Eval(G_0+\Delta, Q_0+\delta)$ using the unconverged results from $Eval(G_0, Q_0)$, denoted as $\approx R_0$. That is, before the convergence of Q_0 on G_0, user may interrupt the evaluation, make additions to the graph and the query batch, and carry out new evaluation incrementally.

We have developed a prototype of BEAD that builds upon the MultiLyra prototype and compared their performance on multiple input graphs and multiple kinds of graph queries. Experiments demonstrate that BEAD's batched evaluation of 256 queries, following graph changes that add up to 100K edges to a billion edge Twitter graph and also query changes of up to 32 new queries, outperforms MultiLyra's batched evaluation by factors of up to 26.16× and 5.66×, respectively.

II. Related Work

Most relevant works include Quegel [20], MultiLyra [12], and SimGQ [16]. First, these three systems do not provide the capabilities supported by BEAD. Since SimGQ is for shared-memory, it has limited scalability. Although Quegel evaluates multiple queries, its employment of expensive indexing prevents it from accepting graph changes which can invalidate the existing indexing results. Also, Quegel's applicability is limited to point-to-point queries [19] as opposed to the more computation-intensive point-to-all queries targeted by BEAD, MultiLyra, and SimGQ. Also note that the query batching in BEAD is different from query evaluation over streaming graphs [13], [14]. First, BEAD performs *continuous evaluation* of a *batch of queries*, that is, following updates, the queries must be reevaluated. In contrast, in streaming graphs, individual queries are evaluated (not batches) and they are evaluated only upon request (not continuously). Finally, [15] evaluates a batch of queries but it is specialized for BFS and [4] executes different queries in different processes making it inefficient in comparison to BEAD, MultiLyra and SimGQ.

III. Background: Batching in MultiLyra

Distributed platforms offer scalable processing of large graphs by partitioning them across machines. MultiLyra [12] adopts the hybrid-cut graph partitioning strategy first introduced by PowerLyra [3]. For low-degree vertices, it distributes the vertices along with their edges evenly among machines (i.e., edge-cut). For high-degree vertices, it distributes their edges evenly among machines (i.e., vertex-cut) to better balance the workload. When a high-degree vertex is partitioned and replicated across multiple machines, one of the replicas is selected as the *master* and the rest become the *mirrors*.

MultiLyra follows the GAS (**G**ather-**A**pply-**S**catter) model first introduced in PowerGraph [5] to perform BSP-style iterative graph computations. For simultaneously processing a batch of queries, like n SSSP queries: {SSSP(v_1), SSSP(v_2), ···, SSSP(v_n)}, MultiLyra maintains a *unified active list* such that a vertex is *active* if it is active for at least one of the queries in the batch. For a single active vertex, it integrates the processing of all queries in the batch to amortize the overhead across queries. Algorithm 1 summarizes the iterative algorithm – Line 3 initializes the `unified_activeList` for a given batch of queries, Lines 4-10 initialize the activeness statuses of queries based on the selected tracking mode (discussed later), vertex values are initialized by Lines 12-14, and Lines 15-21 show the main loop of iterative graph processing. Next we summarize the basic steps of batched evaluation (S_1 to S_5):

– S_1*: Exch-Batch.* In this step, all the local mirrors that were scheduled to be active for the current iteration send an *activation message* to their masters that reside on the remote machines, so that the master would be informed to become active. Note that one single *activation message* is sufficient per local mirror to cover all the queries.

– S_2*: Recv-Batch.* All the masters that are either informed by their local neighbors or through an *activation message* are added to the `unified_activeList`. After that, each of

Algorithm 1 Batching in MultiLyra – The GAS model.

```
1: function EVAL(G, Q, mode)
2:     ▷ Initialize the unified list with source vertices of n queries
3:     unified_activeList ← ⟨v1, v2, ..., vn⟩
4:     if mode == IQT then
5:         ▷ Si is a bitset indicating active queries for vertex i
6:         q_status ← ⟨S1, S2, ..., SN⟩ where Si.set_bit(i)
7:     else                                              ▷ mode = FQT
8:         ▷ si is a bit indicating if query i is still unfinished
9:         q_status ← ⟨s1, s2, ..., sn⟩ where si = 1
10:    end if
11:    ▷ Initialize the vertex values
12:    for each vertex v ∈ G do
13:        R[v][] ← INIT_VAL
14:    end for
15:    while !unified_activeList.empty() do
16:        Exch_Batch(q_status)                          ▷ S1
17:        unified_activeList, q_status ← Recv_Batch()   ▷ S2
18:        Gather_Batch(q_status, mode)                  ▷ S3
19:        Apply_Batch(q_status, mode)                   ▷ S4
20:        Scatter_Batch(q_status)                       ▷ S5
21:    end while
22:    return R
23: end function
```

them sends one single *activation message* to their mirrors to inform them participating the gather phase.

So far, the first two steps have made an consensus on which vertices are active in the current iteration. Based on this, the next three steps perform the GAS operations.

– S_3*: Gather-Batch.* All the active vertices, including both mirrors and masters, on each machine collect data along their incoming edges for all the queries in the batch. In addition, the mirrors send their portion of the locally gathered data to their masters, via a *data message*, so that the masters are aware of all the data they need globally for the *Apply-Batch*.

– S_4*: Apply-Batch.* Based on the collected data from the last step, values of all vertices (except mirrors) are first updated (depending on the graph applications) for all the queries. Then, to maintain the consistency across machines, when a vertex value is updated by at least one of the queries, the vertex values of all queries are sent to their mirrors in one aggregated *data message* to reflect the updates. Along with this message, one single *activation message* is also sent to the mirrors for the following-up scattering step.

