UNIVERSITY OF CALIFORNIA
RIVERSIDE

Runtime Optimizations for Evaluating Batches of Graph Queries

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Chengshuo Xu

December 2021

Acknowledgments

This dissertation would not have been possible without the support and help of many people across the walks in my life.

I would like to express my deepest gratitude to my academic advisor, Prof. Rajiv Gupta, for his constant guidance and support throughout my path to Ph.D. I would like to thank Prof. Gupta for giving me the opportunity to work with him, and learn both technical and non-technical skills from him during the past five years. This dissertation is a culmination of his careful mentoring throughout my Ph.D.

I would like to thank Prof. Zhijia Zhao for the collaboration and guidance during my research. I would like to thank my dissertation committee members, Prof. Nael Abu-Ghazaleh and Prof. Daniel Wong, for their valuable feedback and support.

I would like to acknowledge the support of the National Science Foundation via grants CCF-2028714, CCF-2002554, CCF-1813173, and CCF-1524852 to UC Riverside.

I would like to thank Keval Vora for mentoring me during the first year of my Ph.D. study.

I would express my gratitude to the members in our research group for their help and support: Mahbod Afarin, Arash Alavi, Zachary Benavides, Min Feng, Yongjian Hu, Xiaolin Jiang, Gurneet Kaur, Hongbo Li, Abbas Mazloumi, Shafiur Rahman, Xiaofan Sun, Keval Vora, Xizhe Yin, and Bo Zhou. I am also grateful to all my teachers that I have had throughout my life.

Finally, I would like to thank my family and friends for their love and support throughout my life.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Runtime Optimizations for Evaluating Batches of Graph Queries

by

Chengshuo Xu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Dr. Rajiv Gupta, Chairperson

Graph processing frameworks are typically designed to optimize the evaluation of a single graph query. However, in practice, we often need to respond to multiple graph queries, either from different users or from a single user performing a complex analytics task. This thesis is aimed at simultaneously evaluating batches of graph queries of two types: point-to-all and point-to-point. By fully utilizing system resources, batched evaluation amortizes runtime overheads incurred due to fetching vertices and edge lists, synchronizing threads, and maintaining computation frontiers. In addition, new runtime optimizations are developed that enable faster evaluation of batches of queries than their independent and one by one evaluation.

In context of point-to-all queries we develop the sharing optimization that dynamically identifies shared queries that substantially represent subcomputations in the evaluation of different queries in a batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all queries in the batch. The resulting SimGQ system delivers substantial speedups over a conventional framework that evaluates and responds to queries one by one. We have also adapted the batching principles used by SimGQ to the streaming graph scenario in which we continuously maintain the results of small batch of shared queries and use them for low cost evaluation of arbitrary user queries.

For point-to-point queries we have identified unique characteristics of such queries and based upon them developed two new optimizations. The first optimization, online pruning, eliminates propagation from vertices that are determined to not contribute to a query's final solution and thus enables early termination. The second optimization, dynamic direction prediction, dynamically selects direction in which to evaluate the query – either forward (from source) or backward (from destination) – as their costs can differ greatly. The resulting system, PnP, delivers substantial performance benefits over the state-of-the-art. To solve a batch of point-to-point queries, we extended this system by incorporating the batching principles in SimGQ along with a new query aggregation technique that eliminates the redundant computation across point-to-point queries that share the same source or destination vertex

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs, internet topology, brain networks etc.) to uncover insights by analyzing high volumes of connected data. An iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When the solution values of all vertices become stable, the algorithm terminates. It has been seen that real world graphs are often large with millions of vertices and billions of edges. Moreover, 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 data-intensive and often compute-intensive. Therefore, there has been a great deal of interest in developing scalable graph analytics systems like Pregel [26], GraphLab [24], GraphIt [60], PowerGraph [11], Galois [36], GraphChi [21], Ligra [41], ASPIRE [46].

While the performance of graph analytics has improved greatly due to advances introduced by the above systems, much of this research has focussed on developing highly parallel algorithms for solving a single iterative graph analytic query, which in many cases is a point-to-all graph query computing property values from a single source to all other vertices in the graph (e.g., SSSP(s) query computes shortest paths from a single source s to all other vertices in the graph). There are a variety of such single-query computing platforms for graph analytics including shared-memory systems, distributed clusters, and systems with accelerators like GPUs. However, in practice the following two scenarios involve multiple queries: (a) Single-User scenario in which a single user may conduct a complex analytics task requiring issuing of multiple queries; and (b) Multi-User scenarios as in [34] and [44] where the same data set is queried by many users. In both scenarios, machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a modern server with many cores and substantial memory resources.

In addition to the conventional point-to-all graph query, point-to-point graph query, which computes from a single source vertex to a single destination vertex, is also drawing attention from the research community. For instance, recently Yan et al. [58] observed that many applications on large graphs simply require computing point-to-point variants of heavyweight computations. As an example, when analyzing a graph that represents online shopping history of shoppers, a business may be interested in point-to-point queries over pairs of certain important shoppers. Thus, given a pair of distinct vertices (s, d) in a graph, we are interested in computing point-to-point versions of standard computations such as, shortest path from s to d, widest path from s to d and number of paths from s to d. Yan et al. developed the Quegel [58] framework to solve point-to-point queries. Since many such pairs of queries need to be evaluated, Yan et al. overlap the evaluation of multiple point-to-point queries.

While the focus of above systems is on solving single point-to-all queries and single point-to-point queries, this thesis develops techniques for simultaneously evaluating batches of point-to-all and point-to-point queries. We first explore the opportunity to amortize the runtime overhead and reduce the computational cost for evaluating a batch of point-to-all graph queries on static graphs, and then extend the system for streaming graph scenarios. After that, we discuss the batched evaluation of point-to-point queries following the same batching principle. Since point-to-point graph queries are a class of workload which have not been widely studied in the literature, we start with a study on how to accelerate a single point-to-point graph query, and then build upon the observations for a single point-to-point query to develop further optimizations specifically for evaluating a batch of point-to-point graph queries.

1.1 Dissertation Overview

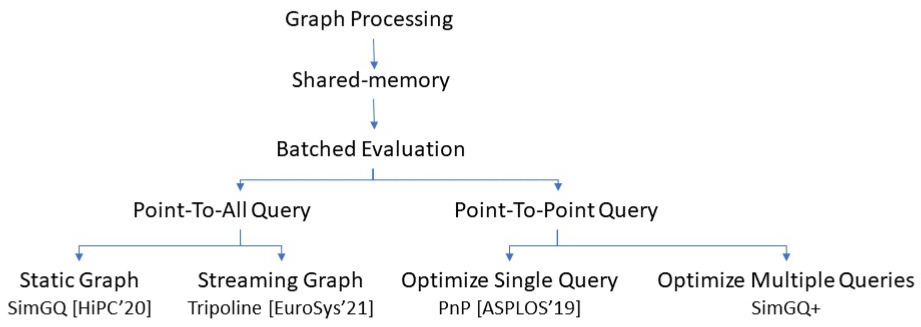


Figure 1.1: Dissertation Overview

In this thesis, we exploit the synergy that exists in batched execution of iterative vertex graph queries to accelerate the evaluation of a batch of point-to-all or point-to-point iterative graph queries. Figure 1.1 shows the overview of this dissertation. We first studied batching in shared-memory architecture and developed online sharing optimization to improve the performance of a batch of point-to-all queries on static and dynamic graphs. And then we study and discover the unique characteristics of point-to-point queries and further tackle the issues in evaluating a batch of point-to-point queries.

1.1.1 Simultaneously Evaluating Batches of Point-to-All Queries

Conventional point-to-all graph queries have been widely studied in the literature. Therefore, we do not further dive into how to evaluate a single point-to-all graph query. Instead, we explore the synergy across a batch of queries. We first study the problem on static graphs which models the relationships between entities that remain unchanged during the execution of graph algorithms, and then adapt the observations and techniques developed for static graphs to streaming graph scenarios.

Processing Static Graphs

Given an input graph and a batch of point-to-all queries, we can synergistically perform simultaneous evaluation of all queries in a batch to deliver results of all queries in a greatly reduced time. Essentially the synergy in evaluation of queries, that exists due to the substantial overlap between computations and graph traversals for different queries, is exploited to amortize the runtime overhead and computation costs across the simultaneously evaluated queries. To amortize computation costs, we develop a novel strategy that dynamically identifies *shared queries* whose computations substantially overlap with the computations performed by multiple queries in the batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all the queries in the batch. With the *sharing* optimization, we develop SimGQ, a graph analytics system aimed at evaluating a batch of vertex queries received from users for different source vertices in the graph.

Processing Streaming Graphs

In many real-world application scenarios, a stream of updates are continuously applied to the graph, often in batches for better efficiency, known as the streaming graph scenario. Incremental computation can be used to quickly update the result of fixed queries in streaming graph scenarios. When a batch of updates is applied to the graph, incremental computation reuses the query result on the previous version of the graph and performs iterative computation starting from property values which is closer to convergence compared with reevaluating query from scratch. However, the applicability of incremental computation is restricted to accelerate fixed queries. In this thesis, we extend SimGQ to generalize incremental computation to handle batches of *arbitrary* user queries in the streaming graph scenario. Essentially we continuously apply incremental computation to maintain the results of a small batch of preselected shared queries upon graph mutations, which can be used to accelerate a batch of arbitrary user queries on the latest version of the graph using the *sharing* optimization in SimGQ.

1.1.2 Efficiently Evaluating Batches of Point-to-Point Queries

Since not much research has been conducted on evaluating point-to-point queries, we first propose a greatly enhanced method, based on which we developed a system named PnP, for evaluating a single point-to-point query by leveraging the observation that point-to-point queries may only require a small portion of computation compared with its heavy-weight point-to-all counterpart. After that, we further study the opportunity to eliminate redundant computation across a batch of point-to-point queries.

Optimizing a Single Point-to-Point Query

To quickly respond to point-to-point queries, we developed the **PnP** system that incorporates optimizations that greatly improve performance over the Quegel system [58]. First, wasteful work performed by Quegel is reduced via an online pruning optimization that eliminates unnecessary propagation to vertices which are determined to not contribute to the query's final answer. Second, we recognize that evaluation times of point-to-point queries in backward and forward directions can greatly differ because different subgraphs are used with the traversal in different directions, and thus introduce a light-weight dynamic direction selection optimization that at runtime predicts the faster direction of execution based on each combination of input query and input graph.

Optimizing a Batch of Point-to-Point Query

Moreover, we are also interested in evaluating a batch of point-to-point queries as users are interested in computing point-to-point information between a subset of vertices in the graph, as mentioned in Quegel [58]. In addition to the batching principle studied in SimGQ and the online pruning technique for a single point-to-point query developed in PnP, we developed a novel query aggregation optimization for evaluating a batch of point-to-point graph queries. Essentially query aggregation eliminates the shared computation across point-to-point queries with the same source or destination which widely exists in the workload of computing point-to-point information between a subset of vertices in the graph.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents SimGQ, the system for batched evaluation of point-to-all graph queries. Chapter 3 describes how the batching principle and sharing optimization in SimGQ can be leveraged to quickly respond to arbitrary queries in the streaming graph scenarios. Chapter 4 presents the optimizations and the resulting two phase algorithm for fast evaluation of a single point-to-point query. Chapter 5 discusses batched evaluation of point-to-point queries which combines the batching principle from SimGQ, the pruning technique from PnP, and a new query aggregation optimization. Chapter 6 discusses various related works in the literature. Chapter 7 concludes the thesis and discusses directions for future work.

Chapter 2

SimGQ – Batched Evaluation with Online Sharing Optimization

Graph analytics is employed in many domains (e.g., social networks, web graphs, internet topology, brain networks etc.) to uncover insights by analyzing high volumes of connected data. An iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When the solution values of all vertices become stable, the algorithm terminates. The existing graph processing frameworks are typically designed to optimize the evaluation of a single iterative graph query. However, in practice, we often need to respond to multiple graph queries, either from different users or from a single user performing a complex analytics task. In both scenarios, machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a server with many cores and substantial memory.

In this chapter we exploit the synergy across a batch of point-to-all graph queries based on which we develop a graph analytics system SimGQ [56], aimed at evaluating a batch of point-to-all queries received from users for different source vertices of a large graph. For example, for SSSP algorithm, we may be faced with the following batch of point-to-all queries: $SSSP(s_1)$, $SSSP(s_2)$, $\cdots\cdots$ $SSSP(s_n)$. Many other important algorithms belong to this category [22, 16, 13] etc. Our overall approach is as follows. Given an input graph and batch of vertex queries, we synergistically perform simultaneous evaluation of all queries in a batch to deliver results of all queries in a greatly reduced time. Essentially the synergy in evaluation of queries, that exists due to the substantial overlap between computations and graph traversals for different queries, is exploited to amortize the runtime overhead and computation costs across the simultaneously evaluated queries. Two techniques, batching and online sharing, are employed to simultaneously and efficiently evaluate a set of queries.

(a) **Batching for Resource Utilization and Amortizing Overheads.** Batching takes a group of queries, forming the batch, and simultaneously processes these queries to achieve higher throughput by fully utilizing system resources and amortizing runtime overhead (e.g., synchronization) costs across queries. SimGQ is capable of evaluating large batches (up to 512 queries) of a general class of queries on a shared-memory system for high throughputs.

(b) **Online Sharing.** To amortize computation costs, we develop a novel strategy that dynamically identifies *shared queries* whose computations substantially overlap with the computations performed by multiple queries in the batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all the queries in the batch. The shared subcomputations are essentially query evaluations for a small set of high degree source vertices, different from the source vertices of queries in the batch, such that they can be used to accelerate the evaluation of all queries in the batch.

Online sharing has multiple advantages over classical global indexing methods for optimizing evaluation of queries. First, indexing entails heavy weight precomputation used to build a large index that can be used to accelerate all future queries (e.g., Quegel [58] uses Hub-Accelerator based indexing). Second, as soon as the graph changes, precomputed indexing/profiling information becomes invalid. The online sharing as performed by SimGQ involves no precomputation and thus eliminates its high cost while also accommodating changes to the graph between different batches of queries. Thus, our approach applies to streaming/evolving graphs.

In SimGQ, the evaluation of a batch of queries is carried out as follows. We partially evaluate the queries in the batch for a few iterations till some high degree vertices enter the frontiers of the queries in the batch. We *pause* the evaluation of the batch queries and select a small set of high degree vertices encountered. Treating selected vertices as source vertices of queries, we construct a batch of *shared queries* and then evaluate this batch. The results of shared queries are then used to *quickly update* the solutions of all queries in the original batch and hence accelerate their advance towards the final stable solution. Finally, we *resume* the paused evaluation of original batch till their stable solutions have been found. By simultaneously evaluating queries we also amortize the runtime overheads incurred, such as costs of accessing vertices and edges, synchronization costs, and maintaining frontiers as multiple queries traverse the same regions of the graph.

We implemented SimGQ by modifying the state-of-the-art Ligra [41] system. Our experiments with multiple input power-law graphs and multiple graph algorithms demonstrate best speedups ranging from 1.53× to 45.67× with batch sizes ranging up to 512 queries over the a baseline implementation that evaluates the queries one by one using the state of the art Ligra system. We also show that both batching and sharing techniques contribute substantially to the speedups.

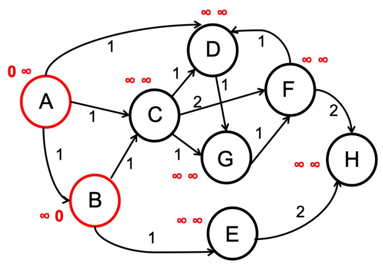
The remainder of the chapter is organized as follows. In section 2.1 we first provide an overview of SimGQ and then present our algorithms in detail. In section 2.2 we experimentally evaluate SimGQ. Finally, we summarize this chapter in section 2.3.

2.1 SimGQ: Evaluating a Batch of Queries

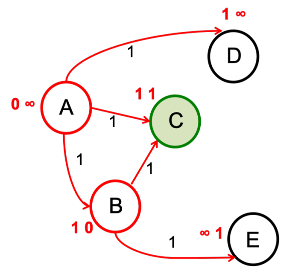
When a group of iterative graph queries are evaluated as a *batch*, following opportunities for speeding up their evaluation arise that are ignored when evaluating the queries one by one. First, it is easy to see that during batched evaluation, we can share the iteration overhead across the queries. This overhead includes the cost of iterating over the loop, synchronizing threads at the barrier, as well as fetching vertex values and edge lists of active vertices to update vertex values and the computation frontier. Second, synergy or overlap between computations performed by the queries can be exploited to reduce the overall computation performed. In particular, we can identify and evaluate *shared queries* whose results can be used to accelerate the evaluation of all the queries in the batch. The computation performed by the shared queries substantially represent subcomputations that are performed by many queries in the batch. This is because different queries typically traverse the majority of the graph and consequently present an opportunity to share a subcomputation across multiple queries. By evaluating the shared queries once, we can speedup the evaluation of the entire batch of queries. Note that the shared queries must be identified dynamically because they may vary from one batch to another.

2.1.1 Overview of SimGQ

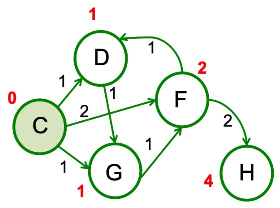
Next we provide an overview of SimGQ via an illustrating example. Figure 2.1 shows how a batch of two queries can be synergistically evaluated by identifying and evaluating shared queries first. While in the example we use a directed graph, our approach works equally well for undirected graphs with a minor adjustment. As in other works, each undirected edge is represented by a pair of directed edges with equal weight.



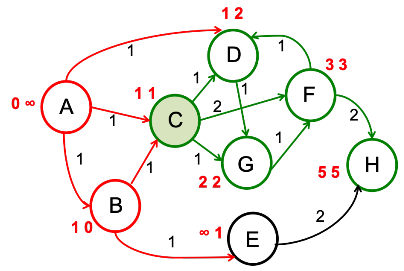
(a) *Initialization* for Batch(A,B) of SSSP Queries.



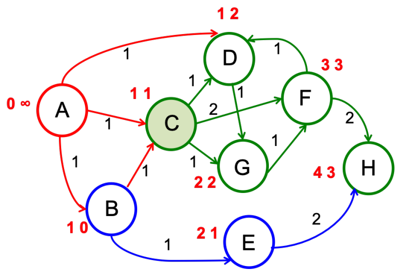
(b) *Phase I.* Evaluate Batch(A,B), Pause and Identify SSET.



(c) *Phase II.* Evaluate SSET.



(d) *Phase II Contd.* Update Batch(A,B) Results Using SSET Results.



(e) *Phase III.* Resume Batch(A,B) Evaluation till Termination.

Figure 2.1: Overview of Sharing Among a Batch of Queries

Initialization Step. Since all queries in a given batch are to be evaluated simultaneously, each vertex is assigned a vector to hold data values for all queries in the batch – each position in the vector corresponds to a specific query in the batch. In Figure 2.1a we aim to solve a batch of two SSSP queries for source vertices A and B marked in red. Each node is annotated with a pair of initial values for the two queries, A first and then B. Initial value 0 is assigned to source vertices and value ∞ to all non-source vertices for each of the SSSP queries.

Phase I: Identifying Shared Queries. Simultaneously starting from the source vertices, we start traversing the graph updating the shortest path lengths for the processed vertices along the frontier as shown in red in Figure 2.1b. The evaluation of the batch continues and once good candidate vertices for *shared queries* SSET are found, the evaluation of the batch is *paused*. Let us assume that after one iteration we identify SSSP(C) (C marked in green) as a good shared query candidate for the two queries in the batch in our example. Thus, we *pause* the evaluation of the batch queries and proceed to the next step to process the identified shared queries.

Phase II. Accelerating Batch Queries Using Shared Queries. In this step we evaluate the shared queries first, that is we evaluate them till their stable results have been computed. For example, in Figure 2.1c we evaluate the shared query SSSP(C). Once the shared queries have been evaluated, their results are used to rapidly update the partial results of all the original batch queries as shown in Figure 2.1d. Note that at this point the results of all vertices except B and E have already reached their final stable values. That is, the evaluation of batch queries has greatly advanced or *accelerated*.

Phase III. Completing the Evaluation of Batch Queries. In this final step we *resume* the evaluation of batch queries from the frontier at which the evaluation was paused earlier. In our example, the resumption of evaluation takes place at vertices B and E and finally the algorithm terminates after updating the results at vertices E and H. Note that if the acceleration performed in Phase II is effective, the combined cost of Phase I and Phase III would be significantly less than the cost of evaluating the batch without employing *sharing* affected via Phase II.

While the above example provides an overview of our approach, many algorithm details and heuristic criteria need to be developed. For example, there are different ways to select shared queries (queries on vertices with high centrality or high degree, queries on vertices that are reachable by most source vertices in the batch etc.). Since our work focuses on power-law graphs that have small diameter and skewed degree distribution, *high degree vertices* are the best candidates for global queries that in general traverse nearly the entire graph. Our algorithm first marks a set of high degree vertices as *potential shared vertices*. At runtime, a heuristic is used to select a small subset of *shared vertices* that are not only marked, but also have been encountered more frequently during partial evaluation of batch queries. After evaluating the *shared queries*, we use the results to quickly update the results of all batch queries. In subsequent subsections we present a push-style evaluation of a batch of queries assisted by our idea of using shared queries.

2.1.2 Push-Style Batch Evaluation With Sharing

Now we present a detailed algorithm that evaluates a batch of vertex queries, employing both batching and sharing, using Push model (a similar algorithm can be easily designed for the Pull model). In Algorithm 1, function EVALUATEBATCH (line 3) simultaneously evaluates a batch of vertex queries for source vertices $s_1, s_2, ..., s_k$, over a directed graph $G\ (V, E)$. The algorithm uses

Algorithm 1 Batched Evaluation With Sharing

```
1:  Given: Directed graph $G(V, E)$; High Degree Set $M \subset V$ of Marked Vertices
2:  Goal: Evaluate a Batch of Queries, QUERYBATCH $\leftarrow \{ Q_1(s_1), Q_2(s_2), ..., Q_k(s_k) \}$
3:  function EVALUATEBATCH( QUERYBATCH )
4:      ▷ [Initialization Step]
5:      INITIALIZE RESULTT for QUERYBATCH
6:      ACTIVE $\leftarrow \{ s_1, s_2, ..., s_k \}$; NEXTTRACK $\leftarrow \phi$; ITER $\leftarrow 0$
7:      CURRTRACK $\leftarrow \{ (s_i, Q_i) : Q_i(s_i) \in$ QUERYBATCH $\}$
8:      ▷ Iterate till Convergence
9:      while ACTIVE $\neq \phi$ do
10:         ▷ [Phase I: Iteration $\leq$ p] [Phase III: Iteration $>$ p]
11:         ▷ Process Active Vertices
12:         ACTIVE $\leftarrow$ PROCESSBATCH ( ACTIVE, ITER, CURRTRACK, NEXTTRACK, RESULTT )
13:         if ITER = $p$ then ▷ [Phase II]
14:             ▷ Identify #SSET as the Most Frequently Visited Vertices from $M$ as the source of Shared Queries
15:             SSET $\leftarrow$ SELECTSHAREDQS ($M$, Visits, #SSET)
16:             ▷ Evaluate Shared Queries with Sources in SSET
17:             SHAREDT $\leftarrow$ EVALUATEBATCH (SSET)
18:             ▷ Update RESULTT using SHAREDT
19:             SHAREUPDATEBATCH ( SSET, SHAREDT, RESULTT )
20:         end if
21:         CURRTRACK $\leftarrow$ NEXTTRACK; NEXTTRACK $\leftarrow \phi$; ITER++
22:     end while
23:     return RESULTT
24: end function
```

Algorithm 2 Batched EdgeMap Function to Update Vertex Values and Compute New Frontier

```
1: function PROCESSBATCH ( ACTIVE, ITER, CURRTRACK, NEXTTRACK, RESULTT )
2:     NEWACTIVE ← $\phi$
3:     for all $v \in$ ACTIVE in parallel do
4:         for all $e \in G.outEdges(v)$ in parallel do
5:             ▷ Apply conventional Update on $e.dest$
6:             $changed$ ← EDGEFUNCBATCH ( $e$, CURRTRACK, NEXTTRACK, RESULTT)
7:             if ( ITER $\leq p$ ) and ( $e.dest \in M$ ) then Visits[$e.dest$]++ end if
8:             ▷ Update Active Vertex Set for next Iteration
9:             if $changed$ then NEWACTIVE ← NEWACTIVE $\cup$ $\{e.dest\}$ end if
10:        end forall
11:    end forall
12:    return NEWACTIVE
13: end function
```

$M \subset V$ as a set of marked high degree vertices from which a small number of vertices are selected to form *shared queries*; different batches of queries yield different shared queries. In our experiments $|M|$ is set to 100 to provide choices that suit different batches, while up to 5 shared queries are selected to limit the overhead of sharing (i.e., SSET size is 5). The algorithm maintains an ACTIVE vertex set, the combined frontier for all queries in the batch. Although ACTIVE tells which vertices are active, it cannot tell which queries are associated with each active vertex. Therefore, in addition to ACTIVE, our algorithm maintains two fine-grained active lists, CURRTRACK and NEXTTRACK, to indicate for each active vertex all the queries whose frontier the active vertex belongs to. While CURRTRACK is the information for active set being processed, NEXTTRACK is the corresponding information for the active set being formed for the next super step of the algorithm. The RESULTT maintains the results of all the queries for each vertex.

Algorithm 3 Batched Edge Update Function

```
 1: function EDGEFUNCBATCH ($e$, CURRTRACK, NEXTTRACK, RESULTT)
 2:     ▷ Initialize RETVALUE to false.
 3:     ▷ Set to true if property value of vertex $e.dest$ is changed.
 4:     RETVALUE $\leftarrow$ false
 5:     for all $Q_i(s_i) \in$ QueryBatch do
 6:         ▷ Only Attemp Update for Queries activated $e.source$
 7:         if $(e.source, Q_i) \in$ CURRTRACK then
 8:             ▷ Perform Update via $e$
 9:             if UPDATEFUNC($e$, $Q_i$, RESULTT) == true then
10:                 ▷ Schedule $e.dest$ for next Iteration
11:                 RETVALUE $\leftarrow$ true
12:                 NEXTTRACK $\leftarrow$ NEXTTRACK $\cup \{(e.dest, Q_i)\}$
13:             end if
14:         end if
15:     end for
16:     return RETVALUE
17: end function
```

Following the initialization step (lines 4-7) in Algorithm 1, in each super iteration (lines 9-22), the vertices in ACTIVE vertex set are processed in parallel by calling function PROCESSBATCH (Algorithm 2). Function PROCESSBATCH updates the value of out-neighbors of active vertices in Push style fashion and generates NEWACTIVE containing the active vertices for next iteration which it returns to EVALUATEBATCH at the end. The work performed by the loop at line 9 executes the three phases of our algorithm. The first p iterations form Phase I, following which, next in Phase II first shared queries SSET are identified by calling SELECTSHAREDQS (line 15) and then the queries in SSET are evaluated (line 17). Finally, the evaluation of original batch of queries is accelerated by

Algorithm 4 Identify Shared Queries from M

```
1:  Given: High Degree Set $M \subset V$ of Marked Vertices
2:          Vector Visits: Number of Visits of All Vertices $\in M$
3:          Constant #SSET: # of Shared Vertices Selected
4:  Goal: Select #SSET most frequently visited Vertices in $M$
5:  function SELECTSHAREDQS ($M$, Visits, #SSET)
6:      ▷ Init: Set of Source Vertices for Shared Queries
7:      SSET ← $\phi$
8:      ▷ Init: Set of (vertex, vertex visits number) pairs
9:      VERTVISITSPAIRS ← $\phi$
10:     for all $v \in M$ do
11:         VERTVISITSPAIRS←VERTVISITSPAIRS∪{$v$,Visits[$v$]}
12:     end for
13:     ▷ Sort Vertices subject to Number of Visits
14:     Sort ( VERTVISITSPAIRS, moreVisits() )
15:     ▷ Select most frequently visited Marked Vertices
16:     for #SSET top {$v$, Visits[$v$]} ∈ VERTVISITSPAIRS do
17:         SSET ← SSET ∪ {$v$}
18:     end for
19:     return SSET
20: end function
```

updating their results in RESULTT using the results of SSET queries in SHAREDT (line 19). Finally in Phase III the computation of batch queries is resumed and completed in remaining iterations of the while loop. During Phase I the algorithm maintains a count of number of visits to each vertex in M. These counts are used for selecting vertices to form SSET, more visits implies greater relevance to queries in the original batch and hence higher priority for inclusion in SSET. Following the call to PROCESSBATCH in the p^{th} iteration (1^{st} in our experiments), we enter Phase II at which point SSET is built. The details of SSET construction are presented in Algorithm 4. If lines 13-20 are eliminated, the algorithm will not perform sharing and thus its execution will revert to simple batched evaluation.

Function PROCESSBATCH (Algorithm 2) loops over each outedge e of every active vertex, and calls function EDGEFUNCBATCH (Algorithm 3) to attempt update of $e.dest$ by relaxing edge e using conventional edge update function UPDATEFUNC. If the relaxation is successful, i.e. the value of $e.dest$ is changed, $e.dest$ becomes an active vertex for next iteration. Note that function EDGEFUNCBATCH does not blindly relax e for all queries. Instead it looks up CURRTRACK to check which queries activated $e.source$ in the previous iteration, and only attempts update of value of $e.dest$ for corresponding queries. The conventional edge update function UPDATEFUNC for four algorithms is given in Table 2.1. Here CASMIN(a, b) sets $a = b$ if $b < a$ atomically using compare-and-swap); and CASMAX(a, b) sets $a = b$ if $b > a$ atomically using compare-and-swap).

Finally, Algorithm 5 shows how we accelerate the convergence of the solution of the original batch of queries in RESULTT using the results of the shared queries in SHAREDT. Since the cost for looping over all vertices and applying share updates is significant, we limit the number of shared vertices with which each query is used to speed up convergence of property values by choosing a small SSET size. Let us see how the result of a shared query with source vertex r can benefit a batch query given that the reachability is known to be true. Given a vertex d, its value in query Q_i can take advantage of the shared result of subquery on vertex r in SHAREDT and be expressed as follows: SHAREUPDATEFUNC(d, r, Q_i, SHAREDT, RESULTT). The above function for four benchmarks is given in Table 2.2. For example, for SSSP, RESULTT[s_i][r] +

Table 2.1: Conventional Updates for Five Algorithms.

ALG	RESULTT$[s_i][e.dest]$ $\leftarrow$ UPDATEFUNC (e, Q_i, RESULTT)
SSWP	CASMAX(RESULTT$[s_i][e.dest]$, min(RESULTT$[s_i][e.src]$, $e.w$)))
Viterbi	CASMAX(RESULTT$[s_i][e.dest]$, RESULTT$[s_i][e.src]$ / $e.w$)
BFS	CASMIN(RESULTT$[s_i][e.dest]$, RESULTT$[s_i][e.src]$ + 1)
SSSP	CASMIN(RESULTT$[s_i][e.dest]$, RESULTT$[s_i][e.src]$ + $e.w$)
TopkSSSP	KSMALLEST($\{$RESULTT$[s_i][e.dest]\} \cup \{$RESULTT$[s_i][e.src]$ + $e.w\}$)

Table 2.2: Directed Graphs: SHAREUPDATEFUNC for Five Algorithms.

ALG	RESULTT$[s_i][d]$ $\leftarrow$ SHAREUPDATEFUNC(d,r,Q_i, SHAREDT,RESULTT)
SSWP	CASMAX(RESULTT$[s_i][d]$, min(RESULTT$[s_i][r]$, SHAREDT$[r][d]$))
Viterbi	CASMAX(RESULTT$[s_i][d]$, RESULTT$[s_i][r]$ * SHAREDT$[r][d]$)
BFS	CASMIN(RESULTT$[s_i][d]$, RESULTT$[s_i][r]$ + SHAREDT$[r][d]$)
SSSP	CASMIN(RESULTT$[s_i][d]$, RESULTT$[s_i][r]$ + SHAREDT$[r][d]$)
TopkSSSP	KSMALLEST($\{$RESULTT$[s_i][e.dest]\} \cup \{$RESULTT$[s_i][r]$ + SHAREDT$[r][d]\}$)

SHAREDT$[r][d]$ is a safe approximation of the shortest path value from source vertex of q_i to d via r, and we can use the estimation to accelerate the convergence of the value of d.

For undirected graphs, when applying update using result of shared queries, we can benefit from a more accurate approximation of the property value from source vertex to shared vertex. Take SSSP as an example. Given an undirected graph, SHAREDT$[r][s_i]$ can be used as the accurate measurement of the distance from s_i to r. Compared with RESULTT$[s_i][r]$ used in Table 2.2, which is an approximation value, SHAREDT$[r][s_i]$ can be used to compute a better estimation of the distance between s_i and d and therefore give better acceleration on the evaluation of the original batch of queries.

Algorithm 5 Accelerate Batch Queries Using Shared Queries From SSET

```
function SHAREUPDATEBATCH (SSET, SHAREDT, RESULTT)
    for all $Q_i(s_i) \in$ QUERYBATCH do
        for $r \in$ SSET do
            ▷ Update using $r$ only if $r$ is reachable from $s_i$
            if RESULTT$[s_i][r] \neq -1$ then
                for $d \in$ ALLVERTICES do
                    ▷ Attempt Update if $d$ is reachable from $r$
                    if SHAREDT$[r][d] \neq -1$ then
                        ▷ Update $d$ for Query $i$ using $r$
                        SHAREUPDATEFUNC ( $d$, $r$, $Q_i$, SHAREDT, RESULTT )
                    end if
                end for
            end if
        end for
    end for
end function
```

2.1.3 Applicability

Our sharing algorithm can be applied to batched iterative graph algorithms where each query in the batch begins at single source vertex and the property values from these sources to all other vertices are computed. Sharing of results of subqueries is effective because they represent overlapping subcomputations. Graph problems with dynamic programming solutions have the opportunity to benefit from our sharing algorithm because of the optimal substructure property of dynamic programming. Examples include monotonic computations like SSWP, Viterbi, TopkSSSP, and BFS used in our evaluation as well as other non-monotonic algorithms like Personalized Page Rank (PPR) [13] used by recommender services like twitter and Single-Source SimRank (SimRank) [16] queries that are evaluated to compute similarities of graph nodes. It does not apply to algorithms with a global solution, i.e. not originating at source-vertex (e.g., Connected Components). Sharing will work less effectively for local queries like 2-Hop queries due to low overlap between them; however, local queries are inexpensive and can be processed efficiently with batching alone. Sharing works well on power-law graphs as they contain high centrality nodes but it is less effective for high-diameter graphs like road-networks. Only when source vertices are in proximity of each other can there be significant reuse in high-diameter graphs.

2.2 Experimental Evaluation

2.2.1 Experimental Setup

For evaluation we implemented our SimGQ framework using Ligra [41] which uses the Bulk Synchronous Model [45] and provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. We evaluate our techniques for evaluation of batches of queries using four benchmark applications (SSWP – Single Source Widest Path, Viterbi [22], BFS – Breadth First Search, and TopkSSSP – Top k Single Source Shortest Paths). We used four

Table 2.3: Input graphs used in experiments.

Graphs	#Edges	#Vertices
Twitter (TT) [5]	2.0B	52.6M
Twitter (TTW) [20]	1.5B	41.7M
LiveJournal (LJ) [3]	69M	4.8M
PokeC (PK) [42]	31M	1.6M

Table 2.4: BASELINE – Total Execution Times for Evaluating Randomly Selected Queries One by One in `Seconds` on the Ligra [41] System. For first 3 benchmarks 512 queries are used and for TopkSSSP we use 64 queries.

Graph	SSWP	Viterbi	BFS	Top 2 & 1 SSSP	
TTW	2,989s	3,737s	2,574s	4,073s	2,337s
TT	3,949s	4,902s	3,538s	2,768s	1,574s
LJ	134s	258s	102s	389s	226s
PK	63s	116s	55s	232s	123s

real world power-law graphs shown in Table 4.3 in these experiments – TT [5] and TTW [20] are large graphs with 2.0 and 1.5 billion edges respectively; and LJ [3] and PK [42] are smaller graphs with 69 and 31 million edges respectively. Benchmarks are implemented using the PUSH model on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 512 GB memory, and running CentOS Linux 7.

For each combination of benchmark application and input graph, we used 512 randomly generated queries to carry out the evaluation, except for TopkSSSP for which we use 64 queries because of runtime cost. The baseline total execution times when the queries are evaluated one by one is given in Table 2.4. Because TTW and TT are far bigger in size than LJ and PK, the execution times for TT and TTW are higher.

2.2.2 Benefits of Sharing and Batching

In this section we present the results of our algorithm, we refer to them as Batch+Share. In addition, we also collect execution times of algorithm that uses batching but no sharing, we refer to this algorithm as Batch. Since the batch size is an important parameter in this evaluation, we vary batch sizes from 4 queries to a very large number of 512 queries. For TTW and TT the maximum batch size was limited to 256 because our machine did not have sufficient memory to run 512 queries for very large graphs. For TopkSSSP maximum batch size is limited to 64 due to its high runtimes.

The results of running the above algorithms are presented in Table 2.5 and Figure 2.2. While Table 2.5 presents the total execution times for 512 queries for batch sizes (number in parentheses) that yielded the highest speedup for each of the algorithms, Figure 2.2 presents average per query execution times for all batch sizes for TT the largest graph.

The data in Table 2.5 shows that our algorithms yield speedups of up to 45.67× over the baseline that executes the queries one by one using the state of the art Ligra system. For the first two benchmarks of SSWP and Viterbi the Batch+Share algorithm delivers speedups ranging from 22.11× to 45.67×. In contrast, for the last two benchmarks of BFS and TopkSSSP the highest speedups observed range from 1.53× to 6.63×.

The sharing algorithm is more profitable if the result values of queries fall in a narrow range and hence often overlap. Like the result of SSWP query is usually an integer between 17 and 25, and the answer of Viterbi is between 0 and 1. In these cases, sharing produces lots of stable values and reduces the number of iterations because vertices made stable by sharing will never be activated again. Sharing is also effective when the vertex update function is expensive even if it produces few stable values – TopkSSSP is a representative graph algorithm from this category. Here sharing reduces the number of updates by 34% but produces few stable values. BFS does not fall into any of these two categories and thus, as expected, does not benefit much from sharing.

Let us consider results in Figures 2.2 which present the *average per query execution times* for varying batch sizes. The trends for the first three benchmarks show that performance continues to improve with increasing batch sizes. For Batch the improvement is due to greater amortization of runtime overheads while for Batch+Share the improvement is greater due to additional benefits of sharing. Further, we observe that on our machine, once we cross the batch size of 64, the improve-

Table 2.5: Best Batching+Sharing and Batching Execution Times in Seconds for all Queries and Corresponding (Batch Sizes) and Speedup Over No-Batching Baseline times from Table 2.4.

Algorithm	SSWP (512 Queries)			Viterbi (512 Queries)			BFS (512 Queries)		
	TTW								
Batch+Share	71	(256)	42.37×	86	(256)	43.42×	440	(128)	5.84×
Batch	629	(256)	4.75×	729	(256)	5.13×	388	(256)	6.63×
	TT								
Batch+Share	90	(256)	43.96×	107	(256)	45.67×	723	(128)	4.90×
Batch	1034	(64)	3.82×	1274	(64)	3.85×	692	(128)	5.12×
	LJ								
Batch+Share	6	(512)	22.11×	12	(128)	22.27×	23	(256)	4.36×
Batch	37	(256)	3.63×	59	(256)	4.34×	18	(256)	5.63×
	PK								
Batch+Share	2	(512)	28.38×	4	(128)	28.97×	11	(512)	5.01×
Batch	20	(512)	3.24×	30	(256)	3.89×	9	(512)	6.40×

Algorithm	Top 2 & 1 SSSP (64 Queries)					
	TTW					
Batch+Share	2671	1260	(32)	(32)	1.53×	1.86×
Batch	3652	1876	(32)	(8)	1.12×	1.25×
	TT					
Batch+Share	1605	858	(8)	(8)	1.73×	1.84×
Batch	2768	1574	(1)	(1)	1.00×	1.00×
	LJ					
Batch+Share	237	135	(64)	(64)	1.64×	1.67×
Batch	375	190	(32)	(32)	1.04×	1.19×
	PK					
Batch+Share	119	58	(64)	(64)	1.95×	2.13×
Batch	196	98	(16)	(16)	1.19×	1.26×

ments in performance are relatively small although the best performances reported in Table 2.5 are for batch sizes of 256 and 512 for majority of the cases (i.e., different graphs and benchmarks). Based upon the trends observed in Figure 2.2, for a larger machine with more memory and number of cores, performance can be expected to scale further with batch size. For TopkSSSP while there is less variation with batch size the difference between Batch and Batch+Share is substantial.

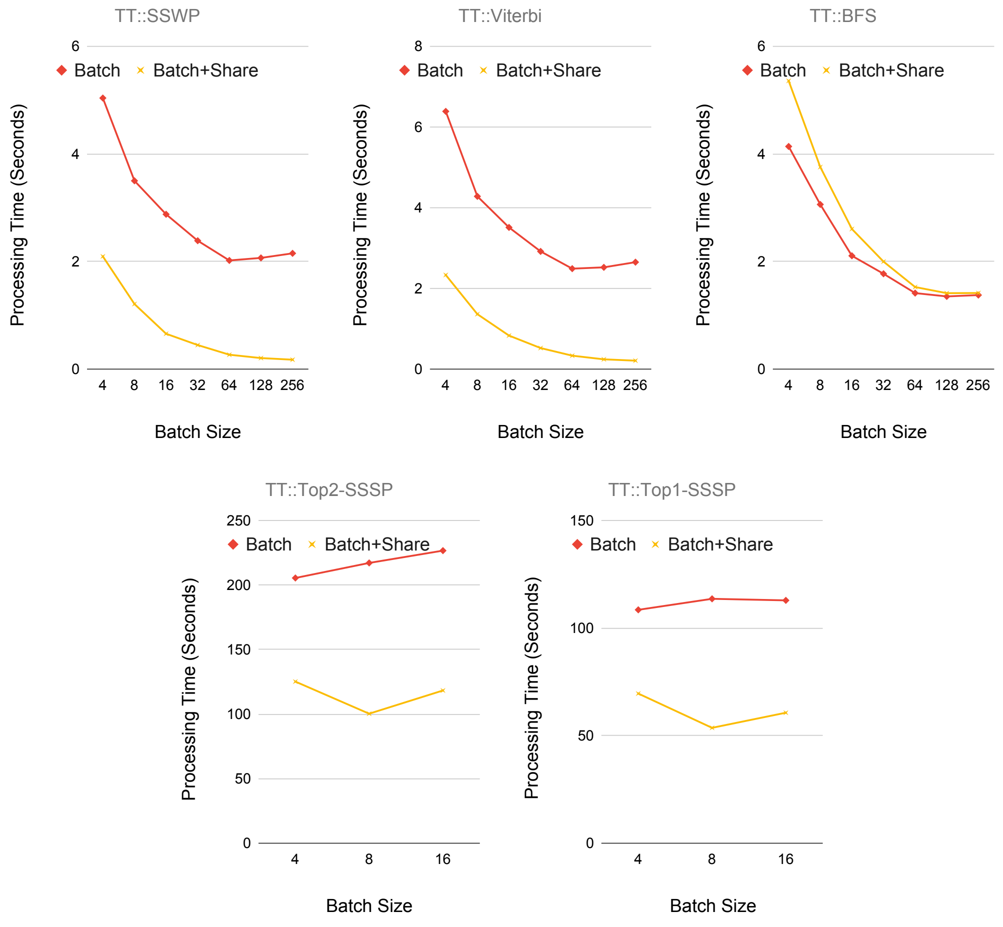


Figure 2.2: Average Per Query Execution Times of Batch vs. Batch+Share.

Table 2.6: Batch+Share Over Batch Alone: Cost of Phase II, Benefit of Phase II, Speedup Due to Batch+Share Over Batch Alone. Speedups computed for best Batch+Share configurations for all Queries.

	SSWP (512 queries)			Viterbi (512 Queries)			BFS (512 Queries)	
TTW								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.08	0.97	8.92×	0.09	0.97	8.47×	0.15	0.07	0.93×
TT								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.06	0.98	12.26×	0.06	0.98	12.64×	0.11	0.07	0.96×
LJ								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.12	0.96	6.43×	0.12	0.92	5.14×	0.26	-0.03	0.77×
PK								
Cost	Benefit	Speedup	Cost	Benefit	Speedup	Cost	Benefit	Speedup
0.09	0.98	8.76×	0.08	0.96	8.22×	0.20	-0.08	0.78×

Top 2 & 1 SSSP (64 Queries)					
TTW					
Cost		Benefit		Speedup	
0.04	0.04	0.31	0.41	1.37×	1.60×
TT					
Cost		Benefit		Speedup	
0.09	0.09	0.63	0.62	2.16×	2.12×
LJ					
Cost		Benefit		Speedup	
0.02	0.02	0.43	0.39	1.69×	1.58×
PK					
Cost		Benefit		Speedup	
0.02	0.02	0.50	0.50	1.94×	1.94×

2.2.3 Contributions of Sharing vs. Batching

We observed that for SSWP and Viterbi both sharing and batching are responsible for delivering high performance while for TopkSSSP batching does not provide benefit, and for BFS sharing does not deliver additional performance improvement. We analyze the cost and benefit of sharing to show that for first three benchmarks the benefit far outweighs the cost while for BFS the benefit is smaller than the cost incurred.

Table 2.7: Factoring Speedups: Batching $\times$ Sharing = Total Speedup.

SSWP	Viterbi
TTW	
4.75 $\times$ 8.92 = 42.37$\times$	5.13 $\times$ 8.47 = 43.42$\times$
TT	
3.58 $\times$ 12.26 = 43.96$\times$	3.61 $\times$ 12.64 = 45.67$\times$
LJ	
3.44 $\times$ 6.43 = 22.11$\times$	4.33 $\times$ 5.14 = 22.27$\times$
PK	
3.24 $\times$ 8.76 = 28.38$\times$	3.52 $\times$ 8.22 = 28.97$\times$

Using the execution times of Batch, which is essentially a shared-memory version of MultiLyra, as baseline, Table 2.6 presents the speedups achieved by Batch+Share. As we can see from the results, for benchmarks of SSWP and Viterbi, the speedups range from 5.14$\times$ to 12.64$\times$ demonstrating that sharing delivers substantial additional speedups over batching alone for these benchmarks. For benchmark of TopkSSSP, the benefit from sharing is less, but there are still descent speedups of up to 2.16$\times$ due to sharing. On the other hand, for benchmark of BFS there is even some slowdown.