– S_5*: Scatter-Batch.* All active vertices (including mirrors and masters) whose values have changed at least for one of the queries during S_4 inform (schedule) their out-neighbors for processing in the next iteration, which may include both local masters and local mirrors of remote vertices.

The above five steps form the *basic* version of MultiLyra. Given an active vertex, it performs integrated processing of all queries in each GAS phase, thus amortizing the overhead.

The *basic* version maintains a unified active list, which does not distinguish the active vertices among queries and is unaware the completion of queries. This leads to wasteful computations and communications. To improve the efficiency, MultiLyra provides two optimized ways of tracking the status of queries and vertices: *Finished Query Tracking* - FQT tracks if any of the queries in the batch has been finished; and *Inactive Query Tracking* - IQT tracks which queries need to be evaluated in one iteration for each vertex. They can skip the vertex evaluation for finished or inactive queries, respectively.

$$\textbf{V.data[]} = [\mathbf{d_{q_1}}, \mathbf{d_{q_2}}, \mathbf{d_{q_3}}, \mathbf{d_{q_4}}, \mathbf{d_{q_5}}, \mathbf{d_{q_6}}, \mathbf{d_{q_7}}, \mathbf{d_{q_8}}]$$

$$\textbf{(a) FQT} \begin{cases} \textbf{Q\_Status}: \ \langle U, F, U, U, U, F, U, U\rangle \\ \textbf{Data\_Msg}: [\mathbf{d_{q_1}}, \mathbf{d_{q_3}}, \mathbf{d_{q_4}}, \mathbf{d_{q_5}}, \mathbf{d_{q_7}}, \mathbf{d_{q_8}}] \end{cases}$$

$$\textbf{(b) IQT} \begin{cases} \textbf{Q\_Status[v]}: \langle I_v, F, I_v, U, I_v, F, I_v, U\rangle \\ \textbf{Data\_Msg}: \ [\mathbf{d_{q_4}}, \mathbf{d_{q_8}}] + \mathbf{00010001} \end{cases}$$

Fig. 2. *Data Message Compression.* F indicates the query has finished, U indicates the query has not finished and is still running, and I_v indicates that query is inactive for active vertex v in the current iteration.

Moreover, these two fine-grained tracking methods enable MultiLyra to support compressed data messages, as shown in Figure 2. These strategies use one bitset or array of bitsets, for *FQT* or *IQT* respectively, to hold the statuses of queries in the running batch (Algorithm 1 lines 4-10) and keeping it updated along with the active list in *Recv_Batch*.

In general, *IQT* works better for large batches of queries in the scenario where a high ratio of queries have not finished but are inactive for many vertices. So the performance benefits of *IQT* can easily eclipse its tracking overhead. *FQT*, on the other hand, works more efficiently for the small batch sizes.

IV. BEAD: Supporting Batching for Evolving Analytics Demands

Next we introduce BEAD which generalizes MultiLyra and adapts its batching strategies to scenarios of *evolving analytics demands* – growing graphs and batch of queries.

A. Batching with Graph and Query Updates

As the graph grows by Δ, it becomes natural that the user wants to expand the query batch with new interesting queries, denoted as δ. Given the full evaluation of query batch Q_0 on graph G_0: $Eval(G_0, Q_0) \rightarrow R_0$, this subsection shows how BEAD handles expansion of the graph and/or the query batch, that is, $Inc(G_0 + \Delta, Q_0 + \delta, R_0) \rightarrow R_1$.

BEAD carries out the incremental evaluation for the new query batch $Q_0 + \delta$ on the evolved graph $G_0 + \Delta$, by using the existing result R_0. The key lies in the suitable initialization of the vertex values, query tracking bitsets, and the list of active vertices. The incremental evaluation can be either *simultaneous* or *ordered*. Each policy has its own advantages. Next, we discuss the policies supported by BEAD in detail, then present an integrated algorithm that can be run under different modes and employ different policies.

Policy I: Simultaneous Evaluation – In this scenario, both changes in graph and queries (i.e., Δ and δ) are considered simultaneously by BEAD. That is, BEAD evaluates the new queries δ alongside the old queries Q_0 in one evaluation as a larger batch $Q_0 + \delta$ on the larger graph $G_0 + \Delta$, in a way that the existing R_0 can be leveraged. The rationale for simultaneous evaluation is that considering all changes together may lead to fewer iterations.

Policy II: Ordered Evaluation – In comparison, this option considers Δ and δ in an order: either Δ-first or δ-first. In the case of Δ-first, BEAD first computes the results of Q_0 on $G_0 + \Delta$ till convergence, then evaluates $Q_0 + \delta$ on $G_0 + \Delta$ till convergence. By contrast, δ-first ordering does the opposite. The rationale for using ordered approaches is that, in the case where Δ is much more significant than δ, computing them separately allows *IQT* and *FQT* to be used for handling Δ and δ, respectively.

Algorithm 2 describes the incremental evaluation of BEAD that directly supports Policy I, but it can be used to emulate and thus support Policy II as well (shown later).

Expanding Data Structures – The number of values stored at each vertex is grown by the number of new queries (Line 20), the same happens to the *q_status* array (Line 28).

Initialization for the New Queries – All the vertex values corresponding to the new queries are set to the initial value INIT_VAL (Line 21-22). After that, the source vertices of all the new queries are also added to the *unified_activList* to start the evaluation of new queries (Line 29). In addition, if IQT

Algorithm 2 BEAD: Reevaluating for Both Graph and Query Updates ($G_0 + \Delta$, $Q_0 + \delta$).

```
1: function INC(G_0, Q_0, R_0, Δ, δ, mode)
2:     G_1 ← G_0.update(Δ)                          ▷ Add new edges and vertices
3:     Q_1 ← Q_0.update(δ)                          ▷ Add new queries
4:     R_1 ← Initialize(G_1, Q_1, R_0, Δ, δ, mode)
5:     V_old ← Source vertices of the top three high out-degree ∈ Q_0
6:     while !unified_activeList.empty() do
7:         ▷ ComputeIteration() performs steps S_1 through S_4
8:         ComputeIteration(q_status, mode)
9:         if δ ≠ φ then
10:            Update(V_old, R_1, δ, R_0)
11:        end if
12:        Scatter_Batch(q_status)                  ▷ S_5
13:    end while
14:    return R_1
15: end function
16: function INITIALIZE(G_1, Q_1, R_0, Δ, δ, mode)
17:    ▷ Initialize R_1
18:    R_1.set_size(G_1.num_vertices())
19:    for each vertex v ∈ G_1 and each q ∈ Q_1 do
20:        R_1[v].set_size(Q_1.size())
21:        if v ∈ Δ or q ∈ δ then                   ▷ v or q are new
22:            R_1[v][q] ← INIT_VAL
23:        else                                     ▷ Otherwise initialize R_1 using R_0
24:            R_1[v][q] ← R_0[v][q]
25:        end if
26:    end for
27:    ▷ Initialize unified_activeList with all the affected vertices
28:    q_status.add_bitset_size_by(δ.size())
29:    unified_activeList ← source vertices v_q of all q ∈ δ
30:    if mode == IQT then
31:        for each q ∈ δ do q_status[v_q].set_bit(q)
32:    end if
33:    for each e ∈ Δ do
34:        v ← e.dest()
35:        unified_activeList ← unified_activeList ∪ v
36:        if mode == IQT then
37:            q_status[v].set_all()
38:        end if
39:    end for
40:    if mode == FQT then
41:        q_status.set_all();
42:    end if
43:    return R_1
44: end function
45: function UPDATE(V_old, R_1, δ, R_0)
46:    if any v_old ∈ V_old has been activated by any q ∈ δ then
47:        ▷ Send the current data of v_h to other machines
48:        ClusterSynced(R_1[v_old][])
49:        ▷ Update current value of all vertices for queries in δ
50:        ▷ Using equations in Table I
51:        R_1 ← UpdateUsingR_0(R_1, δ, R_0, v_old)
52:    end if
53: end function
```

TABLE I
EQUATIONS USED IN UPDATEUSINGR_0() (ALGORITHM 2 - LINE 51) TO UPDATE VERTICES FOR ANY NEW QUERY $q \in \delta$ WHICH REACHES THE SOURCE VERTEX v_{old} OF A QUERY q_{old} IN Q_0.

Algo.	Update Equation
SSSP	$R_1[v][q]$=Min($R_1[v][q]$, $R_1[v_{old}][q]$+$R_0[v][q_{old}]$)
SSWP	$R_1[v][q]$=Max($R_1[v][q]$, Min($R_1[v_{old}][q]$, $R_0[v][q_{old}]$))
Viterbi	$R_1[v][q]$=Max($R_1[v][q]$, $R_1[v_{old}][q] \times Ro[v][q_{old}]$)
BFS	$R_1[v][q]$=Min($R_1[v][q]$, $R_1[v_{old}][q]$+$R_0[v][q_{old}]$)

is selected, it sets the corresponding bit for each new query – initializing the statuses of new queries to be active.

Enabling Indirect Incremental Computations – In addition, BEAD manages to take advantage of old results R_0 to achieve faster convergence for the new queries in δ. As shown in Algorithm 2 (Line 9-11), a new step, called *Update*, is inserted to the main loop right before *Scatter-Batch*, which updates all vertex data of new queries using the equations from Table I for faster convergence. In specific, one old query $q_{old} \in Q_0$ is selected and its results are used by the update equations to improve all the vertex values of a new query q. However, to apply the update equations, the source vertex of q_{old} should be reachable from the source vertex of q – *the source vertex of q_{old} should be activated by the new query in the current iteration* (Line 46). One intuitive heuristic for selecting the old query q_{old} is selecting the one with the highest out-degree source vertex who is more likely to be reached by a new query. In our case, BEAD selects three queries whose source vertices have the top three out-degrees (Line 5). Finally, before apply the update equations (Line 51), the results of the selected old query need to be synchronized across machines (Line 48).

Now we present how Algorithm 2 can be used to emulate different policies: simultaneous evaluation (i.e., $\Delta||\delta$); Δ-first evaluation (i.e., $\Delta \rightarrow \delta$); and δ-first evaluation (i.e., $\delta \rightarrow \Delta$).

$\Delta||\delta$

EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_0) \rightarrow R$
By Calling Algorithm3::Inc($G_0, Q_0, R_0, \Delta, \delta, IQT$)

$\Delta \rightarrow \delta$

EVALUATE $Inc(G_0 + \Delta, Q_0, R_0) \rightarrow R_1$
By Calling Algorithm3::Inc($G_0, Q_0, R_0, \Delta, \phi, IQT$)
EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_1) \rightarrow R$
By Calling Algorithm3::Inc($G_1, Q_0, R_1, \phi, \delta, FQT$)

$\delta \rightarrow \Delta$

EVALUATE $Inc(G_0, Q_0 + \delta, R_0) \rightarrow R_1$
By Calling Algorithm3::Inc($G_0, Q_0, R_0, \phi, \delta, FQT$)
EVALUATE $Inc(G_0 + \Delta, Q_0 + \delta, R_1) \rightarrow R$
By Calling Algorithm3::Inc($G_0, Q_1, R_1, \Delta, \phi, IQT$)

As shown above, by feeding an empty set ϕ alternatively to Algorithm 2, both ordered evaluations can be realized. Note that we do not list FQT for $\Delta||\delta$, as it does not perform as well as IQT, for the reasons we mentioned earlier.