The Cost and Benefit of sharing are also shown explaining the above results. The Cost is the time spent in Phase II while Benefit is reduction in total time spent on Phase I + Phase III due to sharing based updates performed by Phase II. Both the Cost and Benefit are presented as fraction of execution times of corresponding Batch algorithms. Thus, the Speedups are related to the Cost and Benefit as follows: $Speedup = 1/(1 + Cost - Benefit)$. For SSWP, Viterbi, and TopkSSSP, the Benefit far exceeds the Cost while for BFS, the Cost exceeds the Benefit hence the observed speedup results. Finally, Table 2.7 summarizes how the overall speedups achieved for SSWP and Viterbi can be *factored* between batching and sharing showing the importance of employing both batching and sharing techniques.

Table 2.8: Percentage of Vertex Values that become **Stable** due to Sharing Updates.

Graph	Batch Sizes	SSWP	Viterbi	BFS	Top 2 & 1 SSSP
TTW	4	99.99	99.99	28.95	6.93 - 6.93
	8	99.99	99.99	25.14	7.38 - 7.41
	16	99.99	99.99	22.07	5.21 - 5.25
TT	4	99.99	99.99	23.71	16.65 - 16.78
	8	99.99	99.99	20.57	19.85 - 20.03
	16	99.99	99.99	18.14	13.61 - 13.82
LJ	4	99.99	99.64	7.64	2.21 - 1.03
	8	99.99	99.64	6.56	2.01 - 1.20
	16	99.99	99.64	5.61	2.53 - 1.40
PK	4	99.99	99.63	6.26	0.88 - 1.92
	8	99.99	99.63	6.11	1.01 - 2.03
	16	99.99	99.63	5.38	3.24 - 4.35
Average		99.99	99.80	12.87	6.10 - 6.18

The cost of sharing is reasonable because overheads of sharing come from three sources and all of them are low. First, we need to maintain a counter of the number of visits for each marked high degree vertex in Phase I. This overhead is negligible because we only mark a very small amount of high degree vertices (e.g., 100 out of millions in the current setting) and Phase I is very short (e.g., 1 iteration) and thus has relatively small frontier sizes. Second, we need to solve the shared queries in Phase II. Given that it only computes a small number of shared queries (e.g., only 5 from 100) while the batch size for original queries can be much larger (up to 512), the cost is amortized well across all queries in a batch and thus it has little impact on each individual query. Third, we introduce extra computation cost when applying the result of shared queries to accelerate the convergence of original query. Since this step is a linear scan of the array, it leads to better cache performance due to spatial locality compared with the usual updates for a query which can be randomly scattered across the value array in Ligra. Besides, our sharing algorithm only allows each query to reuse the result of one shared query and only once, keeping the reuse cost low.

To better understand the effectiveness of sharing, we also collected the *stable value* percentages – this is the percentage of vertices reachable from the source vertex whose vertex values converge as a result of performing share updates. We collected this data for the Batch+Share configuration. Since we pause the original computation only after the first iteration (i.e., $p = 1$), the percentage of vertices that are stable prior to sharing updates is negligible (less than 0.01%). The percentages of values that are stable following sharing updates are presented in Table 2.8. As shown in the table, sharing greatly benefits SSWP and Viterbi as it causes nearly all the values ($> 99\%$) to converge. To explain the phenomenon that Top2SSSP and Top1SSSP has lower stable percentage than BFS but sharing delivers much more speedups for the former than the latter, we collected the reduction in number of vertex updates resulting from sharing. It turns out that the reduction for Top2SSSP and Top1SSSP (34%) is much higher than the reduction for BFS (7%).

2.2.4 Sensitivity of SimGQ Performance to the p Value

All our preceding experiments were performed for **p** value of **1**, i.e. Phase I lasted one iteration following which Phase II was performed and then the updates from Phase II results optimized the remainder of time spent in Phase III till convergence. We varied the **p** value from 1 to 3 and compared the speedups that were obtained by sharing over batching alone. The results in Table 2.9 show that **p** value 1 delivers best overall speedups and the trend is that speedup falls as **p** value is increased. The only exceptions are LJ::Viterbi and PK::Viterbi where **p** value of 2 slightly outperforms **p** value of 1 (5.48× v.s. 5.14×, 8.57× v.s. 8.22×). There is a performance tradeoff in selecting **p** value. A smaller **p** enables an earlier reuse which leads to earlier convergence of queries. However, if **p** is small, limited number of marked high degree vertices may be visited and considered as candidates for sharing. We conclude the following from this experiment. First, executing Phase I for one iteration is sufficient as high quality SSET nodes have already been encountered. Second, executing Phase II early has the added benefit that greater fraction of overall iterations is optimized by the updates performed from the results of Phase II. We observe that **p** value of 1 causes sharing to deliver much higher speedups than **p** value of 2 for SSWP and Viterbi on large graphs than small graphs. For example, for the TT graph on Viterbi benchmark, the speedup over batching alone for **p** value of 1 is 12.64× while for the second best **p** value of 2, is much smaller 6.36×.

Table 2.9: Sensitivity to **p** Value: Cost of Phase II, Benefit of Phase II, Speedup of Sharing Over Batching Alone on 256 Queries.

p	SSWP			Viterbi		
	TTW					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.08	0.97	8.92×	0.09	0.97	8.47×
2	0.08	0.81	3.74×	0.07	0.84	4.21×
3	0.08	0.49	1.70×	0.08	0.50	1.72×
	TT					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.06	0.98	12.26×	0.06	0.98	12.64×
2	0.06	0.88	5.61×	0.05	0.89	6.39×
3	0.06	0.57	2.05×	0.06	0.59	2.17×
	LJ					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.12	0.96	6.43×	0.12	0.92	5.14×
2	0.13	0.94	5.26×	0.09	0.91	5.48×
3	0.12	0.88	4.19×	0.10	0.85	3.89×
	PK					
	Cost	Benefit	Speedup	Cost	Benefit	Speedup
1	0.09	0.98	8.76×	0.08	0.96	8.22×
2	0.09	0.95	7.25×	0.07	0.95	8.57×
3	0.09	0.83	3.91×	0.08	0.87	4.68×

2.2.5 Dynamic Selection of SSET

One of the key characteristics of our algorithm is that the vertices in SSET are selected dynamically during the evaluation of a batch of queries. This has two main advantages. First, the selection of SSET vertices is customized to the batch of queries being evaluated. This is important that different batches may contain queries that are close to, in terms of number of hops, different high degree vertices and selection of closer high degree vertices offers greater opportunities of sharing. Second, our technique can be used to speedup the evaluation even when only a single batch of queries is to be evaluated. Note that alternatively techniques can be devised to profile executions of batches to identify SSET vertices and then use them to implement sharing in future batches. However, such an approach would lose both of the advantages of our approach mentioned above.

We next confirm that dynamic custom selection of SSET vertices for each batch does indeed lead to selection of different high degree vertices which deliver better speedups. We performed an experiment in which we split 256 queries for the two large graphs TTW and TT into four batches of 64 queries each. We identified the SSET vertices using the first batch and used it to perform sharing in the other three batches. Table 2.10 presents batch running time as follows: time using a single dynamically selected SSET vertex for the batch → time using a single dynamically selected SSET

Table 2.10: Changes in Batch Execution Time (seconds):
Dynamically Selected → From Other Batch.

Graph::Alg.	Batch 2	Batch 3	Batch 4
TTW::SSWP	14.1 → 14.4	12.9 → 14.1	12.3 → 13.3
TTW::Viterbi	15.1 → 16.4	13.4 → 14.5	14.4 → 14.4
TT::SSWP	17.5 → 18.6	16.2 → 17.5	16.2 → 17.3
TT::Viterbi	18.6 → 18.5	17.5 → 17.6	17.1 → 17.3

Table 2.11: Number of Unique Shared Vertices Selected Over Four Batches:
Min < Actual < Max

Graph::Alg.	$\|SSET\| = 1$	$\|SSET\| = 3$	$\|SSET\| = 5$
TTW::SSWP	1 <3 <4	3 <7 <12	5 <9 <20
TTW::Viterbi	1 <3 <4	3 <7 <12	5 <9 <20
TT::SSWP	1 <2 <4	3 <8 <12	5 <9 <20
TT::Viterbi	1 <2 <4	3 <8 <12	5 <9 <20

vertex in the first batch. The results show that for TTW::SSWP, TTW::Viterbi, and TT::SSWP custom/dynamic selection of SSET vertices for the last three batches delivers better performance (i.e., lower execution times) than the speedups that result from using SSET vertices identified using the first batch. For TTW::Viterbi batches 1 and 4 selected the same vertex and hence there is no change in execution time. For TT::Viterbi the nodes selected give nearly the same performance.

Finally, we examined the identities of selected SSET vertices for various batches to study the diversity of SSET vertices. In Table 2.11 we present actual number of distinct vertices included in SSETs versus the minimum number (size of SSET) and maximum number (number of batches $\times$ the size of SSET) of distinct vertices that can be observed. We found that the number of distinct SSET vertices selected are well above the minimum, i.e. during evaluation of different batches often different vertices are selected as SSET vertices.

2.3 Summary

In this chapter, we developed techniques for simultaneous evaluation of large batches of iterative point-to-all graph queries. By employing batching, the overhead costs of query evaluation are amortized across the queries. By employing sharing, the cost of computations involving shared queries are amortized across the original batch of queries. Our experiments based upon the state of the art Ligra system yielded speedups ranging from 1.53$\times$ to 45.67$\times$ across four input graphs and four benchmarks. In the next chapter, we will discuss how to adapt the batching principle and sharing technique to efficiently process point-to-all queries with arbitrary source vertices in the streaming graph scenarios.

Chapter 3

Leveraging Batching and Sharing for the Streaming Graph Scenario

Several streaming graph systems have been proposed recently with supports for incremental evaluation of iterative graph queries such as Kineograph [7], Tornado [43], Naiad [33], KickStarter [48], Graphbolt [27]. The core of these systems is to leverage incremental computation to quickly update the results of fixed queries in streaming graph scenarios, as illustrated by the figure 3.1. In the streaming graph scenarios, usually the updates are streamed to the graph continuously and they are commonly grouped into batches for more efficient digestion. After a batch of updates (e.g., edge insertions) is applied to the graph, incremental computation reuses the query result on the previous version of the graph to initialize the vertex values, and thus performs iterative computation starting from a point closer to convergence compared to evaluating the query from scratch.

However, existing incremental computation has a fundamental limitation - it requires a priori knowledge of the user query. In order to benefit from incremental computation, the query

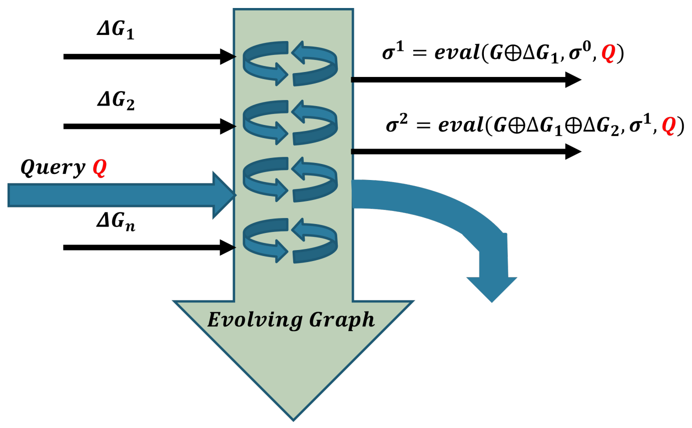

ΔG_1
ΔG_2
Query Q
ΔG_n
$\sigma^1 = eval(G \oplus \Delta G_1, \sigma^0, Q)$
$\sigma^2 = eval(G \oplus \Delta G_1 \oplus \Delta G_2, \sigma^1, Q)$
Evolving Graph

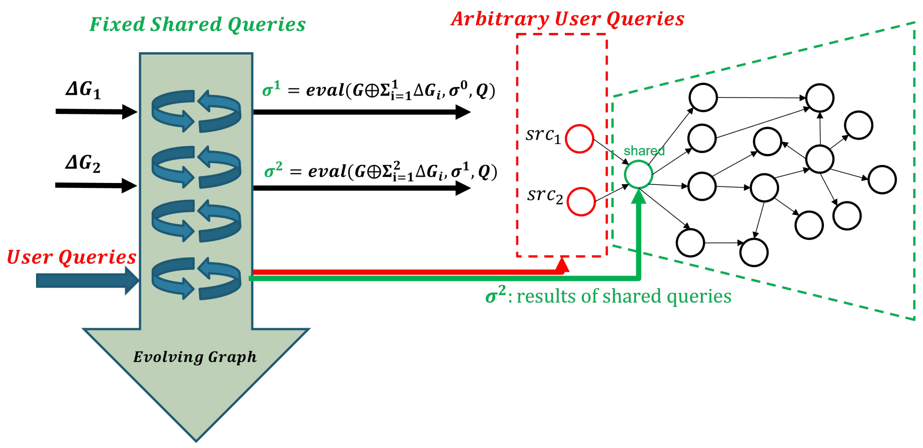
Figure 3.1: Incremental Computation

under evaluation must remain the same to ensure the safety to reuse the query results on a previous version of the graph. Any queries, other than the fixed standing queries maintained by incremental computation, must be reevaluated from scratch with the trivial initialization values for all the vertices in the graph. While this is not an issue for graph algorithms with a global solution such as PageRank and Connected Components, it prevents incremental computation from accelerating arbitrary user queries for vertex graph algorithms originating from a single source vertex such as single-source shortest path (SSSP) and breadth-first search (BFS).

In this chapter, we adapt the sharing and batching proposed in SimGQ to generalize incremental computation to handle batches of *arbitrary* vertex queries in the streaming graph scenario. Essentially we continuously apply incremental computation to maintain the results of a small batch of shared query candidates upon each graph mutation. When a batch of arbitrary user queries comes, the results of one of the shared query candidates will be selected to share with all the user queries using the *sharing* optimization in SimGQ. The performance of both the incremental evaluation of shared query candidates and the evaluation of user queries are further improved using the batching technique to achieve a higher throughput.

3.1 SimGQStream: Evaluating a Batch of Queries in Streaming Graphs

3.1.1 Overview of SimGQStream



Fixed Shared Queries
Arbitrary User Queries
ΔG_1
ΔG_2
$\sigma^1 = eval(G \oplus \Sigma_{i=1}^{1} \Delta G_i, \sigma^0, Q)$
$\sigma^2 = eval(G \oplus \Sigma_{i=1}^{2} \Delta G_i, \sigma^1, Q)$
src_1
src_2
shared
User Queries
σ^2: results of shared queries
Evolving Graph

Figure 3.2: Incremental Computation.

Figure 3.2 shows the architecture of our system. At high level, the system consists of two modules: on the left we have a standing loop that processes fixed shared queries and feeds the results to the user query processing module; on the right we have the user query processing module that uses the shared query results generated from the shared query processing module to accelerate a batch of arbitrary user queries.

Shared Query Processing Module

The shared query processing module on the left of Figure 3.2 is responsible for incrementally maintaining the results of a batch of fixed queries whose results are to be shared with other queries originating from any vertex in the graph. Following our observations in SimGQ, we again use high degree vertices as the candidates for source vertices of shared queries because high degree vertices are generally reachable to more vertices in the graph and thus provide a larger coverage when we apply sharing to accelerate the processing of user queries, especially for power-law graphs such as social networks which constitutes a large portion of real graphs.

Since incremental computation requires only a small portion of work and execution time compared with full evaluation from scratch (e.g., incremental evaluation of 16 to 64 queries with a graph mutation of 10K edges is roughly same as a full evaluation of a single query on graph Twitter), now we can afford computing results for more shared query candidates compared with SimGQ in which usually we can only afford evaluating one shared query due to the expensive full evaluation.

User Query Processing Module

On the right of Figure 3.2, we have a module that answers the user queries. When a batch of user queries comes, this module takes a snapshot of the current version of the graph and waits for the shared query processing module to converge the evaluation of shared queries on the snapshot graph if needed. As soon as the stable results of the shared queries become ready, the user query processing module selects one or more shared queries from the standing loop on the left, and then applies the sharing technique to share the shared queries results to fast-forward the input user queries during the initialization stage.

For example, let's take SSSP on an undirected graph as an example. Let $SP(u, v)$ denotes the shortest path distance between vertex u and vertex v. If the user query originates from vertex u and the shared query originates from vertex r, then $SP(u, v)$ can be safely initialized as $SP(u, r) + SP(r, v)$ for every vertex v in the graph. After initialization, we just start the normal iterative graph algorithm (e.g., SSSP) from the source vertices of the user queries until the user queries' results converge.

Where Batched Evaluation Can Help

Since user queries have been grouped into batches before streamed into our system, user queries can be naturally evaluated in a batched manner. In addition, as mentioned before, with the help of incremental computation, now we can afford maintaining the results of more shared query candidates which can be computed together as a batch upon each graph mutation. Therefore, the performance of both user query processing and shared query processing can be further improved by employing the batch processing paradigm introduced in SimGQ.

3.1.2 Push-Style Batch Evaluation With SimGQStream

Now we present a detailed algorithm for generalized incremental computation leveraging both batching and sharing. The shared query processing module is presented in Algorithm 7. The user query processing module is presented in Algorithm 8. The connection and interaction between two subsystem modules is shown in Algorithm 6 which is the entry point of the overall algorithm. The aforementioned three algorithms are configured to run in serial to maximize resource availability for each individual task: graph update, shared query processing, and user query processing, where each of the three is executed in parallel with multithreading support. In the following, we will first discuss Algorithm 6 and then dive into the implementation of the two subsystems respectively.

Algorithm 6 illustrates the high level idea about how our system responds to graph mutations and user query request in the streaming graph scenarios. When the system is launched, the users need to specify two input parameters: the initial input graph G and the number of shared queries k. A larger k value provides more sharing opportunities while a smaller k value value reduces the cost to maintain the shared query results. According to the results of sensitivity experiments, we set the default value of k to 16 to balance the trade-off. The algorithm begins with initialization stage in which the top k highest degree vertices are retrieved as the source vertex for the shared queries, followed by a batched evaluation of these shared queries on the initial version of input graph (lines 4-7). Once the initialization is complete, the algorithm maintains a standing loop (lines 12-22) that monitors two types of signals: graph mutations and new requests for evaluating user queries. If the captured signal is triggered by graph mutation (lines 13-17), which is a batch of edge insertions in our case, the system will create a task for the shared query processing module by calling function PROCESSSHAREDQUERYBATCH (line 15) which will incrementally update shared query results in a batched manner. If the captured signal is triggered by a new request for evaluating user queries (lines 18-21), the system will create a task for the user query processing module by calling function PROCESSUSERQUERYBATCH (line 20) which will leverage the shared query results to accelerate the user query evaluation in a way similar to the Batch+Share algorithm proposed in SimGQ.

Algorithm 7 illustrates once a graph mutation is detected, how our system keeps the shared query results up-to-date with batched incremental evaluation. The algorithm leverages the shared query results on the previous version of the graph (i.e., SHAREDT on G) to accelerate the processing of the same queries on a new version of graph $G \bigoplus DeltaG$ by initializing the the query property values on the new graph with the query results on the previous version of graph (i.e., SHAREDT) which is a state closer to the convergence state compared with trivial initial values and leads to faster convergence. In order to guarantee the correctness of output, the source vertices of inserted edges need to be put into the active lists (lines 7-12). After the initialization, the algorithm iterates until all the property values becomes stable (lines 14-18). Function EDGEMAPBATCH does almost the same work as function PROCESSBATCH in Algorithm 1 – both of them update the property values of active vertices and computes the new coarse-grained and fine-grained active frontiers (i.e., ACTIVE and NEXTTRACK), except that EDGEMAPBATCH does not maintain the counter of visits for the source vertices of shared queries.

Algorithm 8 illustrates once our system detects a new request for evaluating a batch of user queries, how to leverage both sharing and batching to quickly evaluate the coming arbitrary user queries. The algorithm works in two steps. In the first step, function SHAREUPDATEBATCH is called (lines 4-5), so that the property values of the user queries are fast-forwarded during the initialization stage by sharing with all the user queries in the batch the results of the shared queries which have been evaluated and converged on the current version of graph. Since the details of function SHAREUPDATEBATCH has been discussed in Algorithm 5 in the previous chapter, we do not dive into it again. Essentially the connection between user queries and the shared queries are established by triangle-inequality. For example, let us take single-source shortest path (SSSP) algorithm as the sample benchmark. Let u denote the source vertex of a user query, let r denote the source vertex of a shared query, and let $SP(a, b)$ denote the shortest path distance from vertex a to vertex b. $SP(u, v)$ is bounded by the sum $SP(u, r)$ + $SP(r, v)$ where both $SP(u, r)$ and $SP(r, v)$ are known from the shared query results for undirected graphs. Once the initialization step is complete, the algorithm moves on to the second step which is straightforward – iteratively run the graph application in a batched manner until convergence (lines 9-14). The core operations in SHAREUPDATEFUNC for different graph algorithms can be found in Table 2.2 presented in the previous chapter.

Algorithm 6 Generalized Incremental Computation

```
1: Given: Initial version of directed graph $G(V, E)$; Number of shared queries $k$
2: Goal: Maintain shared query results upon graph mutation and respond to user queries

3: function LAUNCH( $G$, $k$ )
4:     ▷ Initialize shared query set
5:     SSET ← SELECTTOPKHIGHDEGREEVERTICES($G$, $k$)
6:     ▷ Evaluate shared queries in SSET on graph $G$ using SimGQ Batch Algorithm w/o sharing
7:     SHAREDT ← EVALUATEBATCH($G$, SSET)
8:     ▷ Initialize graph update batch and user query batch
9:     DELTAG ← $\phi$
10:    USERQUERYBATCH ← $\phi$
11:    ▷ Standing loop that monitors and responds to graph mutations and user query requests
12:    while GETNEWGRAPHUPDATE(&DELTAG) or GETNEWUSERQUERY(&USERQUERYBATCH) do
13:        if DELTAG $\neq \phi$ then
14:            ▷ Case 1: Trigger is a graph mutation; Incrementally update shared query results
15:            SHAREDT ← PROCESSSHAREDQUERYBATCH($G$, DELTAG, SSET, SHAREDT)
16:            $G \leftarrow G \bigoplus$ DELTAG
17:            DELTAG ← $\phi$;
18:        else
19:            ▷ Case 2: Trigger is a new user query request; Evaluate user queries with help of Sharing
20:            RESULTT ← PROCESSUSERQUERYBATCH($G$, USERQUERYBATCH, SSET, SHAREDT)
21:            USERQUERYBATCH ← $\phi$;
22:        end if
23:    end while
24: end function
```

Algorithm 7 Batched Incremental Evaluation of Shared Queries

```
1: Given: Previous version of graph $G(V, E)$; A set of inserted edges $DeltaG$; The set of source vertices of the shared queries SSET; Shared query results on $G$ named SHAREDT
2: Goal: Update SHAREDT to be the shared query results on the new version of the graph $G \bigoplus DeltaG$
3: function PROCESSSHAREDQUERYBATCH( $G$, $DeltaG$, SSET, SHAREDT )
4:     ▷ Initialization
5:     ACTIVE $\leftarrow \phi$;CURRTRACK $\leftarrow \phi$; NEXTTRACK $\leftarrow \phi$
6:     ▷ Initialize active frontiers w.r.t. to the inserted edges
7:     for all inserted edge $(u, v) \in DeltaG$ in parallel do
8:         ACTIVE $\leftarrow$ ACTIVE $\cup \{u\}$
9:         for all SharedQuery(vertex) : $Q_i(s_i) \in$ SSET in parallel do
10:            CURRTRACK $\leftarrow$ CURRTRACK $\cup \{(u, Q_i(s_i))\}$
11:        end forall
12:    end forall
13:    ▷ Iterate till Convergence
14:    while ACTIVE $\neq \phi$ do
15:        ▷ Update shared query results in current iteration
16:        ACTIVE $\leftarrow$ EDGEMAPBATCH ( ACTIVE, CURRTRACK, NEXTTRACK, SHAREDT )
17:        CURRTRACK $\leftarrow$ NEXTTRACK; NEXTTRACK $\leftarrow \phi$;
18:    end while
19:    return SHAREDT
20: end function
```

3.1.3 Applicability

Since the user query processing module of generalized incremental computation relies on the sharing technique in SimGQ to fast forward the property value array during the query initial-

Algorithm 8 Batched Evaluation of User Queries

```
1: Given: Current version of directed graph $G(V, E)$; A batch of user queries named $UserQueryBatch$; The set of
   source vertices of the shared queries SSET; Shared query results on G named SHAREDT
2: Goal: compute and return RESULTT which is the user query results on $G$

3: function PROCESSSHAREDQUERYBATCH( $G$, $DeltaG$, SSET, SHAREDT )
4:     ▷ Initialize User Query Results with Sharing Optimization in SimGQ
5:     SHAREUPDATEBATCH ( SSET, SHAREDT, RESULTT )
6:     ▷ Initialize active frontiers
7:     ACTIVE $\leftarrow$ { $s_1, s_2, ..., s_k$ }; NEXTTRACK $\leftarrow \phi$;
8:     CURRTRACK $\leftarrow$ { $(s_i, Q_i) : Q_i(s_i) \in$ QUERYBATCH }
9:     ▷ Iterate till Convergence
10:    while ACTIVE $\neq \phi$ do
11:        ▷ Update user query results in current iteration
12:        ACTIVE $\leftarrow$ EDGEMAPBATCH ( ACTIVE, CURRTRACK, NEXTTRACK, RESULTT )
13:        CURRTRACK $\leftarrow$ NEXTTRACK; NEXTTRACK $\leftarrow \phi$;
14:    end while
15:    return RESULTT
16: end function
```

ization stage, the prerequisite for sharing optimization (i.e., vertex query and optimal substructure within the graph problem) also applies for generalized incremental computation described in this chapter. More discussion on the applicability of sharing can be found in Section 2.1.3 in Chapter 2. In addition, the shared query processing module in generalized incremental computation requires that the graph algorithm benefits from conventional incremental computation which applies to many common vertex graph algorithms such as SSSP, BFS, and multiple variants of SSSP.

3.2 Experimental Evaluation

3.2.1 Experimental Setup and Evaluation Methodology

Given that the architecture of SimGQ [56] is close to meeting our requirement for user query evaluation, we implemented the user query processing module by modifying the SimGQ implementation based on Ligra [41] which uses the Bulk synchronous Model [45] and provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. However, since Ligra is designed for processing static graphs rather than streaming graphs, we adopt a state-of-the-art streaming graph engine called Aspen [9] to ingest graph mutations and keep the graph up-to-date in the streaming graph settings. We also extend Aspen to incrementally maintain the results of a group of fixed shared query while the graph is changing.

The evaluation is conducted using four benchmark applications (SSWP – Single Source Widest Path, Viterbi [22], BFS – Breadth First Search, and SSSP - Single Source Shortest Paths). Benchmarks are implemented using the PUSH model on a machine with 32 cores (2 sockets, each with 16 cores) with Intel Xeon Processor E5-2683 v4 processors, 512 GB memory, and running CentOS Linux 7.

We used two real world power-law graphs shown in Table 4.3 in these experiments – LJ [3] and PK [42] with 69 and 31 million edges respectively. For each input graph, we used 512 randomly generated user queries to carry out the evaluation.

The performance of the shared query processing module is evaluated on the same set of four benchmark applications and two input graphs as in the evaluation of the user query processing module. The number of shared queries is set to 16 by default, because according to our prior study in SimGQ, 16 shared queries is sufficient to provide a decent performance benefit on user queries via sharing. To study the sensitivity of shared query performance to graph mutation, we vary the graph update batch size (number of edge insertions) from one thousand (1K) to one million (1M).

3.2.2 Evaluation of User Query Processing Module

In this section, we present the evaluation of user queries. The effectiveness of the user query processing module is measured by the speedup of generalized incremental computation over the baseline that answers user queries in a batched manner but without sharing (i.e., the Batch algorithm in SimGQ). Table 3.1 shows the baseline execution times (averaged per query running time for 512 user queries) with various query batch sizes ranging from 1 to 256. For all the eight combinations of input graphs and benchmark applications, the general trend is the same – the running time decreases as query batch size increases, since the runtime overhead such as access to the edge list are better amortized with larger batch sizes. The improvement of performance from a larger batch size becomes reasonably less significant once the query batch sizes reach a certain threshold (e.g., batch size 64 for SSWP on LJ).

Table 3.3 shows the speedups of the generalized incremental computation of user queries over the non-incremental evaluation baselines with different query batch sizes. As we can see, the speedups fall into a wide range for different benchmarks. Generalized incremental computation provides more speedups for SSWP (20.03x-71.87x, average at 38.06x) and Viterbi (10.45x-78.19x, average at 33.93x) compared with the case for BFS (1.01x -1.26x, average at 1.14x) and SSSP (1.01x-2.40x, average at 1.33x). The divergence is due to the fact that different graph algorithms have different update functions and different property value types. After the initialization stage that accelerates the user query results by sharing the results of the shared queries, different benchmarks

Table 3.1: BASELINE – Averaged Per Query Execution Times for Evaluating Randomly Selected User Queries One by One in `Seconds` on the Ligra [41] System. 512 queries are used.

G	Batch Sizes	SSWP	Viterbi	BFS	SSSP
LJ	1	0.30	0.58	0.18	0.46
	2	0.22	0.38	0.13	0.33
	4	0.16	0.32	0.13	0.26
	8	0.16	0.27	0.10	0.17
	16	0.11	0.20	0.08	0.18
	32	0.09	0.15	0.05	0.11
	64	0.09	0.13	0.05	0.10
	128	0.08	0.12	0.05	0.10
	256	0.08	0.12	0.05	0.09
PK	1	0.14	0.23	0.08	0.20
	2	0.11	0.16	0.06	0.14
	4	0.08	0.13	0.05	0.09
	8	0.08	0.11	0.05	0.10
	16	0.07	0.08	0.04	0.08
	32	0.06	0.08	0.04	0.07
	64	0.05	0.07	0.03	0.06
	128	0.05	0.06	0.03	0.05
	256	0.04	0.06	0.03	0.05

receive diverged property value stable rates which is defined as the ratio of the number of vertices that receive their converged value from initialization via sharing over the total number of vertices in the graph that will have non-trivial values at the end of iterative computation. As shown in Table 3.2, the stable rates from sharing are very high for SSWP and Viterbi (both are more than 99%) and lower for BFS (20%-23%) and SSSP (7%-13%). More detailed discussion for this divergence of benefits from sharing can be found in Section 2.2.2 in Chapter 2.

From Table 3.3, we also observe that, in general, the speedup from incremental computation decreases as the query batch size increases. The trend is more clear for SSWP and Viterbi for which sharing is more effective and leads to very high stable value rates. For SSWP and Viterbi, since sharing can effectively prune out substantial computation during the initialization stage, gen-

Table 3.2: Stable Rates from Initialization via Sharing in Generalized Incremental Computation for Evaluating Randomly Selected 512 User Queries

G	Batch Sizes	SSWP	Viterbi	BFS	SSSP
LJ	4	100.00	99.64	22.82	9.97
	8	100.00	99.64	20.98	8.94
	16	100.00	99.64	21.06	7.25
	32	100.00	99.64	19.91	7.42
	64	100.00	99.64	21.75	7.12
	128	100.00	99.64	21.73	7.04
	256	100.00	99.64	21.73	8.81
PK	4	100.00	99.63	23.32	13.17
	8	100.00	99.63	20.60	12.38
	16	100.00	99.63	19.93	11.60
	32	100.00	99.63	20.06	11.40
	64	100.00	99.63	20.22	11.13
	128	100.00	99.63	20.22	11.25
	256	100.00	99.63	20.22	11.25
Average		**100.00**	**99.64**	**21.04**	**9.91**

Table 3.3: Speedup of Generalized Incremental Computation vs. Non-Incremental Baseline for Evaluating Randomly Selected 512 User Queries

G	Batch Sizes	SSWP	Viterbi	BFS	SSSP
LJ	1	49.60	73.63	1.23	2.15
	2	38.51	39.66	1.14	1.50
	4	29.65	36.76	1.26	1.33
	8	35.53	31.10	1.21	1.24
	16	24.77	22.99	1.23	1.16
	32	25.85	18.96	1.04	1.09
	64	24.25	12.53	1.08	1.13
	128	20.11	11.42	1.06	1.03
	256	20.03	10.45	1.07	1.07
PK	1	71.87	78.19	1.23	2.40
	2	68.72	59.26	1.25	1.51
	4	47.30	47.65	1.22	1.01
	8	39.24	41.77	1.14	1.28
	16	38.83	37.20	1.15	1.49
	32	48.29	30.59	1.17	1.35
	64	37.21	23.12	1.05	1.08
	128	34.01	20.17	1.07	1.07
	256	31.34	15.35	1.01	1.11
Average		**38.06**	**33.93**	**1.14**	**1.33**

eralized incremental computation has less work to do during the iterative computation after initialization stage and thus benefits less from batched evaluation and resource sharing compared with its non-incremental counterpart.

3.2.3 Evaluation of Shared Query Processing Module

In this section, we present the experimental results for shared queries. As aforementioned, we maintain the results of 16 shared queries during graph streaming. We study the following four variants of algorithms for shared query evaluation with various graph update batch sizes (i.e., number of inserted edges) ranging from a small number of 1K to a large number of 1M.

- Baseline: We run queries one by one without incremental computation.
- Inc-Only: We run queries one by one with incremental computation.
- Batch-Only: We group queries into one batch and run it in batching mode.
- Batch+Inc: We employ both batching and incremental computation simultaneously.

The per query baseline execution times in seconds are presented in Table 3.4. The data in Table 3.4 is generated as follows. First, for each combination of input graph, benchmark, and graph update batch size, we compute the average running time per query. And then we aggregate all the data points for the same combination of input graph and benchmark. In general, Viterbi is the most expensive benchmark, because floating points property values take more memory and its update operation is more costful while BFS runs fastest because of the uniform edge weight.

Table 3.5 presents the aggregated speedups of Inc-Only, Batch-Only, and Batch+Inc over the baseline execution times. The data in each cell in Table 3.5 is aggregated from the individual

Table 3.4: Baseline Per Query Running Time for 16 Shared Queries in Seconds. The data in each cell is represented as AVG(MIN-MAX).

G	SSWP	Viterbi	BFS	SSSP
LJ	0.22 (0.22 - 0.23)	0.31 (0.31 - 0.32)	0.09 (0.08 - 0.09)	0.2 (0.19 - 0.2)
PK	0.46 (0.45 - 0.47)	0.58 (0.57 - 0.59)	0.44 (0.43 - 0.45)	0.22 (0.21 - 0.23)

Table 3.5: Speedup of Batch-Only, Inc-Only, and Batch+Inc over Baseline. The data in each cell is represented as AVG (MIN-MAX).

G	Alg.	Batch-Only	Inc-Only	Batch+Inc
LJ	**SSWP**	3.68x (3.54x - 3.85x)	6.18x (3.01x - 10.56x)	38.25x (17.38x - 72.2x)
	Viterbi	3.81x (3.61x - 3.97x)	7.07x (3x - 11.85x)	43.93x (17.27x - 77.3x)
	BFS	3.08x (2.94x - 3.21x)	3.37x (1.83x - 5.15x)	19.33x (8.15x - 36.29x)
	SSSP	3.31x (3.17x - 3.5x)	5.37x (2.61x - 9.12x)	29.02x (10.87x - 50.51x)
PK	**SSWP**	3.15x (3x - 3.32x)	6.8x (3.01x - 12.07x)	42.23x (17.4x - 75.52x)
	Viterbi	3.21x (3.06x - 3.37x)	8.37x (3.35x - 17.48x)	58.98x (19.74x - 130.79x)
	BFS	3.05x (2.9x - 3.3x)	3.26x (1.31x - 6.3x)	17.66x (4.93x - 39.25x)
	SSSP	3.02x (2.86x - 3.31x)	4.98x (1.96x - 9.88x)	26.04x (5.6x - 62.62x)

data for each graph update batch sizes ranging from 1K to 1M. As we can see, Batch-Only provides stable speedup across all graphs and benchmarks with the averaged speedup ranging from 3.02x to 3.87x. Inc-Only provides averaged speedups from 3.26x to 7.07x. The performance of Inc-Only depends on the benchmark where Viterbi benefits most, BFS benefits least, and SSWP and SSSP sit in the middle. Batch+Inc delivers much better performance improvement compared with Batch-Only and Inc-Only, with speedups ranging from 17.66x to 58.98x.

We also studied the impact of the graph update batch size on the performance of shared query processing. The sensitivity of the speedups to graph update batch size on LiveJournal are illustrated in Figure 3.3. The trend for PokeC is the same and thus not presented here to save space. We have observed the following trend from the figures. Overall, the speedup of Inc-Only and Batch-Inc over the baseline decreases as the update batch size increases, since more edge insertion means more vertex activation during initialization stage and, in addition, potentially more unstable

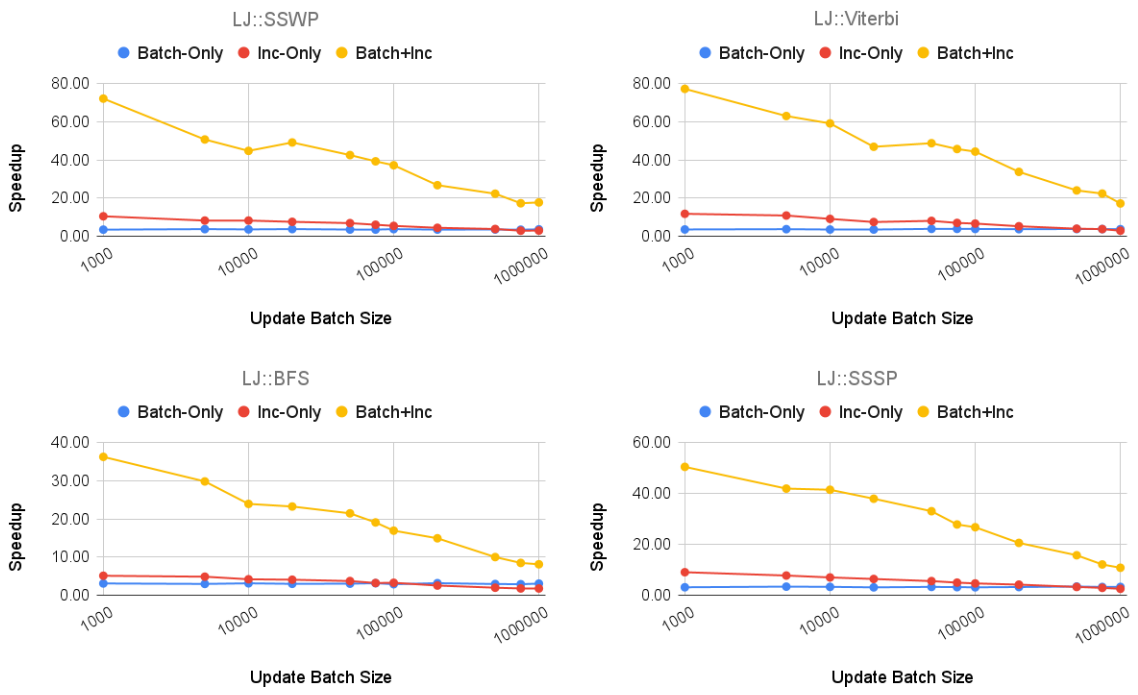


Figure 3.3: Speedup of Batch-Only, Inc-Only, Batch+Inc over Baseline that Runs Shared Query One By One w/o Incremental Computation .

property values to converge during incremental computation due to greater change to the graph structure. The speedup of Batch-Only is not sensitive to the update batch size because batching is a system level optimization for resource sharing and cache efficiency which is independent from incremental computation. Once update batch size reaches 100K and larger, the performance of Inc-Only drops to the same level or even slightly less performant compared with Batch-Only. However, Batch+Inc constantly significantly outperforms both Batch-Only and Inc-Only over the wide range of update batch size from 1K to 1M, although the difference between the three becomes relatively smaller as update batch size increases.

3.3 Summary

In this chapter, we propose generalized incremental computation to efficiently evaluate a batch of arbitrary vertex graph queries by extending the sharing technique and batching principle in SimGQ to the streaming graph scenarios. The results of a small set of fixed shared queries are incrementally maintained while graph is changing. A batch of arbitrary user queries can be efficient handled by sharing the results of the selected shared queries across all the user queries in the batch. Our experiments show that generalized incremental computation delivers averaged speedup from 1.14x to 38.06x over the non-incremental batched evaluation baseline with trivial cost on maintaining shared query results. So far we have discussed the batched evaluation of point-to-all queries on static and streaming graphs. In the following chapters, we will present our work on processing point-to-point graph queries.

Chapter 4

PnP – Evaluating a Point-to-Point Query

Parallel iterative frameworks are used to compute important properties for large real-world graphs. Even though iterative graph analytics algorithms are highly parallel, for large graphs they are expensive due to their *exhaustive* nature (e.g., shortest path algorithm starts from a single source and computes shortest paths to <u>all</u> destination vertices).

Recently Yan et al. [58] observed that many applications on large graphs simply require computing <u>point-to-point</u> variants of heavyweight computations. As an example, when analyzing a graph that represents online shopping history of shoppers, a business may be interested in point-to-point queries over pairs of certain important shoppers. Thus, given a pair of distinct vertices (s, d) in a graph, we are interested in computing point-to-point versions of standard computations such as, shortest path from s to d, widest path from s to d and number of paths from s to d. Yan et al. developed the Quegel [58] framework to solve point-to-point queries.

Although Quegel presents a solution for evaluating point-to-point queries, it is far from optimized. First, Quegel does a significant level of wasteful work as it does not prune traditional <u>one</u>

source to all destinations computation to achieve point-to-point subcomputation. Second, it does not recognize that evaluation times of point-to-point queries in backward and forward directions can greatly differ. In contrast we present **PnP** that addresses the above drawbacks and delivers significant speedups over Quegel.

Quegel supports Hub^2 [18] precomputation to speedup evaluation of individual queries. However, this approach has multiple drawbacks that limits its utility. The experimental data reported in [58] shows that Hub^2 precomputation is expensive. Moreover, in the common scenario where graph structure mutates, the Hub^2 precomputation must be repeated making Quegel unsuitable for streaming (changing) graphs. While KickStarter's value-dependence based trimming strategies [48] can be used to accelerate Hub^2 computation, the repetitive trimming of Hub^2 information does not justify separating it out as a preprocessing step for relatively-inexpensive queries. Finally the Hub^2 [18] precomputation is specifically designed for accelerating shortest path queries on graphs where all edge weights are the same. This limits its use both in terms of types of queries and graphs.

In this chapter, we present **PnP** framework that avoids all of the above limitations of Quegel and efficiently computes lighter weight point-to-point versions of wide range of queries on weighted and unweighted graphs. **PnP** does not require any precomputation thus allowing graph changes in between queries. To quickly respond to queries **PnP** uses dynamic techniques for optimizing query evaluation. In particular, it uses two *general* dynamic techniques: **online Pruning** of graph exploration that eliminates propagation from vertices determined to not contribute to a query's final solution; and **dynamic direction Prediction** method for choosing between solving the query in forward (from source) or backward (from destination) direction as their costs can differ significantly based on the graph structure and computation behavior.

We carry out an experimental study (§4.1) that shows how query characteristics and the direction of evaluation impact runtime. Guided by the observations from the study, we propose **PnP**'s two-phase algorithm (§4.2) that delivers fast evaluation times across queries with differing characteristics. Phase 1 briefly traverses the graph in both forward and backward directions originating from source and destination vertices. By monitoring progress in both directions during this phase we are able to predict the faster direction highly accurately and compute information that enables pruning. Phase 2 completes the point-to-point computation by running the algorithm, with pruning enabled, in the chosen direction to convergence leading to highly efficient query evaluation.

The remainder of the chapter is organized as follows. In Section 4.1 we present a unidirectional framework with pruning and study its performance for 10,000 queries each, with differing characteristics, for multiple graphs and graph algorithms. In section 4.2 we use the observations of the study to guide the development of our two-phase algorithm. In section 4.3 we evaluate the two-phase algorithm. We summarize this chapter in section 4.4.

4.1 Study of Point-to-Point Query Characteristics

In this section we present an algorithm for computing point-to-point queries with simple pruning (sPr) and then analyze the runtime characteristics of the algorithm on 10,000 queries each for four input graphs and multiple analytics problems. This large scale study allows us to uncover runtime characteristics that enable us to develop a new two-phase algorithm that dynamically predicts and adapts execution to deliver highly optimized performance across all types of queries. Note that prior work is limited in its scope – Quegel [58] uses 1000 shortest path queries [58]; thus, the observations made and exploited in this work eluded prior work on Quegel.

Each point-to-point query is of the form $Q(s \rightsquigarrow d, G)$ where G is a directed graph, s is the chosen source vertex, and d is the chosen destination vertex. Thus, we compute the desired property Q with respect to $s \rightsquigarrow d$ (e.g., Shortest Path from s to d, Widest Path from s to d etc.). To avoid negative-weight cycles, edge weights are assumed to be positive. In comparison to standard iterative algorithms, the iterative algorithm for point-to-point query has two distinct features: it employs pruning and it provides direction choice.

- The online **pruning** of graph exploration is enabled by the observation that point-to-point evaluation algorithm only needs to achieve convergence for $s \rightsquigarrow d$ as opposed to all possible (destination) vertices. Pruning dynamically eliminates wasteful computation and propagation that is determined not to contribute to the final solution for the query. Pruning leads to early termination relative to the standard iterative algorithm. The pruning strategy is easily identifiable for monotonic problems, i.e. the solution for the property value being computed monotonically increases or decreases over the iterations of the algorithm before stabilizing.

- In evaluating the query we have **direction choice**. That is, we can either compute $Q(s \rightsquigarrow d, G)$ in forward direction (i.e., starting from s and propagating forward along the directed edges in G), or alternatively, we can compute the query in backward direction as $Q(d \rightsquigarrow s, \widehat{G})$ where we start at d and propagate forward in $\widehat{G}$ which is edge reversed graph corresponding to G (i.e., $\widehat{G}$ is obtained by reversing the direction of all edges in G). We show that the choice of direction impacts execution time.