B. Interruption Handling

Finally, we consider the situation in which the user presents a new request while the prior is still being processed. One way to handle this interruption is waiting for the old request (say Δ_1 and δ_1) to converge then starting the processing of the new request (say Δ_2 and δ_2), which we referred to as *following convergence*. Instead of waiting for the old request to complete, BEAD chooses to merge the processing of both the old and new requests, such that the total processing time could be reduced. We refer to this more proactive option as *anytime interruption*. Algorithm 3 describes how anytime interruption works in the presence of a new user request while the old request is being processed. Right after the new request interruption is received, BEAD combines the new request with

the old meanwhile leveraging the current intermediate results of the old request (i.e., $\approx R_1$).

Algorithm 3 BEAD: Reevaluating for Anytime Simultaneous Update ($G_0 + \Delta$,$Q_0 + \delta$).

```
1: function InterruptHandling(G_1, Q_1, ≈R_1, mode)
2:     Δ', δ' ← UserInterrupt.get_new_request()
3:     G_1 ← G_1.update(Δ')                    ▷ Add new edges and vertices
4:     Q_1 ← Q_1.update(δ')                    ▷ Add new queries
5:     ▷ Reinitialize R_1 based on prior unfinished results ≈R_1
6:     R_1 ← Initialize(G_1, Q_1, ≈R_1, Δ', δ', mode)
7:     return R_1
8: end function
```

V. Experiments

Experimental Setup - We developed BEAD by integrating the implementations of incremental evaluation algorithms into the MultiLyra [12]. Our evaluation covers four common graph applications - Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), Breadth First Search (BFS), and Viterbi (VT) [8]. Two input graphs include Twitter (TT) [2], [7] with 2 billion edges and LiveJournal (LJ) [1], [9] with 69 millions of edges. We generated queries by randomly selecting source vertices. Experiments were run on a cluster of four homogeneous machines with 32 Intel Broadwell cores and 256GB memory, and CentOS Linux release 7.4.1708.

We compare BEAD against the MultiLyra baseline under the following scenarios. A part of the graph (50%, 70% and 90%) is randomly selected from the full graph and chosen as G_0, the first version of the graph. Then, additional portions of the graph are added to G_0 in batches of Δ to emulate a growing graph. All the Δ batches were randomly chosen of different sizes – 1k, 10k, 100k for LJ and 10k, 100k, and 1000k edges for TT; Since TT has roughly ten times the number of vertices as LJ, Δ sizes chosen for TT are ten times that of LJ. Additional δ queries are added to Q_0 to reflect the growing batch of queries. These additional randomly chosen δ queries are added to Q_0 in increments of 8, 16 and 32 queries.

As BEAD is built on top of MultiLyra, to be self-contained and to demonstrate the promises of batching evaluation, we briefly report the baseline performance (more details in [12]).

A. MultiLyra – Scalability with Batch Sizes

We first show the benefits of batching achieved by MultiLyra for G_0 (50%, 70%, 90%) during the evaluation of a total of 256 SSWP queries. For each G_0 (of LJ), we ran the SSWP queries first one by one (i.e., non-batching which is equivalent to PowerLyra [3]) and then in batches of 64, 128, and 256 queries (in *IQT* mode). Table II shows execution time in seconds and the speedups of batching over non-batching. The results show that batching in MultiLyra brings more speedups as the batch size increases, meanwhile the gains decreases as the batch size approaches 256 queries (more details in [12]). Also note that the total number of iterations is reduced dramatically, as the number of iterations for a batch of queries is determined by the "slowest" query, rather than the sum of those for all the queries as in the non-batching case.

TABLE II
Total Execution Time of running 256 SSWP queries using MultiLyra on LJ to compute $Eval(G_0, Q_0) \rightarrow R_0$ with varying batch sizes.

G_0	Batch Size	#Iter.	Time (s)	
50%	1(non-batching)	9286	2759.42	Speedup
	64	400	538.48	5.13×
	128	200	432.81	6.38×
	256	100	431.58	6.39×
70%	1(non-batching)	9420	2462.62	Speedup
	64	400	592.93	4.15×
	128	200	495.03	4.98×
	256	100	457.38	5.38×
90%	1(non-batching)	10060	2713.6	Speedup
	64	400	642.39	4.22×
	128	200	527.05	5.15×
	256	100	483.57	5.61×

B. Graph Updates: BEAD vs. MultiLyra

In this section, we compare the handling of graph updates Δ by BEAD that incrementally reevaluates batch of queries Q_0, with MultiLyra that must evaluate queries Q_0 on graph $G_0 + \Delta$ from scratch. Table III presents the speedups obtained by BEAD over MultiLyra for evaluating 256 queries in a single batch on the updated graph $G_0 + \Delta$, where G_0 is 50% and Δ is set to 100K edges and 10K edges for TT and LJ, respectively. Both BEAD and MultiLyra ran in *IQT* mode. The last column of Table III reports the execution time for MultiLyra.

Speedups delivered by BEAD range from 6.21× for *SSWP* to 26.16× for *SSSP* on TT and from 3.99× for *Viterbi* to 5.34× for *SSSP* on LJ. Note that generally higher speedups are achieved for the larger TT graph than for the smaller LJ graph. This indicates that the savings in work achieved by BEAD's incremental algorithm are greater for the larger TT graph. The overall speedup for *SSWP* on TT is lower than those of the other three graph algorithms. This is because a few queries in Q_0 take much longer to converge than the rest of the queries for *SSWP* – note the very high number of iterations for *SSWP* shown in **#Iter** column in Table III. Consequently, for most iterations, only a few queries are actually active (25 active queries after iteration 15), limiting the benefits of batching.