It is crucial to note that point-to-point queries can also be formulated on undirected graphs. While the techniques presented in this paper work for undirected graphs as well, we present them using directed graphs for simpler exposition. In particular, our direction monitoring and selection techniques are primarily based on the direction of value propagation from/to source/destination vertices, and hence, remains oblivious to the underlying graph being directed or undirected.

In Algorithm 9 EVALUATE carries out *push-style* evaluation of a query for vertex pair $(s \rightsquigarrow d)$ starting at the source vertex s by iteratively processing active vertices by calling PROCESS till the set of active vertices becomes empty and propagation ceases. However, in contrast to standard algorithm, it constructs a *pruned active set*. Pruning is achieved by comparing the newly computed value of each vertex v with that of destination vertex d (line 16). If it is determined that propagating v's current value through the graph cannot cause a change in d's value, then propagation of v's value is pruned. Consider the evaluation of shortest path from s to d. At any execution point, d's current value represents the length of the shortest path from s to d that has been found so far. If v's value, that represents the length of the shortest path from s to v, is greater than or equal to d's value then it need not be propagated as it can only discover longer paths to d.

The above framework relies upon the user to provide two essential functions for each algorithm: **EDGEFUNCTION** is the main computation function that updates the property value of a destination vertex and returns whether the update succeeded or not (CASMIN$(a,\ b)$ sets $a = b$ if $b < a$ atomically using compare-and-swap); and **DONOTPRUNE** which determines whether the propagation can be pruned. For illustration, the two functions for the shortest path point-to-point query SP$(s \rightsquigarrow d, G)$ are given below.

EDGEFUNCTION (e)**:** CASMIN$(e.dest.value,\ e.source.value + e.weight)$

DONOTPRUNE $(v,\ d)$**:** $v.value < d.value$

To solve the query in backward direction we can instead compute SP$(d \rightsquigarrow s, \widehat{G})$.

Algorithm 9 Point-to-Point with Simple Pruning (**sPr**).

```
function EVALUATE ( Q ( $s \rightsquigarrow d$, $Graph$ ) )
    ▷ Initialize active set of vertices
    ACTIVE ← INITIALIZE ( Q ( $s \rightsquigarrow d$ ) )
    ▷ Iterate
    while ACTIVE $\neq \phi$ do
        ACTIVE ← PROCESS ( ACTIVE, $d$, $Graph$ )
    end while
end function

function PROCESS ( ACTIVE , DEST, $Graph$ )
    NEWACTIVE ← $\phi$
    ▷ Compute new property values
    for all $v \in$ ACTIVE do
        for all $e \in Graph.outEdges(v)$ do
            $changed$ ← EDGEFUNCTION ($e$)
            if $changed$ and DONOTPRUNE ($e.dest$, DEST) then
                NEWACTIVE ← NEWACTIVE ∪ $\{e.dest\}$
            end if
        end for all
    end for all
    return NEWACTIVE
end function
```

Table 4.1: Shortest Path Query: Forward sPr.

ACTIVE	A	B	C	D	E	F	G	H
–	∞	∞	∞	∞	∞	∞	∞	∞
S (= 0)	2	1	1	∞	∞	∞	∞	∞
A,B,C	2	1	1	**3**	∞	3	3	3
	Early Termination							
D,F,G,H	2	1	1	**3**	5	3	3	3
E	2	1	1	**3**	5	3	3	3
ϕ	*Normal Termination*							

Table 4.2: Shortest Path Query: Backward sPr.

ACTIVE	S	A	B	C	E	F	G	H
–	∞	∞	∞	∞	∞	∞	∞	∞
D (=0)	∞	∞	2	∞	2	1	2	∞
B,E,F,G	**3**	2	2	3	2	1	2	4
A	**3**	2	2	3	2	1	2	4
ϕ	*Early Termination*							

The example in Figure 4.1 illustrates the above algorithm and the early termination it achieves via pruning. Table 4.1 shows the progress of the shortest path computation from S to D, iteration by iteration. In each row the set of active vertices that are processed is presented along with the updated values following their processing. The values marked in green are those that have changed requiring further propagation while at the same time are not pruned; thus they are used to

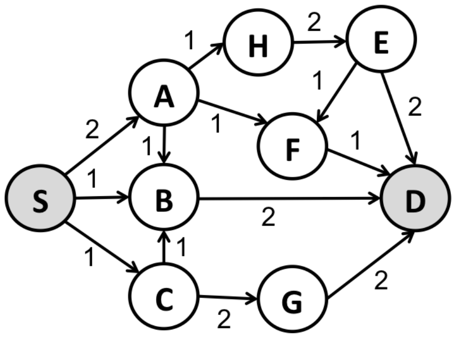


Figure 4.1: Shortest path evaluation $SP(S \rightsquigarrow D)$.

compute the ACTIVE set for the following iteration. The values marked in red are those that have changed but pruned because they are greater than or equal to the value of the D, the destination vertex. Therefore pruning of vertices F, G, H in row three leads to early termination. If pruning is not performed the algorithm takes two additional iterations to terminate. Note that during these iterations the value for vertex D does not change further confirming that the processing of vertices that were pruned does not contribute to the query solution. Table 4.2 illustrates backward evaluation of the shortest path from S to D . When we compare the results of Table 4.2 with that of Table 4.1 we observe that cost of the two algorithms vary. In this case we find that the forward algorithm processes fewer active vertices (and edges) and takes fewer iterations.

Next we present results of our study that analyzes the behavior of above algorithm for different types of queries. We first describe the experimental setup below.

Experimental setup. For this study we implemented our framework using Ligra [41] which uses the Bulk Synchronous Model [45] and provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. The study is based upon four algorithms – Shortest Path (SP), Widest Path (WP), Number of Paths (NP), and Breadth First Search (BFS). We use four input graphs listed in Table 4.3 – two are billion edge graphs (TTW, TT), the other two have tens of millions of edges (LJ, PK). For each input graph, we generated 10,000 queries and used them to evaluate all algorithms. No vertex appears more than once, either as a source or destination, in

Table 4.3: Input graphs.

Graphs	**#Edges**	**#Vertices**	**#Queries**
Twitter (TTW) [20]	1.5B	41.7M	10K
LiveJournal (LJ) [3]	69M	4.8M	10K
Twitter (TT) [5]	2.0B	52.6M	10K
PokeC (PK) [42]	31M	1.6M	10K

these queries. Moreover, the vertices chosen as sources and destinations are selected by sampling all the vertices ordered according to their degrees. All experiments were performed on a 64 core (8 sockets, each with 8 cores) machine with AMD Opteron 2.3 GHz processor 6376, 512 GB memory, and running CentOS Linux release 7.4.1708.

For purpose of analysis, the 10,000 queries used can be classified into four distinct categories based upon combination of two properties: (Fwd vs. Bwd) queries for which forward evaluation is faster belong to Fwd and those for which backward evaluation is faster belong to Bwd; (NR vs. R) queries that reveal that destination is non-reachable from the source belong to NR and queries where destination is reachable from source belong to R. Therefore, the queries on a given workload can be divided into four categories: FwdNR, BwdNR, FwdR, and BwdR. The distribution of the 10,000 queries based upon faster/slower direction and reachability/non-reachability is shown in Table 4.4. We observe there are a good number of queries of all four types. Note that numbers for NR queries are same for different benchmarks as they are mainly determined by graph structure.

Analysis of execution times – We ran all 10,000 queries for each input on sPr versions of all four graph algorithms and collected their forward and backward evaluation times. For reachable queries sPr carries out pruning once it finds the first approximation of query solution while for non-reachable queries pruning never takes place as query has no result. Average execution times of all queries by category are given in Tables 4.5 (Non-Reachable) and 4.6 (Reachable). Figure 4.2 shows a representative scatter plot of the execution times (all plots are shown in later section) – the times of queries in order of FwdNR FwdR, BwdR, and BwdNR from left to right are plotted. Based upon the data we make two key observations.

Table 4.4: Characteristics of 10,000 queries used in experiments: Fwd – Forward faster; Bwd – Backward faster; R – Reachable; and NR – Non-reachable.

G	Queries	WP	SP	NP	BFS
TTW	FwdR	49.26%	44.55%	15.24%	31.07%
	BwdR	13.30%	18.01%	47.32%	31.49%
	FwdNR	20.41%	20.41%	20.41%	20.41%
	BwdNR	17.03%	17.03%	17.03%	17.03%
LJ	FwdR	10.81%	13.23%	12.84%	7.89%
	BwdR	37.41%	34.99%	35.38%	40.33%
	FwdNR	24.75%	24.75%	24.75%	24.75%
	BwdNR	27.03%	27.03%	27.03%	27.03%
TT	FwdR	41.86%	10.61%	28.86%	29.23%
	BwdR	12.40%	43.65%	25.40%	25.03%
	FwdNR	35.02%	35.02%	35.02%	35.02%
	BwdNR	10.72%	10.72%	10.72%	10.72%
PK	FwdR	3.60%	5.27%	1.65%	6.69%
	BwdR	17.30%	15.63%	19.25%	14.21%
	FwdNR	38.55%	38.55%	38.55%	38.55%
	BwdNR	40.55%	40.55%	40.55%	40.55%

Table 4.5: (sPr on NR queries) Avg. Execution Times in Faster Direction (seconds); and Avg. Slowdown Factor in Slower Direction.

G	Q	WP		SP		NP		BFS	
TTW	FwdNR	.0130s	1096.5×	.0129s	1137.6×	.0268s	128.2×	.0058s	3.42×
	BwdNR	.0365s	318.3×	.0258s	562.8×	.0303s	145.1×	.0089s	105.2×
LJ	FwdNR	.0009s	484.4×	.0010s	875.9×	.0055s	40.9×	.0010s	79.9×
	BwdNR	.0009s	666.9×	.0013s	620.4×	.0071s	34.6×	.0012s	67.1×
TT	FwdNR	.0191s	772.9×	.1771s	88.7×	.0551s	124.3×	.0170s	65.7×
	BwdNR	.0282s	620.5×	.0154s	1560.2×	.0595s	117.3×	.0299s	44.7×
PK	FwdNR	.0005s	250.3×	.0004s	458.7×	.0024s	72.98×	.0004s	76.02×
	BwdNR	.0004s	478.7×	.0006s	263.0×	.0023s	73.2×	.0004s	75.7×

Observation 1 – Fwd vs. Bwd: *direction is important.* Picking the right direction for solving a query is important. From Figure 4.2 we can easily see that for non-reachable queries the difference in forward and backward execution times is consistently high and the time in the faster direction

Table 4.6: (sPr on R queries) Avg. Execution Times in Faster Direction (seconds); and Avg. Slowdown Factor in Slower Direction.

G	Q	WP		SP		NP		BFS	
TTW	FwdR	5.5598s	2.21 ×	9.5778s	1.33 ×	2.2778s	1.21 ×	0.2546s	2.12 ×
	BwdR	7.5349s	1.51 ×	11.177s	1.16 ×	2.4258s	1.37 ×	0.4611s	1.43 ×
LJ	FwdR	0.2480s	2.36 ×	0.7036s	1.18 ×	0.1316s	1.16 ×	0.0437s	1.20 ×
	BwdR	0.1645s	4.90 ×	0.5869s	1.38 ×	0.1205s	1.32 ×	0.0355s	1.60 ×
TT	FwdR	7.2006s	1.94 ×	11.8350s	1.27 ×	3.8697s	1.30 ×	0.4501s	1.46 ×
	BwdR	9.5975s	1.74 ×	14.7070s	1.36 ×	4.2492s	1.27 ×	0.6047s	1.29 ×
PK	FwdR	0.0742s	1.87 ×	0.1319s	1.16 ×	0.0683s	1.10 ×	0.0175s	1.26 ×
	BwdR	0.0481s	3.68 ×	0.1125s	1.31 ×	0.0607s	1.31 ×	0.0125s	1.47 ×

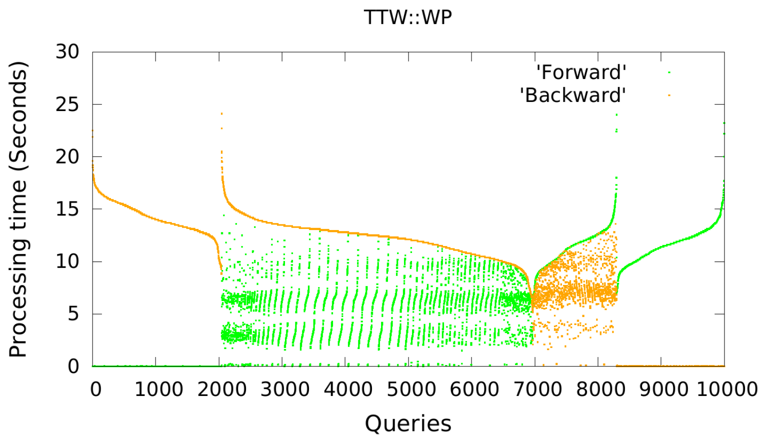


Figure 4.2: Forward and Backward Evaluation Times.

is very small; and for reachable queries the difference between forward and backward evaluation times varies from very large to very small. This observation holds across all algorithms and all input graphs as shown by the average times in the faster direction in Tables 4.5 and 4.6. Each table also presents the factor by which the average execution time increases if a query is solved in the slower direction as opposed to faster direction. From Table 4.6 for NR queries not only is the execution in faster direction very small (tens of milliseconds), in the slower direction it is orders of magnitude slower (around a second). From Table 4.6 for reachable queries the average execution time in faster direction is higher (several seconds for large graphs) and the slowdown factor is lower.

Observation 2 – NR vs. R: *reachability is important.* Picking the right direction alone is not enough to achieve the best performance. We need a strategy for handling both non-reachable and reachable queries effectively. In particular, we note that FwdNR/BwdNR queries can be evaluated significantly faster than FwdR/BwdR queries – well over two and often over three orders of magnitude faster. For example, for SP on TTW, average times for FwdNR/BwdNR are 0.0129s/0.0258s while for FwdR/BwdR they are 9.5778s/11.177s. In other words, since at the start of a query evaluation we do not know whether it is NR or R, we need to design a strategy that quickly classifies it as NR or R and then appropriately handles them to get the fast execution times.

Next we develop a two-phase algorithm that exploits the above observations in delivering fast evaluation of all four types of queries.

4.2 PnP Two-Phase Framework

The goal of this section is to develop a general algorithm that delivers execution times that are close to the execution times in the faster direction for all types of queries. Based upon the observations in the preceding section, we can set the requirements that must be met by the point-to-point query evaluation algorithm as follows:

- *RQ1: effectively handle non-reachable and reachable queries (first observation);*
- *RQ2: identify the faster direction and use it for query evaluation (second observation); and*
- *RQ3: maximize the use of pruning for reachable queries for quickly responding to each query.*

In this work we develop an algorithm that by design meets RQ1, predicts direction to meet RQ2, and embodies a significantly enhanced pruning strategy to meet RQ3.

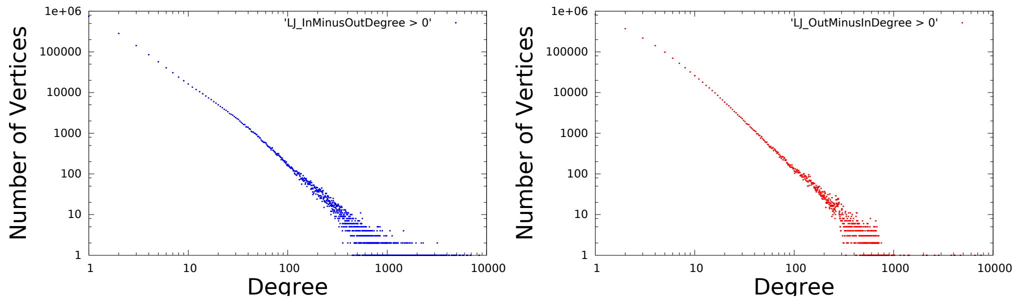


Figure 4.3: In- and Out-Degree Distribution of LiveJournal.

In general, both reachability (RQ1) and direction (RQ2) requirements must be handled dynamically as queries constructed from sampling of vertices were found to fall in all four categories (see Table 4.4). Clearly reachability is function of the graph structure and thus without exploring the graph at runtime we cannot determine whether a query is a NR query or R query. The choice of direction matters because the cost of forward evaluation is high if forward propagation encounters many high outdegree vertices while backward evaluation is high if backward propagation encounters many high indegree vertices. We cannot simply statically look at the graph and select the same preferred direction for all queries as the overall characteristics of G and $\widehat{G}$ are similar. In Figure 4.3 we plot the in-degree and out-degree distributions for the LiveJournal graph. As we can see, both in-degrees and out-degrees have similar power-law distributions. Thus, for a given query, without dynamically exploring the graph in both directions we cannot establish a basis for selecting the preferred direction. Finally for meeting requirement RQ3 we need to quickly find the first approximation of the query result as soon as possible so that pruning is enabled early and greater fraction of execution is optimized via pruning.

Therefore, to simultaneously meet all three requirements, we propose a two-phase algorithm such that Phase 1 dynamically and very quickly finds a suitable configuration (for NR vs. R;

choose Fwd vs. Bwd) for evaluating a given query and then execution transitions to Phase 2 that evaluates the query under the selected configuration. More specifically, Phase 1 classifies the query as non-reachable or reachable (RQ1), selects preferable direction for query evaluation as forward or backward (RQ2), and enables pruning by finding a safe estimate of query's result value (RQ3). That is, Phase 1 sets the stage for requirements stated above to be met. Upon completion of Phase 1 execution moves to Phase 2 that solves the query in the preferable direction using the safe estimate of query's result to prune graph exploration. Next we discuss the design of phases in greater detail.

Phase 1: Bidirectional Exploration for Identifying Configuration. At the start we are faced with two questions: NR vs. R? and Fwd vs Bwd?. Thus, Phase 1 must decide which one to target first. Recall that in the previous section we observed that for NR queries typically one direction solves the query very quickly than any other case, i.e. for NR query in opposite direction or R query in any direction. The reason for this behavior is that typically, NR query evaluation in faster direction examines only a small fraction of the graph that is examined for its evaluation in slower direction, or evaluation of a R query examines a very large fraction of the graph. For example, for a sample of 10 NR and 10 R queries for WP on LiveJournal, the average percentage of vertices visited in Fwd (Bwd) direction was $< 1\%$ (87%) for NR queries and 84% (84.4%) R queries.

Given the above observation, to answer NR queries quickly we design Phase 1 to first distinguish between NR and R queries by attempting to identify a directed path from the source vertex and the destination vertex. Since we do not know which direction, forward or backward, is preferable, Phase 1 uses bidirectional exploration in both directions: forward from the source vertex; and backward from the destination vertex. During bidirectional exploration, the exploration in the fast direction quickly establishes that the query is of NR kind while relatively little time is expended in exploring the graph in the slower direction. In other words, NR queries will be identified and answered very quickly without even having to predict the preferable direction.

On the other hand, if the query is a R query, the bidirectional exploration runs a bit longer till Phase 1 determines the existence of a path from source to destination. This happens when bidirectional exploration causes some vertex to be visited from both forward and backward directions. As soon as this occurs, we know that we have a R query. The extra time spent executing allows us to observe the progress in both directions and make a choice of direction. Moreover, since a path has been fully traversed we can generate our first estimate of query's result that can be used for pruning. Now the execution transitions to Phase 2 by continuing propagation in the chosen direction while terminating propagation in the other direction, with pruning turned on right from start of Phase 2. Since Phase 1 is fast, pruning use is maximized.

The above approach meets all the requirements as follows: (RQ1) it optimizes evaluation of both NR and R queries; (RQ2) it addresses direction problem by avoiding it for NR queries that can be quickly solved using bidirectional exploration and by predicting the preferable direction for R queries; and (RQ3) it guarantees that pruning is turned on for entire Phase 2.

Next we explain the details of how the desirable direction is predicted and safe approximation of query solution is computed to enable pruning at the start of Phase 2.

Direction prediction – To predict the faster direction we considered a number of measures: (Work remaining) as estimated by number of active vertices in each direction; (Work performed) as estimated by tracking the number of vertices processed in each direction; and (Hybrid) method that uses a combination of preceding two measures giving more importance to the first measure. Our experience showed that the first measure provides the highest prediction rate and thus, the direction for which there are fewer active vertices is predicted as the faster direction and used in Phase 2. An advantage of this measure is that it does not incur extra tracking overhead involved in the other measures.

Safe approximation of query solution – Since for some vertex v we have at least found a path from source to v in the forward direction and a path from v to destination in the backward direction, we can compute an estimate for query's result. When multiple vertices are visited from both directions we select the best approximation provided across all these vertices. To make use of the two-phase algorithm of our **PnP** framework, the user must provide two additional functions: one for the estimation of query result, **ESTIMATEAPPROX**, from a single vertex; and another for safe approximation for a query, **SAFEAPPROX**, from **ESTIMATEAPPROX** values of all vertices that are visited in both directions. We illustrate these by providing the functions for the shortest path query SSSP($s \rightsquigarrow d$).

ESTIMATEAPPROX(v)	$v.\overrightarrow{value}(s) + v.\overleftarrow{value}(d)$
SAFEAPPROX	$\forall\ v\ \min($ ESTIMATEAPPROX(v) $)$

Phase 2: Query Evaluation. Upon termination of Phase 1, the execution transitions into Phase 2 where the propagation in the predicted direction is run to completion while the execution in the non-predicted direction is discontinued. Note that all the processing performed in Phase 1 for the predicted direction is not wasted as computation continues from where it was for the predicted direction. At the start of Phase 2, if the predicted direction is forward the initial value for destination vertex d is set to **SAFEAPPROX** produced by Phase 1 and if the predicted direction is backward the initial value for the source vertex s is set to **SAFEAPPROX**.