Next, we perform more detailed experiments, for *SSSP* on TT and *SSWP* on LJ, to study the sensitivity of performance benefits (BEAD over MultiLyra) with respect to a number of factors, including: (a) Varying Δ – for TT, this was varied across 10k, 100k, and 1000k while for LJ it was varied across 1k, 10k, and 100k; (b) Varying the size of G_0 – for both

TABLE III
Speedups of BEAD over MultiLyra when computing $Inc(G_0 + \Delta, Q_0, R_0)$ given $Eval(G_0, Q_0) \rightarrow R_0$, where $G_0 = 50\%$.

G_0	Graph Algo.	Batch Size	Δ	BEAD		MultiLyra
				Speedup	#Iter	Exe. Time
TT (50%)	SSSP	256	100K	26.16×	11	1141.5s
	SSWP	256	100K	6.21×	100	2753.6s
	BFS	256	100K	15.00×	7	510.2s
	VT	256	100K	18.61×	21	1506.9s
LJ (50%)	SSSP	256	10K	5.34×	26	337.7s
	SSWP	256	10K	4.15×	45	431.6s
	BFS	256	10K	4.00×	11	111.0s
	VT	256	10K	3.99×	19	195.0s

TT and LJ this was varied across 50%, 70%, and 90%; (c) Varying batch size – the 256 queries were run in one batch of 256, 2 batches of 128, and 4 batches of 64; and (d) using IQT vs FQT. Table IV and Table V present the results for *SSSP* on TT and Table VI and Table VII present the results for *SSWP* on LJ. The speedups are calculated by comparing the execution time of each configuration with the execution time of the corresponding MultiLyra configuration.

Following are our observations from the above experiments.

(a) *Sensitivity to Varying* Δ – As the size of changes to graph increases, the speedup of the incremental evaluation decreases since more computation is needed to reach convergence. Table IV shows that BEAD on SSSP, for $G_0 = 50\%$, achieves a maximum speedup of 33.37× when $\Delta = 10K$ and a minimum speedup of 7.86× when $\Delta = 1000K$. Similar trend is also observed for SSWP on LJ (see Table VI).

(b) *Sensitivity to Varying* G_0 – Table IV and Table VI show that BEAD's speedups decrease when larger portions of the graph are loaded as G_0. Since larger parts of the graph are more connected for larger G_0, it starts longer evaluation waves through the graph as the graph grows. These tables show the maximum speedups of: 33.37× for $G_0 = 50\%$ vs. 25.91× for $G_0 = 70\%$ for SSSP on TT; and 4.53× for $G_0 = 50\%$ vs. 4.34× for $G_0 = 70\%$ for SSWP on LJ.

(c) *Sensitivity to Varying* Q_0 – We ran 256 queries divided into varying batch sizes to study impact of varying Q_0 size. Our results from Table IV and Table VI show that although BEAD's speedups for different sizes of Q_0 vary, the variation across different Δ sizes is mostly small and no specific size of Q_0 gives the best speedups across different Δ sizes.

(d) *Sensitivity to IQT / FQT* – As mentioned earlier, *IQT* and *FQT* are two modes of evaluation that enable the opportunities to shrink not only the amount of computations but also the amount of data communicated between master and mirror vertices hosted on different machines. To examine if these two modes are still relevant during BEAD's incremental evaluation as the graph grows, we collected the number of messages communicated, as shown in Table V and Table VII. After dividing these messages according to their type, Active vs. Data, it can be seen that 60-84% of the communications are in form of Data messages which is similar to our observations for non-incremental MultiLyra communication in [12]. Thus, as in case of MultiLyra, during incremental evaluation *IQT* typically outperforms *FQT*. The only exception is Table VI where, for small query batch size and/or small Δs, *FQT* performs slightly better by leveraging its low tracking overhead while *IQT* performs better in larger batch sizes and Δs.

C. Graph and Query Updates: BEAD vs. MultiLyra

In this section, we evaluate BEAD when both the graph and the batch of queries simultaneously grow. We evaluate all three policies discussed earlier in Section IV-A: (i) applying graph change then the query change ($\Delta \rightarrow \delta$), (ii) applying the query change then graph change ($\delta \rightarrow \Delta$), and (iii) simultaneously applying both changes ($\Delta \| \delta$). The initial setup is running an original batch of 256 queries (Q_0) for different algorithms on 50% of TT and LJ (G_0). The new batch of queries δ can be of size varying among 8, 16, and 32, while the graph updates Δ are set to 100K for TT and 10K for LJ. Table VIII shows the speedups of BEAD under the three policies. The baseline execution times were collected by running the same batch of $Q_0 + \delta$ queries on the updated graph $G_0 + \Delta$ using MultiLyra.

As shown in Table VIII where the best speedups are marked in red, the ordered evaluation ($\Delta \rightarrow \delta$ and $\delta \rightarrow \Delta$) obtains better performance comparing to simultaneous evaluation ($\Delta \| \delta$). The reason can be understood as follows. During the simultaneous evaluation, the new queries (δ) and old queries (Q_0) are merged into a larger batch ($Q_0 + \delta$). As the old queries were started earlier than the new queries, their vertex values tend to converge earlier. Once their values are converged, they become the overhead of the following iterative evaluation, slowing down the progress of the new queries. Next, we examine how the ordering between the graph updates and query updates affects performance, i.e. $\Delta \rightarrow \delta$ vs. $\delta \rightarrow \Delta$.