Algorithm 10 Two-Phase PnP Evaluation (**2Phase**) .

```
1:  function 2PHASE( Query ( $s \rightsquigarrow d, G$ ) )
2:      ▷ Initialization
3:      VISITF (*) ← VISITB (*) ← False
4:      FACTIVE ← INITIALIZE ( Query ( $s \rightsquigarrow d, G$ ) )
5:      BACTIVE ← INITIALIZE ( Query ( $d \rightsquigarrow s, \widehat{G}$ ) )
6:      SAFEAPPROX ← Query.Initialize
7:      ▷ Phase 1
8:      while TRUE do
9:          ▷ Process active vertices
10:         PROCESSED ← FACTIVE ∪ BACTIVE
11:         FACTIVE ← PROCESS ( FACTIVE, $d$, $G$ ); BACTIVE ← PROCESS ( BACTIVE, $s$, $\widehat{G}$ )
12:         ▷ Update Visit Flags of processed vertices
13:         VISITF ($v$) ← True, $\forall\ v \in$ FACTIVE; VISITB ($v$) ← True, $\forall\ v \in$ BACTIVE
14:         ▷ Case I: Non-Reachable Query
15:         if FACTIVE $= \phi \vee$ BACTIVE $= \phi$ then
16:             return ( Not-Reachable )
17:         end if
18:         ▷ Case II: Reachable Query
19:         for all $v \in$ PROCESSED do
20:             if VISITF ($v$) $\wedge$ VISITB ($v$) then
21:                 REACHABLE ← TRUE; NEWVALUE ← ESTIMATEAPPROX($v$)
22:                 SAFEAPPROX ← $f_{approx}$ ( NEWVALUE, SAFEAPPROX )
23:             end if
24:         end for all
```

Algorithm 11 Two-Phase PnP Evaluation (**2Phase**) (Cont.).

```
25:         if REACHABLE then
26:             PREDICTION ← |FACTIVE| > |BACTIVE| ? BACKWARD : FORWARD
27:             break
28:         end if
29:     end while
30:     ▷ Phase 2
31:     if PREDICTION = FORWARD then
32:         ▷ Initialize destination $d$ vertex value
33:         $d.value$ = SAFEAPPROX
34:         ▷ Continue iterating: forward direction only
35:         while FACTIVE $\neq \phi$ do
36:             FACTIVE ← PROCESS ( FACTIVE, $d$, $G$ )
37:         end while
38:         return ( Reachable, d.value )
39:     else  ▷ Prediction is Backward
40:         ▷ Initialize source $s$ vertex value
41:         $s.value$ = SAFEAPPROX
42:         ▷ Continue iterating: backward direction only
43:         while BACTIVE $\neq \phi$ do
44:             BACTIVE ← PROCESS ( BACTIVE, $s$, $\widehat{G}$ )
45:         end while
46:         return ( Reachable, s.value )
47:     end if
48: end function
```

Algorithm 10 summarizes the two-phase algorithm. The iterative loop (lines 7–29) representing Phase 1 processes active vertices and identifies active vertices for the next iteration. Phase 1 terminates under two conditions. First is when the query is found to be non-reachable because the active set in one of the directions becomes empty and thus the algorithm terminates (see lines 14–17). Second is when the query is found to be reachable in which case safe approximation is computed and direction for Phase 2 is predicted (see lines 18–28). The Phase 2 (lines 30–47) simply continues processing in the predicted direction, using the safe approximation, and terminates when the algorithm converges. During processing of active vertices in Phase 1 pruning is always off while in Phase 2 pruning is always on.

Note that the proposed algorithm satisfied all three requirements. Our approach handles both non-reachable and reachable queries (RQ1). For non-reachable queries our execution time is expected to be close to the faster direction time which is much smaller than the slower direction time. For reachable queries since Phase 1 is fast, Phase 2 is highly optimized as our algorithm accurately predicts the faster direction (RQ2) and maximizes the use of pruning by ensuring that it is enabled right from the start of Phase 2 (RQ3).

Applicability of PnP. The **PnP** two phase algorithm minimizes computations by limiting propagation of values via direction selection and safe pruning. We further understand how direction selection and pruning can be applied to a wide variety of graph algorithms. Graph algorithms are typically convergence based iterative algorithms wherein vertex values propagate as they change across iterations. These propagations happen across the structure of the input graph, and hence, they can be viewed as occurring in certain pattern or direction. At an elementary level, propagation of a vertex value occurs in the "outward" direction through out-neighbors of the vertex; for example, in Algorithm 9, the out-neighbors of vertices get processed (line 14) as values propagate across the graph. However, an important characteristic of point-to-point queries is the two special vertices (a source and a destination) that concretely define an expected direction for propagation: forward direction from source to destination. **PnP** further extracts the hidden reverse direction to leverage the disparity in propagation and limits overall computations via pruning. Path based algorithms naturally fit this class of point queries where values are expected to be propagated from source to destination. For general algorithms like PageRank, every vertex acts like a source; thus, it is difficult to deduce a single direction of flow of values that can be leveraged by **PnP**. On the other hand, pruning of value propagation occurs when we know (a) what to prune; and, (b) how to prune it.

— What to prune? While **PnP** prunes value propagations (or edge computations) in a broader sense, the semantics of each graph algorithm needs to be carefully analyzed to identify the exact propagation paths across which values will never be transferred. These semantics can be captured by characterizing the aggregation function used to compute vertex values. The most common aggregation functions used across graph algorithms are shown in Table 4.7. Since selection based aggregation functions (min, max, or) effectively select values coming from a single incoming path to a given vertex, **PnP** can safely prune values coming from other incoming paths to a vertex, hence supporting several graph algorithms, some of which are listed in Table 4.7. Complete aggregations (sum, product) on the other hand combine values coming from multiple incoming edges into a single value. This means the value contributions from individual incoming paths cannot be discarded throughout the computation, and hence, **PnP** does not prune value propagations but instead only performs direction selection. In our experiments (§4.3), we use NumPaths as an example for complete aggregation to show that **PnP** is very useful even without availability of pruning.

Table 4.7: Applicability of **PnP**.

Aggregation	**Type**	**Graph Algorithms**
min, max, or	Selection	Shortest Paths, Widest Paths, Connected Components, Reachability Minimum Spanning Tree Betweenness Centrality
sum, product	Complete	NumPaths, PageRank, SpMV, Belief Propagation

— How to prune? Once we have identified the propagation paths to prune, we rely on the algorithmic semantics to perform pruning. Vertex values for path based algorithms that rely on selection functions often progress in a monotonic fashion, i.e., subsequent values of vertices are either non-increasing (e.g., shortest paths) or non-decreasing (e.g., widest paths). **PnP** monitors the destination vertex's values and performs numerical comparison ($\geq$, $\leq$) to safely prune out propagations that cannot contribute to the result. For algorithms beyond monotonic convergence (e.g., PageRank), algorithm-specific pruning conditions can be formulated by the user.

4.3 Evaluation of PnP Two-Phase

We evaluate the two-phase algorithm with four input graphs and four graph analytics benchmarks. We use four input graphs from Table 4.3. Four types of queries are considered – Widest Path (WP), Shortest Path (SP), Number of Paths (NP), and Breadth First Search (BFS). We first evaluate the two-phase algorithm for non-reachable queries and then for reachable queries. The algorithms compared are as follows:

- 2Phase (2Ph) – our two-phase algorithm (from §4.2); and
- sPr – simple Pruning algorithm that can be run in forward or backward direction (from §4.1).

4.3.1 Evaluation for Non-Reachable (NR) Queries

The execution times for 2Phase as well as sPr in forward and backward directions for all non-reachable queries are shown in the scatter plots of Figure 4.4. As we can see, for vast majority of queries the execution time of 2Phase algorithm is very close to the time for the faster direction which is significantly smaller that the time in the slower direction.

The average times across all queries for sPr in the faster direction (sPr:FastNR) and slower direction (sPr:SlowNR) as well as 2Phase algorithm can be found in Table 4.8. The effectiveness of 2Phase algorithm is computed as

$$\frac{sPr : SlowNR - 2Phase}{sPr : SlowNR - sPr : FastNR} \times 100$$

which computes $\underline{\text{actual benefit}}$ of two-phase as a percentage of $\underline{\text{available benefit}}$ – this number is shown in parenthesis in Table 4.8. This number is often over 90%. The last column in the table (**Vertices Visited**) indicates the fraction of vertices in the entire graph that are visited by each algorithm. The numbers for sPr:FastNR and sPr:SlowNR confirm that a tiny fraction of the vertices are visited ($<$ 0.03%) in the fast direction while vast majority of vertices are visited (80-90%) in the slower direction. Finally, two-phase algorithm visits less than 0.5% percent of the vertices explaining its effectiveness for non-reachable queries.

4.3.2 Evaluation for Reachable Queries

Let us analyze the performance of the two-phase algorithm for reachable queries. For this analysis we again compare its performance with that of the limits of performance of the sPr

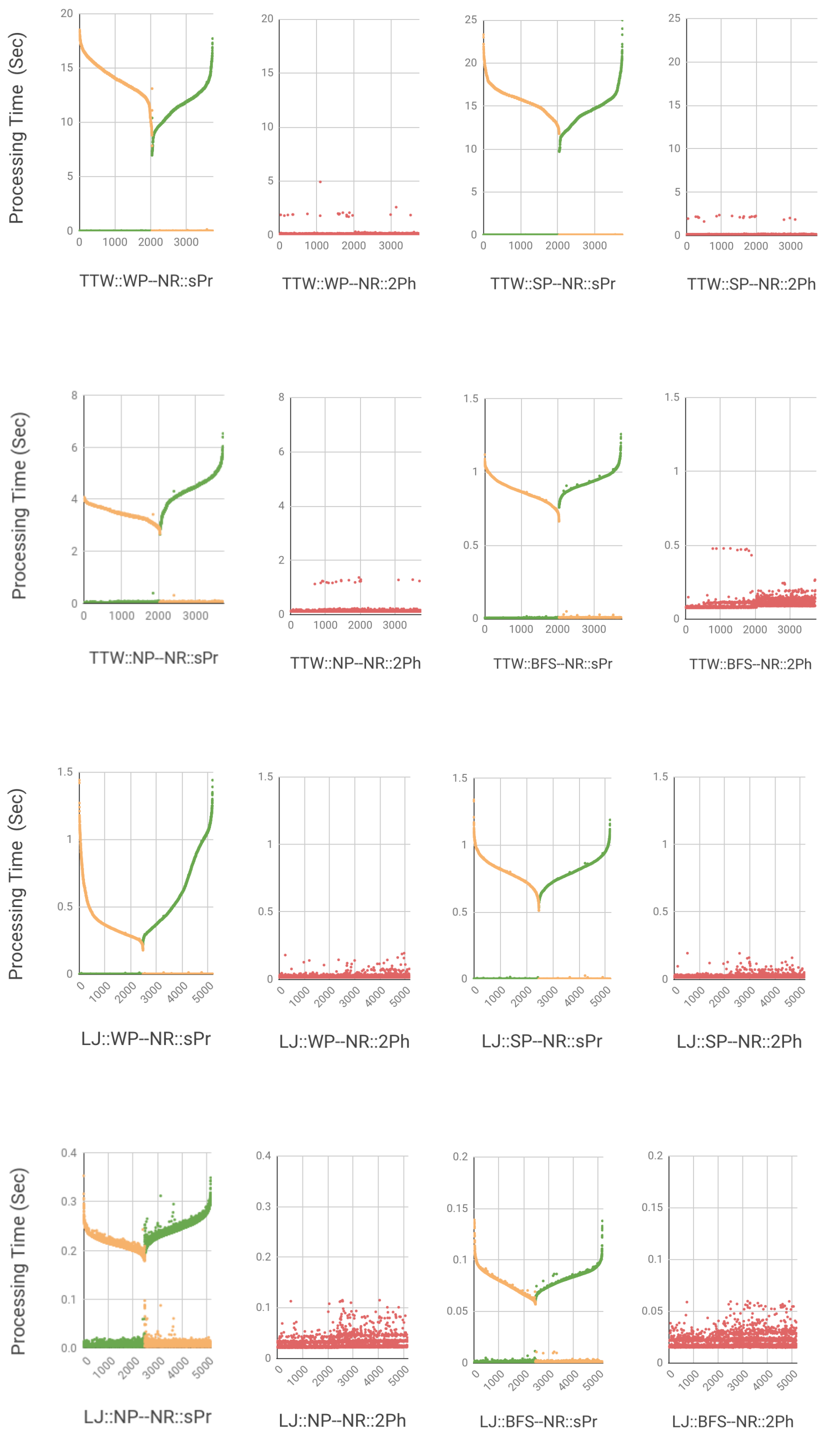


Figure 4.4: **Execution Times of NR Queries**: [Left] sPr Forward (Green) & sPr Backward (Gold); and [Right] 2Ph (Red).

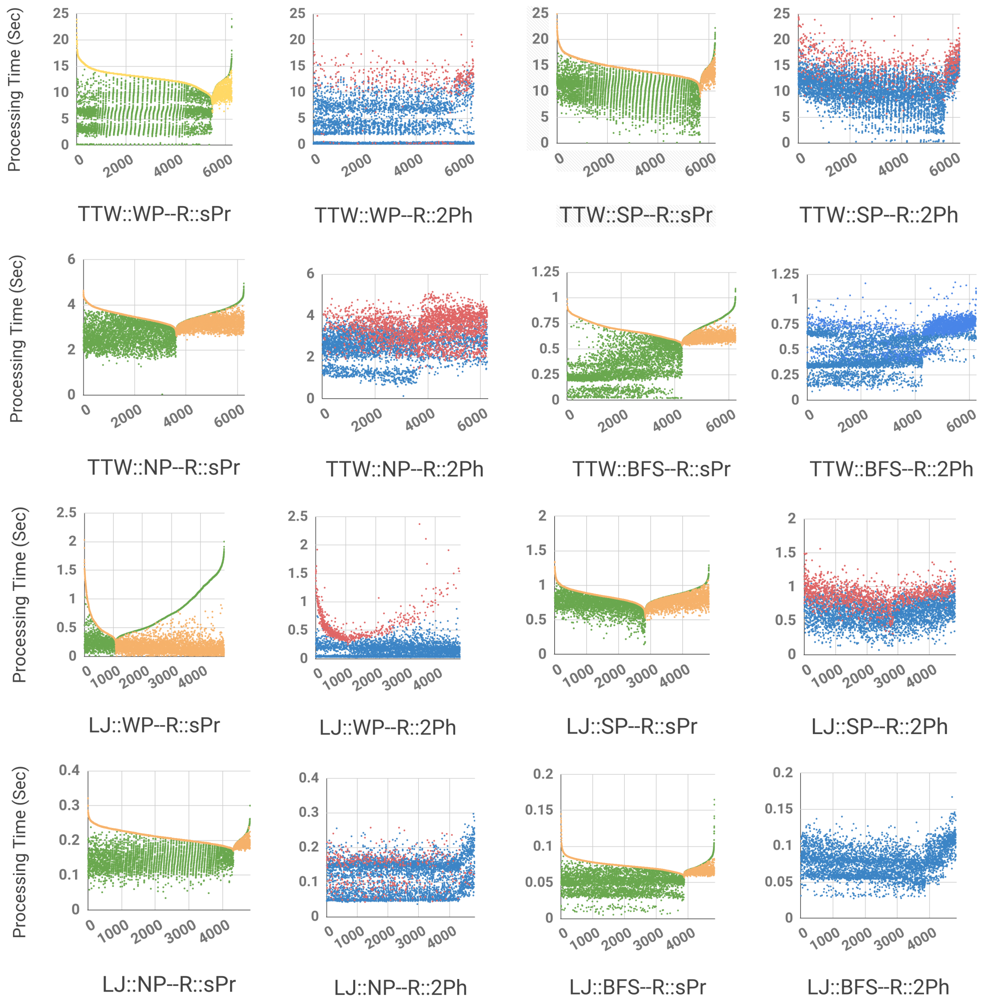


Figure 4.5: **Execution Times of R Queries**: [Left] sPr Forward (Green) & sPr Backward (Gold); and [Right] 2Ph Direction Correctly Predicted (Blue) & 2Ph Direction Mispredicted (Red).

Table 4.8: **NR Queries 2Ph vs. sPr**: Average Execution Times (seconds); and % of Vertices Visited.

G	Algorithm	WP		SP		NP		BFS	
TTW	2Phase	0.144s	(99.1%)	0.145s	(99.1%)	0.140s	(97.1%)	0.101s	(89.2%)
	sPr:FastNR	0.024s		0.019s		0.028s		0.007s	
	sPr:SlowNR	13.03s		14.61s		3.85s		0.87s	
LJ	2Phase	0.019s	(96.6%)	0.019s	(77.8%)	0.027s	(91.0%)	0.019s	(77.5%)
	sPr:FastNR	0.001s		0.001s		0.006s		0.001s	
	sPr:SlowNR	0.534s		0.820s		0.237s		0.080s	
TT	2Phase	0.174s	(99.0%)	0.199s	(99.7%)	0.267s	(96.9%)	0.149s	(88.8%)
	sPr:FastNR	0.021s		0.139s		0.056s		0.020s	
	sPr:SlowNR	15.39s		17.66s		6.88s		1.17s	
PK	2Phase	0.008s	(94.3%)	0.009s	(94.8%)	0.012s	(94.1%)	0.009s	(71.6%)
	sPr:FastNR	.0004s		.0005s		.0023s		.0004s	
	sPr:SlowNR	0.141s		0.155s		0.171s		0.029s	

G	Algorithm	Vertices Visited
TTW	2Phase	0.1767%
	sPr:FastNR	0.0000029%
	sPr:SlowNR	90.62%
LJ	2Phase	0.38%
	sPr:FastNR	0.0299%
	sPr:SlowNR	88.95%
TT	2Phase	0.47%
	sPr:FastNR	0.0000024%
	sPr:SlowNR	84.60%
PK	2Phase	0.0688%
	sPr:FastNR	0.000066%
	sPr:SlowNR	80.77%

algorithm (i.e., in faster and slower directions for all queries). The scatter plots for reachable queries are shown in Figure 4.5.

Average execution times across all reachable queries for algorithms sPr and 2Phase are given in Table 4.9. As we see in most cases execution times of algorithms are related as follows: sPr:FastR $<$ 2Phase $<$ sPr:SlowR. This is to be expected as sPr:FastR is in a sense ideal time

Table 4.9: **R Queries**: Avg. Execution Time Per Query (secs).

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	**3.5116s**	10.827s	2.9134s	0.5396s
	2Ph100%	**3.2826s**	10.466s	2.6813s	–
	sPr:FastR	5.9797s	10.038s	2.3898s	0.3585s
	sPr:SlowR	12.0750s	12.793s	3.1887s	0.6007s
LJ	2Ph	0.1998s	0.6928s	**0.1179s**	0.0781s
	2Ph100%	**0.1572s**	0.6543s	**0.1169s**	–
	sPr:FastR	0.1832s	0.6190s	0.1234s	0.0369s
	sPr:SlowR	0.7569s	0.8168s	0.1575s	0.0560s
TT	2Ph	**4.3782s**	16.727s	4.7800s	0.8338s
	2Ph100%	**3.8370s**	15.049s	4.5825s	–
	sPr:FastR	7.7483s	14.145s	4.0473s	0.5214s
	sPr:SlowR	14.6030s	19.041s	5.2159s	0.7136s
PK	2Ph	0.0705s	0.1392s	0.0980s	0.0342s
	2Ph100%	0.0631s	0.1361s	0.0968s	–
	sPr:FastR	0.0526s	0.1174s	0.0613s	0.0141s
	sPr:SlowR	0.1705s	0.1492s	0.0789s	0.0195s

where overhead of prediction is nil and prediction rate is 100%. In comparison 2Phase algorithm involves overhead of direction prediction and has less than perfect prediction rate. However, as we can see 2Phase is frequently far closer to sPr:FastR than sPr:SlowR. This indicates that 2Phase is highly effective. To further demonstrate its effectiveness, we also present the average execution time 2Phase100% which is computed assuming perfect 100% prediction rate. We can see that 2Phase is only slightly greater than 2Phase100%. Finally, it should be noted that in some cases 2Phase $<$ sPr:FastR (WP on TTW, NP on LJ) or at least 2Phase100% $<$ sPr:FastR (WP on LJ). This is because the 2Phase pruning strategy significantly outperforms the pruning carried out by sPr and thus more than offsets the cost of prediction. Note that for BFS no times for 2Phase100% are provided as BFS terminates at the end of Phase 1. Next we further analyze prediction and pruning in greater detail.

Prediction effectiveness of 2Phase algorithm is analyzed in Table 4.10. The prediction rates (PR) of the 2Phase algorithm are presented – on an average the prediction rates exceed 80%. For the two cases where 2Phase < sPr:FastR we can see that the prediction rates exceed 90% (92.74% for WP on TTW; 90.17% for NP on LJ). Additional data in Table 4.10 shows that for queries where predictions are correct, on average, the difference in execution times in two directions (ΔP) is typically greater than for queries where missprediction occurs (ΔMP). That is, benefit of correct predictions is higher than the loss due to incorrect predictions.

Pruning effectiveness of 2Phase algorithm is analyzed in Table 4.11. We present the percentage of execution time over which pruning is not enabled – lower numbers are better. For 2Phase algorithm this time is the percentage of execution time spent in Phase 1. For sPr algorithm we found this time by noting the point at which the first approximation of query result is generated for use in pruning during remainder of the execution. From the results we can see that for the

Table 4.10: 2Ph Prediction Effectiveness: (PR) Prediction Rate of 2Phase Algorithm; and Difference Between Average Execution Times (seconds) in Faster and Slower Directions for Predicted Queries (ΔP); and Mispredicted Queries (ΔMP). **BFS** omitted as it has no Phase 2.

G	Pred	WP	SP	NP
TTW	PR	92.74%	87.69%	56.51%
	ΔP	11.64s	5.67s	0.90s
	ΔMP	3.16s	2.93s	0.53s
LJ	PR	86.35%	71.67%	90.17%
	ΔP	0.61s	0.22s	0.04s
	ΔMP	0.31s	0.14s	0.01s
TT	PR	88.32%	57.24%	58.39%
	ΔP	12.75s	4.84s	0.45s
	ΔMP	4.63s	3.92s	0.48s
PK	PR	87.99%	87.80%	81.67%
	ΔP	0.13s	0.06s	0.015s
	ΔMP	0.06s	0.026s	0.003s

2Phase algorithm this time is often significantly smaller than for sPr:FastR algorithm. That is, 2Phase pruning is substantially better than simple pruning performed by sPr.

Tables 4.12 and 4.13 present the work performed in terms of total number of active vertices encountered and number of iterations for convergence. As we can see, the number of active vertices is significantly smaller for 2Phase in comparison to sPr:FastR. This reduction is the highest for NP because the SafeApprox is computed by multiplying the NP values in forward and backward direction causing pruning to be highly effective. This is another indicator of the enhanced pruning strategy of 2Phase algorithm being significantly superior than that of sPr. On the other hand, the average number of iterations for 2Phase and sPr:FastR are fairly close. Note that even though the 2Phase algorithm for BFS terminates at end of Phase 1, its vertex computations count

Table 4.11: **R Queries**: % of Execution Time for which Pruning is Inactive. **BFS** is omitted because it does not require Phase 2 as it terminates at the end of Phase 2.

G	Algorithm	WP	SP	NP
TTW	2Ph	28%	1.6%	6.1%
	2Ph100%	30%	1.7%	6.1%
	sPr:FastR	54%	31%	64%
	sPr:SlowR	45%	36%	74%
LJ	2Ph	21%	3.7%	22%
	2Ph100%	22%	3.1%	22%
	sPr:FastR	32%	8.3%	45%
	sPr:SlowR	42%	18%	86%
TT	2Ph	16%	1.2%	3.3%
	2Ph100%	14%	1.0%	2.6%
	sPr:FastR	29%	17%	28%
	sPr:SlowR	41%	27%	49%
PK	2Ph	2.6%	1.1%	2.9%
	2Ph100%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
	sPr:FastR	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
	sPr:SlowR	41%	25%	49%

Table 4.12: **R Queries**: Average Active Vertex Count Per Query (in millions).