Δ-First vs. δ-First Evaluation – As indicated in Table VIII under columns $\Delta \rightarrow \delta$ and $\delta \rightarrow \Delta$, in general, whether applying the graph change Δ at the first place for Q_0 or at second place for $Q_0 + \delta$ with the availability of stable results from the previous step only makes limited differences in performance. However, since the evaluation of sub-batch δ starts from scratch (despite the use of indirect incremental computations), it could take more iterations to traverse the changed graph $G_0 + \Delta$ than the original graph G_0. This effect is more significant when the original graph G_0 is relatively small or the graph change Δ is relatively large. In our setup, graph TT is about 29X larger than LJ in terms of the number of edges, but its update batch size is only 10X larger than that of LJ. Consequently, as shown in Table VIII, when comparing the speedups on TT, with the those on LJ, δ-First evaluation works better on TT, whereas Δ-First evaluation shows superiority on LJ, the smaller graph. For example, Δ-first evaluation obtains a maximum speedup of 5.39× for *SSSP* on TT whereas δ-first evaluation achieves a maximum speedup of 3.28× for *SSSP* on LJ. Note that the speedups after including new queries δ, in addition to graph updates Δ, are lower than those with only graph updates (comparing to Table III) because although queries in Q_0 terminate rapidly, the queries in δ being new take much longer time.

D. Interruption Handling

Finally, we evaluate BEAD in the scenario of interruption – a new request (say Δ_2 and δ_2) arrives in the middle of the incremental evaluation for the prior request (say evaluating $Q_0 + \delta_1$ on graph $G_0 + \Delta_1$). For this evaluation, we first ran 256 queries (Q_0) for each algorithm on the 50% input graphs (G_0) using BEAD. Then, we let BEAD incrementally evaluate the first request – the updated query batch $Q_0 + \delta_1$ on the updated graph $G_0 + \Delta_1$, where δ_1 = 16 and Δ_1 is 100K for TT and 10K for LJ. After that, in the middle of this evaluation, at the points when 50%, 75%, and 100% of the evaluation has been done (in terms of elapsed time), the second request (Δ_2 and δ_2) from the user interrupts BEAD and asks for updated evaluation, that

TABLE IV

SENSITIVITY STUDY OF RUNNING SSSP ON TT USING BEAD WHEN GRAPH CHANGES: $Inc(G_0 + \Delta, Q_0, R_0)$ GIVEN $Eval(G_0, Q_0) \rightarrow R_0$.

				Mode		
				IQT		FQT
G_0	Δ	# $\times Q_0$	#Iter.	Time (s)	Speedup	Speedup
50%	10K	4 × 64	37	69.46	27.79×	27.77×
		2 × 128	19	47.59	33.37×	31.59×
		1 × 256	10	41.36	27.60×	27.09×
	100K	4 × 64	44	65.11	29.65×	22.58×
		2 × 128	22	50.44	31.49×	26.12×
		1 × 256	11	43.63	26.16×	25.45×
	1000K	4 × 64	72	126.32	15.28×	13.75×
		2 × 128	40	121.26	13.10×	10.04×
		1 × 256	23	145.30	7.86×	5.26×
70%	10K	4 × 64	40	81.24	24.15×	20.63×
		2 × 128	22	60.09	25.91×	21.78×
		1 × 256	12	52.08	25.07×	24.47×
	100K	4 × 64	52	88.78	22.10×	14.32×
		2 × 128	29	68.92	22.59×	17.43×
		1 × 256	16	63.52	20.56×	20.27×
	1000K	4 × 64	66	122.00	16.08×	14.39×
		2 × 128	34	90.33	17.24×	13.24×
		1 × 256	17	83.59	15.62×	12.93×
90%	10K	4 × 64	32	84.88	25.64×	20.26×
		2 × 128	18	59.22	27.25×	26.68×
		1 × 256	9	50.01	27.85×	27.72×
	100K	4 × 64	41	89.27	24.38×	15.80×
		2 × 128	22	67.45	23.92×	21.21×
		1 × 256	12	54.53	25.54×	24.29×
	1000K	4 × 64	50	102.26	21.28×	17.56×
		2 × 128	26	68.95	23.40×	23.07×
		1 × 256	14	61.96	22.48×	20.48×

TABLE V

EXTRA NUMBER OF COMMUNICATIONS NEEDED FOR RUNNING SSSP ON TT USING BEAD WHEN GRAPH CHANGES TO COMPUTE: $Inc(G_0 + \Delta, Q_0, R_0)$ GIVEN $Eval(G_0, Q_0) \rightarrow R_0$.

			Message Type		
G_0	Δ	# $\times Q_0$	Active	Data	Total ($\times 10^6$)
50%	10K	4 × 64	20.54%	79.46%	0.04
		2 × 128	20.82%	79.18%	0.03
		1 × 256	21.08%	78.92%	0.03
	100K	4 × 64	20.09%	79.91%	0.57
		2 × 128	20.79%	79.21%	0.44
		1 × 256	21.22%	78.78%	0.36
	1000K	4 × 64	21.54%	78.46%	69.02
		2 × 128	21.67%	78.33%	67.06
		1 × 256	21.74%	78.26%	65.74
70%	10K	4 × 64	18.72%	81.28%	0.05
		2 × 128	19.37%	80.63%	0.03
		1 × 256	19.91%	80.09%	0.02
	100K	4 × 64	21.16%	78.84%	1.16
		2 × 128	21.49%	78.51%	1.08
		1 × 256	21.73%	78.27%	1.01
	1000K	4 × 64	20.04%	79.96%	32.47
		2 × 128	20.01%	79.99%	24.02
		1 × 256	19.94%	80.06%	14.80
90%	10K	4 × 64	17.28%	82.72%	0.03
		2 × 128	17.47%	82.53%	0.02
		1 × 256	17.94%	82.06%	0.01
	100K	4 × 64	18.66%	81.34%	0.27
		2 × 128	19.03%	80.97%	0.19
		1 × 256	19.35%	80.65%	0.14
	1000K	4 × 64	18.12%	81.88%	3.44
		2 × 128	18.64%	81.36%	2.67
		1 × 256	19.02%	80.98%	2.09

TABLE VI

SENSITIVITY STUDY OF RUNNING SSWP ON LJ USING BEAD WHEN GRAPH CHANGES: $Inc(G_0 + \Delta, Q_0, R_0)$ GIVEN $Eval(G_0, Q_0) \rightarrow R_0$.

				Mode		
				IQT		FQT
G_0	Δ	# $\times Q_0$	#Iter.	Time (s)	Speedup	Speedup
50%	1k	4 × 64	155	118.96	4.53×	4.57×
		2 × 128	81	103.98	4.16×	4.15×
		1 × 256	45	107.77	4.00×	4.03×
	10k	4 × 64	155	119.92	4.49×	4.56×
		2 × 128	81	102.14	4.24×	4.36×
		1 × 256	45	103.75	4.15×	4.21×
	100k	4 × 64	155	126.93	4.24×	4.09×
		2 × 128	81	109.28	3.96×	3.31×
		1 × 256	45	126.77	3.40×	2.81×
70%	1k	4 × 64	178	136.75	4.34×	4.24×
		2 × 128	130	168.99	2.86×	2.89×
		1 × 256	73	171.02	2.67×	2.55×
	10k	4 × 64	178	142.20	4.17×	4.14×
		2 × 128	130	167.32	2.89×	2.90×
		1 × 256	73	178.26	2.57×	2.53×
	100k	4 × 64	178	137.46	4.31×	4.26×
		2 × 128	130	168.05	2.88×	2.87×
		1 × 256	73	175.88	2.60×	2.33×
90%	1k	4 × 64	265	233.74	2.75×	2.93×
		2 × 128	140	218.56	2.41×	2.16×
		1 × 256	87	201.21	2.40×	2.10×
	10K	4 × 64	265	233.71	2.75×	2.87×
		2 × 128	140	226.76	2.32×	2.04×
		1 × 256	87	203.87	2.37×	2.03×
	100K	4 × 64	265	255.06	2.52×	2.47×
		2 × 128	140	229.02	2.30×	2.06×
		1 × 256	87	227.71	2.12×	1.76×

TABLE VII

EXTRA NUMBER OF COMMUNICATIONS NEEDED FOR RUNNING SSWP ON LJ USING BEAD WHEN GRAPH CHANGES TO COMPUTE: $Inc(G_0 + \Delta, Q_0, R_0)$ GIVEN $Eval(G_0, Q_0) \rightarrow R_0$.

			Message Type		
G_0	Δ	# $\times Q_0$	Active	Data	Total ($\times 10^6$)
50%	1K	4 × 64	39.78%	60.22%	0.36
		2 × 128	39.76%	60.24%	0.34
		1 × 256	39.68%	60.32%	0.31
	10K	4 × 64	37.98%	62.02%	0.38
		2 × 128	38.77%	61.23%	0.35
		1 × 256	39.14%	60.86%	0.32
	100K	4 × 64	15.85%	84.15%	24.31
		2 × 128	15.82%	84.18%	24.18
		1 × 256	15.78%	84.22%	24.09
70%	1K	4 × 64	35.91%	64.09%	0.26
		2 × 128	35.96%	64.04%	0.26
		1 × 256	35.77%	64.23%	0.25
	10K	4 × 64	34.66%	65.34%	0.27
		2 × 128	35.32%	64.68%	0.27
		1 × 256	35.43%	64.57%	0.25
	100K	4 × 64	26.93%	73.07%	0.43
		2 × 128	30.34%	69.66%	0.35
		1 × 256	32.49%	67.51%	0.29
90%	1K	4 × 64	34.93%	65.07%	0.45
		2 × 128	35.02%	64.98%	0.45
		1 × 256	35.01%	64.99%	0.45
	10K	4 × 64	34.30%	65.70%	0.47
		2 × 128	34.70%	65.30%	0.46
		1 × 256	34.84%	65.16%	0.45
	100K	4 × 64	15.34%	84.66%	16.08
		2 × 128	15.33%	84.67%	15.99
		1 × 256	15.32%	84.68%	15.94

TABLE VIII
SPEEDUPS OF BEAD OVER MULTILYRA ON SIMULTANEOUS GRAPH AND QUERY UPDATES: COMPUTING $Inc(G_0+\Delta, Q_0+\delta, R_0)$ GIVEN $Eval(G_0, Q_0) \rightarrow R_0$, WHERE $G_0 = 50\%$ AND $Q_0 = 256$.

				BEAD			
				$\Delta \rightarrow \delta$	$\delta \rightarrow \Delta$	$\Delta \| \delta$	
G	**Algo.**	**Δ**	**δ**	**Speedup**	**Speedup**	**Speedup**	MultiLyra
TT	**SSSP**	**100K**	**8**	5.39×	5.28×	2.67×	1241.6s
			16	4.94×	5.02×	2.50×	1303.0s
			32	3.88×	4.02×	2.25×	1429.1s
	SSWP	**100K**	**8**	5.57×	5.66×	5.01×	2891.6s
			16	5.42×	5.39×	4.84×	2887.0s
			32	5.33×	4.98×	4.30×	3131.8s
	BFS	**100K**	**8**	5.10×	5.39×	2.89×	543.7s
			16	4.02×	4.21×	2.28×	562.7s
			32	3.24×	3.46×	2.08×	571.4s
	VT	**100K**	**8**	4.13×	4.25×	2.33×	1571.4s
			16	3.39×	3.55×	1.99×	1576.7s
			32	2.67×	2.79×	1.80×	1725.8s