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	5.82m	22.17m	27.86m	6.93m
	2Ph100%	5.17m	21.80m	15.27m	–
	sPr:FastR	14.65m	32.46m	54.13m	11.54m
	sPr:SlowR	51.16m	55.34m	48.06m	13.46m
LJ	2Ph	1.58m	6.12m	0.32m	0.80m
	2Ph100%	1.24m	6.31m	0.29m	–
	sPr:FastR	1.98m	6.85m	3.50m	0.85m
	sPr:SlowR	8.47m	12.15m	5.72m	2.13m
TT	2Ph	7.56m	30.56m	42.13m	7.51m
	2Ph100%	6.27m	39.07m	42.13m	–
	sPr:FastR	16.74m	59.87m	54.10m	12.04m
	sPr:SlowR	60.03m	50.34m	63.87m	17.77m
PK	2Ph	0.61m	1.38m	0.39m	0.28m
	2Ph100%	0.56m	1.37m	0.77m	–
	sPr:FastR	0.81m	1.70m	2.57m	0.37m
	sPr:SlowR	2.70m	2.99m	3.46m	0.67m

is consistently lower than that for sPr:FastR indicating that the bidirectional traversal is more effective that unidirectional traversal. Finally, occasionally 2Phase performs fewer active vertices than 2Phase100% (e.g., SP on LJ). This is because the runtime cost depends upon additional factors (e.g., number of edges associated with active vertices, cache misses etc.), i.e. the direction in which fewer active vertices are processed can have higher execution time.

Beamer's Bidirectional BFS [4] vs. *Two-Phase PnP.* Recently bidirectional BFS was proposed by Beamer that switches directions at iteration boundaries to minimize the work performed – the frontier sizes in two directions are compared to select the cheaper direction for the next iteration. Although this is an effective algorithm, **PnP** relies upon direction selection over bidirectional search. First, **PnP** is general which solves problems besides BFS while Beamer's algorithm only

Table 4.13: **R Queries**: Average Number of Iterations Per Query (rounded).

G	Algorithm	WP	SP	NP	BFS
TTW	2Ph	5	9	5	4
	2Ph100%	4	9	4	–
	sPr:FastR	5	9	5	4
	sPr:SlowR	16	12	5	4
LJ	2Ph	10	28	4	6
	2Ph100%	6	27	4	–
	sPr:FastR	7	27	7	6
	sPr:SlowR	47	29	7	6
TT	2Ph	5	9	5	4
	2Ph100%	4	10	5	–
	sPr:FastR	5	11	5	4
	sPr:SlowR	17	10	5	4
PK	2Ph	8	14	4	5
	2Ph100%	6	14	4	–
	sPr:FastR	6	14	7	5
	sPr:SlowR	20	16	7	5

Table 4.14: No-pruning (nPr) vs. Pruning in Two-Phase.

G	Algorithm	WP	SP	NP	BFS
		#Iter.	#Iter.	#Iter.	#Iter.
TTW	2Ph	5	9	5	4
	noPr:FastR	20	21	23	19
	noPr:SlowR	21	21	27	20
LJ	2Ph	10	28	4	6
	noPr:FastR	20	30	15	14
	noPr:SlowR	46	32	16	15

applies to BFS. Second, due to **PnP**'s aggressive pruning, the number of iterations in the two-phase algorithm are greatly reduced and this limits the potential benefits of bidirectional approach. Table 4.14 shows that the number of iterations of two-phase are much smaller than for no-pruning (noPr) scenario considered in bidirectional BFS. Thus, bidirectional approach is not expected to yield significant additional benefit in the presence of pruning.

Table 4.15: **PnP** average execution times in seconds for 50 queries of each kind on one 8-core 32 GB machine; and **PnP** speedups over Quegel [58] for the same queries on 1 and 4 machines.

Q	G	PnP :: 1 machine			
		WP	SP	NP	BFS
FwdNR	LJ	0.020s	0.020s	0.037s	0.030s
	PK	0.007s	0.009s	0.011s	0.013s
BwdNR	LJ	0.027s	0.028s	0.045s	0.034s
	PK	0.007s	0.007s	0.012s	0.013s
FwdR	LJ	0.035s	0.198s	0.200s	0.136s
	PK	0.013s	0.081s	0.151s	0.054s
BwdR	LJ	0.042s	0.169s	0.214s	0.152s
	PK	0.012s	0.071s	0.148s	0.055s

Quegel :: 1 machine				
WP	SP	NP	BiBFS	BFS
12.9×	13.1×	7.02×	11.0×	8.63×
62.5×	45.2×	23.0×	22.9×	31.0×
679.4×	644.5×	877.3×	24.5×	521.1×
1082.9×	1050.9×	1516.8×	22.8×	611.0×
557.9×	97.7×	192.0×	17.5×	115.6×
630.3×	99.5×	114.7×	22.3×	129.6×
438.0×	110.6×	169.7×	19.3×	101.5×
629.6×	113.2×	112.3×	23.8×	122.2×

Quegel :: 4 machines				
WP	SP	NP	BiBFS	BFS
14.9×	15.1×	8.20×	12.0×	10.2×
58.2×	41.9×	97.1×	24.7×	28.0×
364.8×	344.2×	427.5×	19.2×	290.8×
618.7×	596.1×	3116.1×	23.4×	297.4×
320.8×	56.5×	98.4×	12.0×	67.5×
353.9×	55.2×	229.0×	14.5×	64.4×
262.8×	58.3×	83.0×	13.5×	61.2×
356.6×	63.1×	233.9×	15.5×	61.3×

4.3.3 Quegel vs. PnP

Finally we compare the performance of **PnP** with Quegel, that is aimed at point-to-point iterative graph analytics. Table 4.15 shows the average execution times of **PnP** for 50 queries of each of four kinds on a single 8-core machine, and the average relative speedups achieved by **PnP** over Quegel on 1 and 4 machines (8-cores per machine). Quegel's optimization that combines messages with the same destination vertex is turned on, and results are shown for Quegel's bidirectional BFS (BiBFS) as well as unidirectional BFS.

On an average across all types of queries, **PnP** on a single machine outperforms Quegel on four (one) machines by 8.2× to 3116× (7× to 1517×). Furthermore, it was interesting to observe that Quegel's BFS performed better than it's BiBFS in few cases; nevertheless our prediction and pruning strategies allowed **PnP** to significantly outperform both Quegel's BiBFS and BFS.

4.4 Summary

In this chapter, we present observations on the performance characteristics of point-to-point graph queries. Based on the observations, we have developed **PnP** that incorporates a novel two-phase algorithm for evaluating iterative point-to-point queries involving a single source and destination vertex pair. The algorithm derives its efficiency from selecting the faster direction for evaluating the query and pruning the computation to achieve early termination. Our solution is applicable to streaming graphs as following the solving of one query and before beginning of the next, graph updates can be applied. **PnP** substantially outperforms Quegel, the only previous framework for computing point-to-point queries. In the next chapter, we will combine our knowledge on batched evaluation of point-to-all graph queries and the characteristics of point-to-point query to develop a solution for efficiently processing a batched of point-to-point queries.

Chapter 5

Batched Evaluation of Point-To-Point Queries

In the previous chapter, we have discussed two techniques, online pruning and online direction prediction, that improves the performance of a single point-to-point query. Earlier in the thesis (chapter 2), we have also shown how to explore the synergy between a group of vertex queries to reduce the execution time by amortizing the runtime overhead.

In this chapter, we extend SimGQ (chapter 2) and PnP (chapter 4) to efficiently evaluate a batch of point-to-point queries resulting a system called SimGQ+. To avoid the redundant computation between point-to-point queries, we propose query aggregation, an optimization which merges multiple point-to-point queries sharing the same source vertex into a coarse-grained one-to-many query with a single source vertex and multiple destination vertices. Similarly we may aggregate point-to-point queries sharing the same destination vertex into many-to-one queries. Query aggregation eliminates the shared computation among point-to-point queries and thus delivers better performance. In addition, we generalize pruning and direction prediction, two optimizations proposed in PnP, from the single point-to-point query scenario to the aggregated one-to-many query scenario to further improve performance of batched evaluation.

While query aggregation requires that point-to-point queries share source or destination vertices, the oppotunity for vertex sharing is not uncommon in real world applications. For example, there are use cases in which the analysts may issue many-to-many queries (e.g., in Quegel [58], the authors report that some ecommerce companies are interested in the pairwise queries among a group of important customers). In this case, there are rich opportunities to combine point-to-point queries that share the same source or destination vertices. In addition, the likelihood that a vertex appears as source or destination of a query is not evenly distributed across all vertices in the graph. For example, considering the scenario where an Expedia user search for a flight on a transportation graph in which vertices represent airports and edges represent flights, large airports with higher degrees (e.g., LAX) are expected to appear much more frequent as the start or destination of a flight query compared with local airports with lower degrees. Given the frequent appearance of high degree vertices, we can expect that there are opportunities to aggregate point-to-point queries.

5.1 SimGQ+: Evaluation of a Batch of Point-To-Point Queries

Next we present multiple optimizations for batched evaluation of point-to-point queries which can be combined together seamlessly. We first propose the query aggregation technique and present the motivating data to illustrate the potential of this approach. Next we generalize the pruning technique in PnP to handle coarse-grain one-to-many and many-to-one query generated via aggregation. Finally we discuss extending direction prediction for batched PnP scenario.

Table 5.1: Relationship between Execution Time and Number of Queries.

Aggregation	Point-To-Point	One-To-Many	Many-To-One
# of Queries	50	38	13
Time w/o Batching	21.83s	16.90s	5.58s
Time w/ Batching	4.62s	3.89s	1.66s

5.1.1 Query Aggregation - Exploit Shared Computation

When a group of point-to-point queries are evaluated simultaneously, new opportunities for optimization arise. First of all, we may apply batching for better resource utilization and cache locality, just like what we did for point-to-all queries in the previous section. Moreover, we can develop optimizations specific to multiple point-to-point queries. In this work, we are particularly interested in eliminating shared computations across batch queries. We consider *forward-aggregation* which combines multiple point-to-point queries that share the same source vertex into one point-to-many query which has one source vertex and multiple destination vertices. Similarly we consider *backward-aggregation* which combines the point-to-point queries that share the same destination vertex into a many-to-point query with one destination vertex and multiple source vertices. Such query aggregation reduces the number of queries and can further reduce the computation overhead because it reduces the number of distinct frontiers we need to maintain.

To illustrate the performance benefit from query aggregation, we conducted the following motivating experiment. Fifty point-to-point shortest path queries are selected such that many of them share source vertex or destination vertex. The input graph is soc-LiveJournal [23] with 4847571 vertices and 68993773 edges.

In Table 5.1, we compare the total running times under three different aggregation policies. Point-to-Point is the baseline without aggregation. one-to-many is the result of forward aggregation. many-to-one is the result of backward aggregation. As we can see, both one-to-many and many-to-one generate fewer queries than point-to-point and also run faster than point-to-point which shows that query aggregation is effective and leads to less query time. In addition, by comparing the running time with and without batching, we observe that batching is effective for aggregated queries. Thus batching and sharing can be applied together for better performance.

5.1.2 Adapting Pruning to Multiple One-To-Many Query Scenario

In our prior work PnP [57], we introduced an online pruning optimization to accelerate point-to-point iterative graph algorithms by eliminating the wasteful propagation which are determined not to contribute to the final answer. Pruning is achieved by comparing the new value of an active vertex v (i.e., $v.value$) with the current value of the destination vertex d (i.e., $d.value$). For instance, in the case of the shortest path algorithm, we can safely prune out vertex v from the active vertex frontier if $v.value \geq d.value$. Pruning reduces the amount of vertex propagation and leads to early termination compared with standard iterative graph algorithms.

We can adapt pruning from the scenario of a single point-to-point query to the scenario with a single one-to-many query. Suppose we have an one-to-many query with a single source vertex s and k destination vertices d_1, d_2, …, d_k. We can safely prune out vertex v from the active vertex frontier if $v.value$ cannot contribute to any of the k point-to-point queries which include $s \rightarrow d_1$, $s \rightarrow d_2$, …, $s \rightarrow d_k$. In the case of the shortest path problem, we can prune out v if $v.value \geq max(d_i.value)$ where $i \geq 1$ and $i \leq k$.

We can further generalize pruning from the scenario with a single one-to-many query to the scenario with multiple concurrent one-to-many queries. Let us see how pruning can be combined with batching. Suppose we have l one-to-many queries which originate from source vertices

$s_1, s_2, \ldots, s_l$. Each one-to-many query has k destinations. Let Q_i denote the ith one-to-many query which originates from s_i. Let THRESHOLD[Q_i] denote the pruning threshold for query Q_i. Then THRESHOLD[Q_i] = max(RESULTT[Q_i][d_j]) for all $j \in [1, k]$ where RESULTT[Q_i][d_j] is the tentative shortest path value from s_i to d_j. The pruning condition for vertex v in the multiple query scenario will be $v.value >= max(\text{THRESHOLD}[Q_i])$ for $i \in [1, l]$.

5.1.3 Discussion of Full-Mapping Workload - Breaking Tie between Forward and Backward Aggregation using Direction Prediction

Yan et al. [58] observed that many applications on large graphs simply require computing point-to-point variants of heavyweight computations. As an example, when analyzing a graph that represents online shopping history of shoppers, a business may be interested in all point-to-point queries over an important set of shoppers. Therefore in this work we focus on this kind of full-mapping workload. Moreover, we identify high degree vertices in the input graph as vertices of interest because high degree vertices are usually important vertices in power-law graphs.

Under this workload, the set of input point-to-point queries can be represented as a full mapping from a set of source vertices to the same set of destination vertices (see Figure 5.1). As a result, forward/backward aggregation generate a minimal number of one-to-many/many-to-one queries. In other words, there is a tie between the forward and backward aggregation. Then the question comes which direction to be chosen for aggregation: forward or backward?

Inspired by the direction prediction heuristic developed in PnP [57], we developed a heuristic for predicting the faster direction for a batch of point-to-point queries based on the estimation of work. While in PnP direction prediction is conducted dynamically after a bidirectional

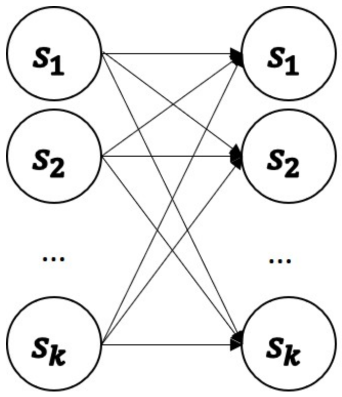


Figure 5.1: Full-Mapping Queries

phase one execution by comparing the frontier sizes from different directions, in this work we predict the direction for a given batch of point-to-point queries statically before iterative computation starts. We make this design choice because we observe that the computation for point-to-point queries between high-degree vertices usually finishes in a few iterations making the dynamic direction selection overhead high relative to the small overall work.

Here is the direction prediction heuristic that we employ. For each point-to-point query in the input batch, we assume the query is forward-fast if the outdegree of the source vertex is greater than the indegree of the destination vertex and similarly assume the query is backward-fast otherwise. If there are more forward-fast queries in the batch, we predict the forward direction is faster and run the batch in the forward direction starting from the source vertices on the original graph. Otherwise we predict the backward direction to be the faster direction, and run the batch in the backward direction starting from the destination vertices on the edge-reversed graph.

An alternative direction prediction heuristic for a batch is to compare the sum of outdegrees of all source vertices and the sum of the indegrees of all destination vertices and select the direction with smaller total degree. However, we found that this metric does not reflect the relative amount of work in different directions as accurate as the first approach because this alternative approach may over-emphasize the importance of a single point-to-point query in the batch and consequently misleadingly hide the impact of other batch queries. For instance, among 16 point-to-point queries in a batch, 15 queries are forward-fast and only 1 query is backward-fast; however the backward-fast query has a source with very high outdegree and a destination with very small indegree and dominate the overall prediction.

5.1.4 Push Style Batched Evaluation of Full-Mapping Point-to-Point Queries

Now we present a detailed algorithm that computes the pairwise property values between each pair of vertices from a set of important vertices using the Push model. In Algorithm 12, function EVALUATEPAIRWISE works as follows. First, function PREDICT (line 5) applies one of the static degree-based heuristics (discussed in Section 5.1.3) to predict the faster direction for the given input query on the given input graph. For k query vertices, there are k^2 corresponding point-to-point queries. Based on the prediction result, the algorithm decides how to aggregate these point-to-point queries into more coarse-grained queries. If the predicted direction is forward, k^2 point-to-point queries are aggregated in the forward direction into k one-to-many queries each of which has a single source vertex and k destination vertices (line 10-13). If the predicted direction is backward, point-to-point queries are aggregated in the backward direction into k many-to-one queries each of which has a single destination vertex and k source vertices (line 15-18). After generating the aggregated queries, function EVALUATEBATCHEDONETOMANY is called to evaluate the aggregated queries as a batch (lines 13 and 18). To benefit from a unified interface for evaluating one-to-many queries and many-to-one queries, the evaluation of many-to-one queries is conducted as follows. We first convert many-to-one queries into one-to-many queries by calling the function REVERSE (e.g., many-

Algorithm 12 Pairwise Evaluation between a Group of Vertices

```
1:  Given: Directed Graph: $G(V, E)$; The Edge-Reverse Graph: $\hat{G}(V, \hat{E})$; and Vertex set QueryVertices which contains
    a group of $k$ vertices
2:  Goal: Compute point-to-point values between each pair of vertices in QueryVertices
3:  function EVALUATEPAIRWISE(QueryVertices($s_1, s_2, s_3, ..s_k$), $G$, $\hat{G}$)
4:      ▷ Predict faster Direction
5:      Prediction ← PREDICT(QueryVertices, $G$)
6:      ▷ Generate Point-to-Point Queries from Input Vertex Set
7:      PTPQueries ← GENPTPQUERIES(QueryVertices);
8:      ▷ Aggregate Point-To-Point Queries and Evaluate in the Predicted Direction
9:      if Prediction = Forward then
10:         ▷ Aggregate Point-To-Point Queries into One-To-Many Queries
11:         OneToManyQueries ← FWDAGGREGATE(PTPQueries)
12:         ▷ Evaluate aggregated Queries
13:         EVALUATEBATCHEDONETOMANY(OneToManyQueries, $G$)
14:     else
15:         ▷ Aggregate Point-To-Point Queries into Many-To-One Queries
16:         ManyToOneQueries ← BWDAGGREGATE(PTPQueries)
17:         ▷ Evaluate aggregated Queries
18:         EVALUATEBATCHEDONETOMANY(REVERSE(ManyToOneQueries), $\hat{G}$)
19:     end if
20: end function
```

to-one query $(\{s_1, s_2, ..., s_k\}, s_i)$ is reversed to an one-to-many $(s_i, \{s_1, s_2, \ldots, s_k\})$. After the conversion, we can get the result of the original many-to-one query by evaluating the reversed one-to-many queries on the edge-reversed graph $\hat{G}$ (line 18).

Algorithm 13 Batched Evaluation of One-To-Many Queries

```
1: Given: Directed Graph Graph(V, E); QueryBatch which is set of k One-To-Many queries: $Q_1(s_1, \{s_1, s_2, s_3, ..s_k\})$,
   $Q_2(s_2, \{s_1, s_2, s_3, ..s_k\}), \cdots Q_k(s_k, \{s_1, s_2, s_3, ..s_k\})$
2: Goal: Evaluate the given batch of One-To-Many queries
3: function EVALUATEBATCHEDONETOMANY( QueryBatch )
4:     ▷ Initialization Step
5:     Initialize RESULTT for QueryBatch
6:     ACTIVE ← { $s_1, s_2, ..., s_k$ }; NEXTTRACK ← $\phi$;
7:     CURRTRACK ← { $(s_i, Q_i) : Q_i(s_i) \in$ QueryBatch }
8:     Initialize THRESHOLD[] for pruning for each $Q_i$ in QueryBatch
9:     ▷ Iterate till Convergence
10:    while ACTIVE $\neq \phi$ do
11:        ▷ Process Active Vertices
12:        ACTIVE ← PROCESSBATCH (ACTIVE, CURRTRACK,NEXTTRACK, RESULTT)
13:        ▷ Update the Pruning Threshold for each One-To-Many Query in the Batch
14:        for all $(s_i, \{s_1, s_2, ..s_k\}) \in$ QueryBatch in parallel do
15:            THRESHOLD[$Q_i$] ← AGGREGATELOOSE(ResultT[$Q_i$][$s_j$]) for $j = 1..k$
16:        end forall
17:        ▷ Prune active frontier using pruning threshold
18:        ACTIVE ← PRUNEACTIVE(ACTIVE, NEXTTRACK, THRESHOLD, RESULTT)
19:        CURRTRACK ← NEXTTRACK; NEXTTRACK ← $\phi$;
20:    end while
21:    return RESULTT
22: end function
```

Let us dive into function EVALUATEBATCHEDONETOMANY which simultaneously evaluates a batch of one-to-many queries over a directed graph $G\ (V, E)$ where the ith query originates at source vertex s_i and has k destination vertices $s_1, s_2, ..., s_k$. The function is explained in Algo-

Algorithm 14 Prune Active Vertex Frontier using Threshold

```
1: function PRUNEACTIVE( ACTIVE, NEXTTRACK, THRESHOLD, RESULTT)
2:   NEWACTIVE ← $\phi$
3:   for all $v \in$ ACTIVE in parallel do
4:     for all $Q_i \in$ QueryBatch do in parallel
5:       ▷ $v$ cannot be pruned if any query needs propagation of new value of $v$
6:       if DONOTPRUNE($v$, $Q_i$) then
7:         NEWACTIVE ← NEWACTIVE ∪ {v}
8:       else
9:         NEXTTRACK ← NEXTTRACK \ {($v$, $Q_i$)}
10:      end if
11:    end forall
12:  end forall
13:  return NEWACTIVE
14: end function
```

rithm 13. It is similar to the simple batching algorithm for point-to-all queries (Algorithm 1 without lines 13-20). The algorithm maintains an ACTIVE vertex set, the combined frontier for all queries in the batch as well as two fine-grained active lists, CURRTRACK and NEXTTRACK, that provides information about which vertex is activated by which query. The RESULTT maintains the results of all the queries for each vertex. What makes a difference is pruning (line 18). After each iteration of propagation, THRESHOLD[Q_i] for pruning is updated for each one-to-many query Q_i from QueryBatch using the tentative query results by applying an aggregation for loose boundary (line 14-16). For instance, AGGREGATELOOSE is max in the case of shortest path and BFS while it is min in the case of widest path and Viterbi algorithm. Pruning threshold is computed as loose boundary rather than tight boundary because we cannot prune a vertex v for a one-to-many query as long as the new value of v may contribute to the value of any (rather than all) destination vertices. With the updated THRESHOLD, the active vertex frontier is pruned by calling the function PRUNEACTIVE (line 18). Function PRUNEACTIVE is described in detail in Algorithm 14. An active vertex v cannot be pruned from the active vertex set if at least one of the queries does not want to prune it (line 6-7). Function DONOTPRUNE (line 6) varies from benchmark to benchmark. For instance, in the case of shortest path problem, DONOTPRUNE returns true if and only if RESULTT$[Q_i][v] <$ THRESHOLD$[Q_i]$.

5.2 Experimental Evaluation

5.2.1 Experimental Setup

For evaluating a batch of point-to-point (PTP) queries, we reused the batching interface from SimGQ and on top of that we implemented optimizations for batched processing of point-to-point queries including query aggregation, dynamic pruning, and direction prediction. We evaluated our techniques using four benchmark applications – SSWP, Viterbi, BFS, and SSSP. We once again used as input the four power-law graphs in Table 4.3. Benchmarks are implemented using the PUSH model. Experiments are conducted on a machine with 32 cores and 512 GB memory as described in Section 4.1.

For each combination of benchmark application and input graph, we evaluate the point-to-point property values between each pair of vertices from a set of query vertices which are the vertices with highest total degrees from the input graph and with both indegree and outdegree above a set default threshold of 500.