			BEAD			
			$\Delta \rightarrow \delta$	$\delta \rightarrow \Delta$	$\Delta \| \delta$	
G	**Δ**	**δ**	**Speedup**	**Speedup**	**Speedup**	MultiLyra
LJ	**10K**	**8**	3.19×	3.28×	2.18×	343.3s
		16	2.92×	2.74×	1.86×	363.5s
		32	2.65×	2.60×	1.85×	399.9s
	10K	**8**	3.23×	3.25×	2.66×	456.6s
		16	3.20×	3.09×	2.43×	471.5s
		32	2.96×	2.76×	2.28×	469.4s
	10K	**8**	2.73×	2.65×	1.80×	112.6s
		16	2.69×	2.53×	1.65×	121.3s
		32	2.40×	2.38×	1.51×	125.4s
	10K	**8**	2.88×	2.73×	2.15×	211.4s
		16	2.52×	2.50×	1.82×	214.0s
		32	2.16×	2.03×	1.74×	221.0s

TABLE IX
PERFORMANCE OF BEAD UNDER USER INTERRUPTIONS COMPUTING $Inc(G_0+\Delta_1+\Delta_2, Q_0+\delta_1+\delta_2, R_0)$ IN TWO REQUESTS GIVEN $Eval(G_0, Q_0) \rightarrow R_0$, WHERE $G_0 = 50\%$.

					Latency (Seconds)		
					Interruption Points		
G	**Algo.**	**Q_0**	**Δ_1:δ_1**	**Δ_2:δ_2**	**50%**	**75%**	**100%**
TT	**SSSP**	**256**	**100K:16**	**10K:8**	469.94	465.49	489.41
	SSWP	**256**	**100K:16**	**10K:8**	667.24	719.30	795.71
	BFS	**256**	**100K:16**	**10K:8**	215.06	208.57	231.69
	VT	**256**	**100K:16**	**10K:8**	747.49	770.42	787.78
LJ	**SSSP**	**256**	**10K:16**	**1K:8**	200.12	230.57	236.12
	SSWP	**256**	**10K:16**	**1K:8**	144.53	161.26	169.83
	BFS	**256**	**10K:16**	**1K:8**	68.63	82.92	85.14
	VT	**256**	**10K:16**	**1K:8**	115.24	136.44	138.18

is, $Q_0+\delta_1+\delta_2$ on $G_0+\Delta_1+\Delta_2$. The 50% and 75% scenarios correspond to the *anytime interruption* strategy used by BEAD whereas the 100% scenario mimics the *following convergence* strategy that can be used alternatively (see Section IV-B).

In Table IX the 100% scenario (i.e., *following convergence*) ensures that the precise results R_1 are available to the incremental computation of the second request, while in 50% and 75% scenarios only the approximate results $\approx R_1$ are available. We observe that the immediately starting of the second request using $\approx R_1$ (i.e., *anytime interruption*) leads to lower response latency for the second request. Although using precise results R_1 can reduce the work performed in evaluating the second request, waiting to compute the second request outweighs this benefit for the 50% and 75% interruption points.

VI. CONCLUSION

In this paper, we generalized our prior work on MultiLyra to consider scenarios in which analytics demands of the user evolve. While MultiLyra delivers high performance by solving batches of queries simultaneously, BEAD achieves the same in the presence of changes to the graph and/or query set. Experiments show that BEAD's batched evaluation of 256 queries after graph and also query changes on TT, outperforms MultiLyra by factors of up to 26.16× and 5.66×.

ACKNOWLEDGEMENTS: Supported by NSF grants CCF-2002554, CCF-1813173, and CCF-2028714 to UCR.

REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. "Group formation in large social networks: membership, growth, and evolution," In *Int. Conf. on Knowledge Discovery and Data Mining*, 2006.

[2] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. "Measuring user influence in twitter: The million follower fallacy," In *International AAAI Conference on Weblogs and Social Media*, 2010.

[3] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing*, 5(3), 13, 2019.

[4] P. Pan and C. Li. "Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines," In *IEEE ICCD*, 2017.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. "Powergraph: Distributed graph-parallel computation on natural graphs," In *USENIX Symppsium on OSDI*, pages 17-30, 2012.

[6] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and I. Stoica. "GraphX: Graph processing in a distributed dataflow framework," In *USENIX Symposium on OSDI*, pages 599-613, 2014.

[7] 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.

[8] J. Lember, D. Gasbarra, A. Koloydenko, and K. Kuljus. "Estimation of Viterbi Path in Bayesian Hidden Markov Models," *arXiv:1802.01630*'18.

[9] J. Leskovec. "Stanford large network dataset collection," *http://snap.stanford.edu/data/index.html*, 2011.

[10] 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," *Proc. VLDB Endowment 5*, 8 (2012).

[11] 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 Int. Conf. on Management of Data*, 1010.

[12] A. Mazloumi, X. Jiang, and R. Gupta. "MultiLyra: Scalable Distributed Evaluation of Batches of Iterative Graph Queries," In *IEEE Big Data*'19.

[13] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi. "Naiad: a timely dataflow system," *ACM SOSP*, pages 439–455, 2013.

[14] X. Shi, B. Cui, Y. Shao, and Y. Tong. "Tornado: A System For Real-Time Iterative Analysis Over Evolving Data," *SIGMOD*, 2016.

[15] 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 *Proc. of the VLDB Endowment*, 2015.

[16] C. Xu, A. Mazloumi, X. Jiang, and R. Gupta. "SimGQ: Simultaneously Evaluating Iterative Graph Queries," In *IEEE HiPC*, 2020.

[17] K. Vora, S-C. Koduru, and R. Gupta. "ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency based DSM," In *SIGPLAN OOPSLA*, pages 861-878, 2014.

[18] K. Vora, C. Tian, R. Gupta, and Z. Hu. "CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing," *ACM ASPLOS*, 1017.

[19] C. Xu, K. Vora, and R. Gupta. "PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics," In *ACM ASPLOS*, 2019.

[20] 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 *Proc. of the VLDB Endowment*, 9(7), pages 564-575, 2016.