Since number of query vertices is an important parameter in this evaluation, in the experiments we vary this number from 4 vertices (i.e., 16 point-to-point queries) to 128 vertices (i.e., 16,384 point-to-point queries). The maximum number of query vertices is set to 64 (i.e., 4,096 point-to-point queries) for TTW graph and 32 (i.e., 1,024 point-to-point queries) for TT graph because of high execution times due to large sizes of these graphs.

5.2.2 Effectiveness of Aggregation and Batching

In this section we present the results of our algorithms to evaluate the effectiveness of both query aggregation and batching. We present the performance of three algorithms for comparison. We refer to the algorithm that employs both query aggregation and batching as Aggregate+Batch. In addition, we also collect the execution time of the algorithm with query aggregation but no batching where we refer to this algorithm as Aggregate. The baseline algorithm evaluates point-to-point queries (with pruning) one by one, we refer to this algorithm as PTP-OneByOne. For all three algorithms, we present the data for the largest number of query vertices for each graph (i.e., 128 for LJ and PK, 64 for TTW, and 32 for TT).

First, let us consider the results of the algorithms for the *forward direction*. Table 5.2 presents the total execution time of the baseline algorithm while Table 5.3 gives the execution times of Aggregate and Aggregate+Batch algorithms as well as their speedups over the baseline algorithm. As we can see, both query aggregation and batching contribute to the speedup significantly. Query aggregation alone gives speedups ranging from 16.46$\times$ (SSWP on TT) to 48.64$\times$(SSSP on PK). When batching is combined with aggregation, the speedups are further boosted to 28.81$\times$ (SSSP on TT) to 166.27$\times$ (SSWP on PK).

We also present the speedups of Aggregate and Aggregate+Batch over the baseline using algorithms running in the backward direction over the edge-reverse graph and compare the

Table 5.2: Running Time (Seconds) of Baseline in Forward Direction. Number of Query Vertices: 128 for LJ and PK, 64 for TTW, and 32 for TT.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	2180	2306	1001	3351
TT	202	229	126	243
LJ	350	413	144	469
PK	625	613	151	567

Table 5.3: Running Time and Speedup of Aggregate and Aggregate+Batch over baseline PTP-OneByOne. In each cell, the Left Number is Execution Time in Seconds while the Right Number is Speedup over Baseline.

Algorithm	SSWP		Viterbi		BFS		SSSP	
	TTW							
Aggregate	106	20.62×	117	19.70×	25	40.29×	159	21.05×
Aggregate+Batch	25	86.17×	28	83.85×	17	60.58×	38	88.36×
	TT							
Aggregate	12	16.46×	13	17.68×	5	24.18×	15	16.66×
Aggregate+Batch	7	30.82×	7	31.41×	3	41.99×	8	28.81×
	LJ							
Aggregate	10	34.03×	13	32.40×	4	33.00×	16	30.16×
Aggregate+Batch	3	121.54×	3	118.12×	2	84.56×	4	118.35×
	PK							
Aggregate	14	46.32×	14	44.56×	3	45.31×	12	48.64×
Aggregate+Batch	4	166.27×	4	158.36×	1	134.17×	4	159.00×

results of backward execution with that of forward execution. Table 5.4 shows the speedup that can be achieved by selecting the faster direction over the slower direction. As we can see, the difference between forward and backward execution is more significant for large graphs with fewer query vertices (i.e., TTW with 32 query vertices and TT with 32 query vertices) while the difference between two directions is smaller for smaller input graphs with larger number of query vertices (i.e., LJ and PK with 128 queries). A possible reason for the difference between TTW/TT and LJ/PK is the difference in their graph structure. In particular, the difference between outdegrees and indegrees of the query vertices of TTW/TT is much larger than that of LJ/PK.

Table 5.4: Speedup of Faster Direction over Slower Direction using Aggregate+Batch.

Algorithm	SSWP	Viterbi	BFS	SSSP
	TTW			
PTP-OneByOne	2.81×	2.75×	2.56×	3.28×
Aggregate	3.35×	3.35×	2.32×	3.61×
Aggregate+Batch	2.24×	2.32×	2.22×	2.25×
	TT			
PTP-OneByOne	3.51×	3.43×	3.38×	3.80×
Aggregate	5.20×	4.80×	3.59×	5.29×
Aggregate+Batch	2.97×	3.04×	2.68×	3.04×
	LJ			
PTP-OneByOne	1.19×	1.11×	1.11×	1.21×
Aggregate	1.27×	1.13×	1.02×	1.13×
Aggregate+Batch	1.18×	1.02×	1.01×	1.07×
	PK			
PTP-OneByOne	1.02×	1.02×	1.00×	1.08×
Aggregate	1.01×	1.04×	1.02×	1.08×
Aggregate+Batch	1.02×	1.04×	1.01×	1.05×

5.2.3 Sensitivity To Number of Query Vertices

Figure 5.2 presents execution times for varying number of query vertices for LJ. The general trend is that speedup increases as the number of query vertices increases. This is because greater amounts of redundant computation can be eliminated via query aggregation and greater amount of runtime overhead is amortized via batching. This is reflected in Figure 5.2 as the gaps between the baseline and Aggregate (for aggregation) and between Aggregate and Aggregate+Batch (for batching) increase as the number of query vertices increases. For instance, as shown in Table 5.5, in the case of SSWP on LJ, the speedup from aggregation and speedup from batching is 2.34× and 1.55× respectively for 4 query vertices. The corresponding speedups grow to 19.18× and 3.57× as the number of query vertices increases to 64.

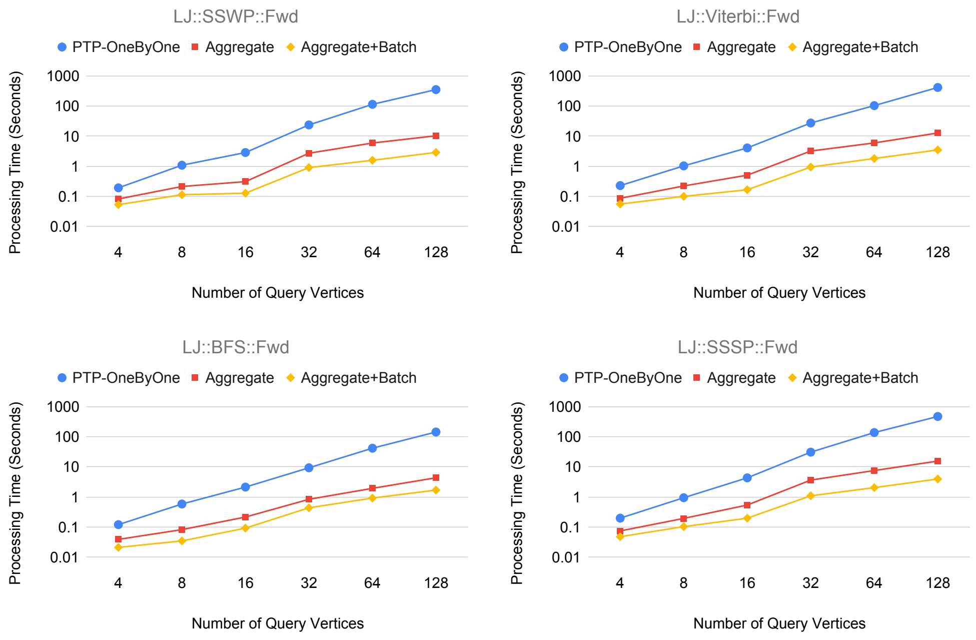


Figure 5.2: Total Query Execution Times of PTP-OneByOne vs. Aggregate vs. Aggregate+Batch.

Table 5.5: Execution Times (Seconds) of SSWP on LJ for Varying # of Query Vertices.

# Query Vertices	4	8	16	32	64	128
PTP-OneByOne	0.19	1.09	2.86	23.62	113.74	350.38
Aggregate	0.08	0.21	0.31	2.72	5.93	10.29
Aggregate+Batch	0.05	0.11	0.13	0.91	1.59	2.88

5.2.4 Accuracy of Direction Prediction

As shown in Table 5.4, the running time of Aggregate+Batch can differ a lot when evaluating in different directions. Thus it is important to figure out the faster direction. Since we observe that the number of iterations can be as small as a few iterations for our workload, we decide to use static direction prediction heuristic in the beginning before the iterative computation starts. We evaluated the following two degree-based heuristics. The underlying assumption is that lower

Table 5.6: Prediction Rate - First Heuristic.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	100.00%	100.00%	100.00%	100.00%
TT	100.00%	100.00%	100.00%	100.00%
LJ	66.67%	100.00%	50.00%	83.33%
PokeC	50.00%	50.00%	33.33%	50.00%

degree indicates less work to do which has been verified in prior work PnP [57]. In the discussion below, k denotes the number of query vertices.

In the first heuristic, for each of the k query vertices we compare their outdegree and indegree. If outdegree is less than indegree, the vote for forward execution increases by one. Otherwise, the vote for backward execution increases by one. After examining every query vertex, the direction with more votes will be selected as the desired direction.

In the second heuristic, for each of the k^2 point-to-point queries we compare the outdegree of the source vertex and the indegree of the destination vertex. If the former is less than the latter, the vote for forward execution increases by one. Otherwise, the vote for backward execution increases by one. After examining all the point-to-point queries, the direction with more votes will be predicted as the faster direction.

For each combination of benchmark application and input graph, we compute the prediction rate based on the overall results on different numbers of query vertices. For instance, in the case of SSWP on LJ, the prediction rate is based on six data points which are collected for six different numbers of query vertices 4, 8, 16, 32, 64, and 128.

Tables 5.6 and 5.7 present the prediction rates for the two heuristics. Both prediction heuristics give very good results for TTW and TT and reasonably good results for LJ and PK.

Table 5.7: Prediction Rate - Second Heuristic.

Graph	SSWP	Viterbi	BFS	SSSP
TTW	100.00%	100.00%	100.00%	100.00%
TT	100.00%	100.00%	100.00%	100.00%
LJ	66.67%	100.00%	50.00%	83.33%
PokeC	66.67%	33.33%	83.33%	100.00%

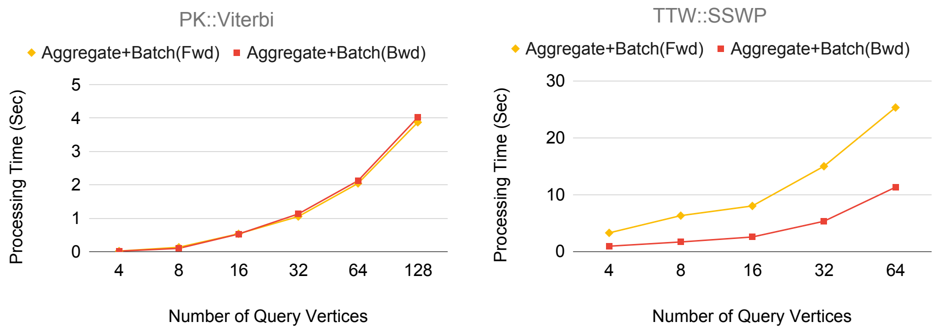


Figure 5.3: Aggregate+Batch (Forward) vs. Aggregate+Batch (Backward) Running Times.

When prediction heuristics work less efficiently, the forward and backward execution times are usually close to each other (e.g., Viterbi on PokeC for which the plots of running time in different directions are shown in Figure 5.3). In contrast, the difference between execution times in forward and backward is usually much more significant when the prediction rate is more accurate (e.g., SSWP on TTW, for which difference between directions is shown in Figure 5.3). Therefore, with our prediction heuristics, even with misdirection, we still get good performance in the end.

5.3 Summary

In this chapter, we developed techniques for simultaneous evaluation of large batches of iterative point-to-point graph queries. By embracing the batching paradigm, the overhead costs of query evaluation are amortized across the input queries. By employing query aggregation, pruning, and direction prediction for point-to-point queries, the cost of computations involving shared workload are amortized across the original batch of queries. Our experiments based upon the Ligra system show that our system yields significant speedups, where all four techniques, batching, aggregation, pruning, and direction prediction, contribute to speedups albeit to different degrees.

Chapter 6

Related Work

This chapter discusses research in the literature that are related to our work. We first present the graph processing frameworks with various architectures including shared-memory systems, distributed systems, and disk-based out-of-core frameworks. And then we discuss frameworks designed for processing multiple queries which is related to our batching paradigm. And then we go over graph databases and query systems because they support point-to-point queries and various indexing techniques. Finally we close up this chapter with streaming graph frameworks.

6.1 Graph Processing Frameworks

There are a number of single machine shared-memory frameworks [1, 41, 36, 19]. Ligra [41] provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. [36] presents a shared-memory based implementations of these DSLs on a generalized Galois system and compares its performance with the original implementations. These frameworks are based on the Bulk Synchronous Parallel (BSP) [45] model. GRACE [51], a shared memory graph processing system, uses message passing and provides asynchronous execution. To efficiently process large graphs our prior work has employed Graph Reduction [19] and built a system on top of Galois. On a single machine large graphs may not fit in memory. Therefore other methods have been proposed for processing extremely large graphs. For a single multicore machine a number of out-of-core processing systems have been recently proposed (GraphChi [21], X-Stream [40], GridGraph [64], DynamicShards [49], Turbograph [14], Flashgraph [63], Bishard [35]). Alternately distributed systems that combine memories of multiple machines to handle large graphs can be used (Pregel [26], PowerLyra [6], PowerGraph [11], GraphLab [24], ASPIRE [46], CoRAL [47]). Recent works show that asynchronous algorithms are more capable of tolerating communication latencies of distributed systems [55, 15, 46, 49].

6.2 Multi Query Frameworks

Recently, MultiLyra [29] and its extensions in BEAD [30] were developed to simultaneously evaluate a batch of iterative graph queries. There are important differences between the algorithms developed in this paper and MultiLyra/BEAD. First, MultiLyra and BEAD are frameworks for distributed systems and hence its emphasis is on amortizing *communication costs* between machines of a cluster while in this paper we show how batching can be deployed on a single multicore shared-memory machine to amortize overhead costs. Second, we show how to *dynamically* identify shared queries and exploit them to amortize computation costs of queries in a *single batch*. MultiLyra presents a limited algorithm that profiles multiple batches to find *fixed* shared queries that it uses to help speedup future batches. Thus, it cannot be used to speedup a single batch of queries and it cannot select shared queries that are customized to the batch being evaluated. Also in [44] authors show that a batch of BFS queries starting from different source vertices can be simultaneously evaluated efficiently. In [17] authors group vertices into multiple batches to reduce message passing and remote memory access in computing pruned landmark labels. However, they do not exploit sharing and are aimed at a specific application.

Congra [34] schedules a group of concurrent queries to fully utilize the memory bandwidth while preventing contention between different queries. It relies upon offline profiling with different number of threads to determine the scalability and memory bandwidth consumption of different graph algorithms on different input graphs. Multiple queries are processed by creating different processes for different queries where each process has suitable number of threads. This approach thus exploits available system resources fully. In contrast, SimGQ does not require offline profiling but is entirely online, lightweight, and enjoys additional benefits from sharing and batching because it does not use multiple processes. Unlike our sharing of computation across queries, Congra does not exploit shared computations across multiple queries in a batch and thus it does not reduce the amount of computation in terms of number of updates or active vertices scheduled. As for batching, we group the updates from different queries on the same vertex together to achieve better cache performance, while Congra cannot do so as execution of each query is decoupled from other queries. Other works on concurrent query processing include CGraph [59] that merely studies the opportunity to share the graph in the context of out-of-core system and Seraph [53] that studies the opportunity to share the graph and emphasizes its capability to help in fault tolerance.

6.3 Graph Databases and Query Systems

The work closely related to our PnP work is Quegel [58]. However, as discussed earlier, it relies upon offline Hub^2 computation that is limited to shortest path queries on unweighted graphs and does not allow graphs to change between queries. All these problems are addressed by **PnP** using dynamic pruning and dynamic direction prediction. Quegel also supports another scenario where on a distributed system a batch of queries are simultaneously solved by efficiently sharing memory and computing resources among the queries. This is different from the scenario we consider – solving a stream of queries on a single machine, and answering each query as quickly as possible. Moreover, their batching algorithm also relies on Hub^2 pre-computation. Note that our technique can benefit from connected components precomputation but we prefer dynamic techniques to avoid disadvantages of precomputation. There are also works that improve performance of specific algorithms (e.g., delta stepping for SSSP [31]).

There has been a great deal of work on graph based query languages (e.g., Gremlin [39]) and query support in graph databases (e.g., Neo4J and DEX [8, 25, 2]) that enable graph traversals and joins via lower-level graph primitives (e.g., vertices, edges, etc.). However, they are not efficient for iterative graph algorithms over large graphs. Their strength lies in their ability to program wide range of queries. They are more suitable for neighborhood queries [37, 50, 38, 28] including query decomposition and incremental processing devoted to pattern matching [54, 52]. In [37] authors develop algorithms for efficiently answering k-nearest neighbor queries (k-NN) that prunes the search to limit the graph that is explored. In [50] authors develop a fast neighborhood graph search algorithm using a new data structure called the bridge graph constructed from a large number of bridge vectors. In [28] a compressed representation of social networks is proposed to facilitate computation of neighbor queries. NScale [38] is another system for neighborhood-centric analytics on large graphs including analysis tasks such as ego network analysis, social circles, personalized recommendations, link prediction, influence cascades, and motif [32] counting. GraphX [12] supports both kinds of graph operators (i.e., neighborhood aggregation as well as join and structural operators) and iterative algorithms.

In [62] authors present SPath, an indexing method which leverages decomposed shortest paths around neighborhood of each vertex as basic indexing unit, to accelerate subgraph matching queries. SPath performs very large amounts of precomputation (to enable the optimization) before it can begin to answer queries. In fact the overhead is substantial – comparable to solving a very large number of queries. SimGQ requires no precomputation, rather it identifies shared computation for a batch of queries such that performing it once leads to net reduction in execution time.

6.4 Streaming Graph Frameworks

There are multiple graph processing frameworks that target at solving graph analytics problems in the streaming graph scenarios. Tornado [43] takes a snapshot of the current version of the graph and creates a separate branch to compute the query results using incremental computation on the snapshot graph. Kineograph [7] is a distributed streaming graph processing framework that focuses on incremental computation along with push and pull models. Naiad [33] employs iterative and incremental algorithms. Kickstarter [48] and Graphbolt [27] track the dependencies to enable fast query processing on streaming graphs. The above systems focuses on processing fixed queries replying on a priori knowledge on the user query, while the generalized incremental computation we proposed in Chapter 3 aim at handling vertex queries that originate from arbitrary source vertices.

In addition to the above systems, there are also existing works that focus on efficient ingestion of graph updates. STINGER [10] uses a novel data structure to enable fast insertions and deletions. Aspen [9] proposes a graph representation using a compressed purely-functional tree data structure that enables efficient updates to the graph. As mentioned earlier, our work on generalized incremental computations adopts Aspen as the streaming graph engine.

Chapter 7

Conclusions and Future Work

7.1 Contributions

In this thesis, we study and leverage the synergy across a group of queries to improve the overall throughput when simultaneously evaluating a batch of point-to-all or batch of point-to-point queries. We first explore the opportunity to amortize the runtime overhead and reduce the computational cost for evaluating a batch of point-to-all graph queries on static graphs, and then extend the system for streaming graph scenarios. After that, we study the optimizations for a batch of point-to-point queries. Since point-to-point graph queries are a class of workload which have not been widely studied in the literature, before moving to the batched version, we start with a study on how to leverage the performance characteristics of point-to-point queries, which are different from that of point-to-all queries, to accelerate a single point-to-point graph query, and then build upon the observations for a single point-to-point query to develop further optimizations that take effect under the batching scenario.

Batched Evaluation of Point-To-All Queries

We presented a system that optimizes simultaneous evaluation of a group of point-to-all queries that originate at different source vertices. The performance benefits of our system are achieved via batching and sharing. Batching fully utilizes system resources to evaluate a batch of queries and amortizes runtime overheads incurred due to fetching vertices and edge lists, synchronizing threads, and maintaining computation frontiers. Sharing dynamically identifies shared queries that greatly represent the shared subcomputations in the evaluation of different queries in a batch, evaluates the shared queries once, and then uses their results to accelerate the evaluation of all queries in the batch. We also generalized the incremental computation by adapting batching and sharing to the streaming graph scenarios. More specifically, we maintain the results of a small number of preselected shared queries using conventional incremental computation for fixed queries and then share the results of the shared queries with arbitrary user queries upon graph mutation. Meanwhile, both the computation of shared queries and computation of user queries can be evaluated in a batched fashion resulting in a higher throughput.

Batched Evaluation of Point-To-Point Graph Queries

We first studied the characteristics of point-to-point queries that differentiates them from the point-to-all queries. Based on the observations, we developed a two-phase algorithm with two novel features: online pruning of graph exploration that eliminates propagation from vertices that are determined to not contribute to a query's final solution; and dynamic direction prediction for solving the query in either forward or backward direction as the cost in two directions can differ greatly. For batched evaluation of point-to-point queries, we developed a new query aggregation technique that reduces the shared subcomputation across point-to-point queries that share the same source or destination queries. The query aggregation optimization can be applied together with batching, and the aforementioned optimizations for a single point-to-point query for better performance.

7.2 Future Work

Batched Evaluation of Point-To-All Queries

In this thesis, we assume that all the input queries form a single batch for batched evaluation. However, from the experimental results on salability w.r.t. query batch size, we can see that the performance gain from larger batch size becomes less significant as the batch size exceeds a threshold (e.g., 64 for multiple combinations of input graphs and benchmark applications). In addition, we have discovered in Chapter 2 that different user queries may prefer different shared queries. Thus, it would be an interesting topic to study whether we can achieve higher throughput by grouping queries into multiple mini-batch so that we may apply more fine-tuned optimizations which are better tailored for each mini-batch to reduce the runtime overhead and computation cost.

Batched Evaluation of Point-To-Point Queries

We have carefully studied two different workloads for point-to-point queries – a single point-to-point queries and the full-mapping pairwise point-to-point queries between a group of vertices. It would be interesting to study more general batching workloads with weaker assumptions on the point-to-point queries that form a batch. One potential direction is to study the algorithm to minimize the number of one-to-many and many-to-one queries after aggregating queries that share the same source vertex or same destination vertex which is trivial for full-mapping workload (because forward-only and backward-only are optimal in this case), but complicated for the general workload. Another possible direction is to explore the sharing opportunities for point-to-point queries. While there are some existing works on indexing techniques for point-to-point queries (e.g., Hub^2 [18]), most of them focus on BFS or SSSP. There might be space for optimizations that work across a broader spectrum of graph algorithms.

Bibliography

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis (SC), pages 1-11, 2010.

[2] A.B. Ammar. Query Optimization Techniques in Graph Databases. In Int. J. of Database Management Systems, Vol.8, No.4, August 2016.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In KDD, pages 44-54, 2006.

[4] S. Beamer, K. Asanovic, and D. Patterson. Direction-Optimizing Breadth-First Search. In ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis (SC), 10 pages, November 2012.

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

[6] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In European Conference on Computer Systems (EuroSys), Article 1, 15 pages, 2015.

[7] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In EuroSys, pages 85–98, 2012

[8] DEVELOPERS. Neo4J. Graph NoSQL Database, 2012.

[9] L. Dhulipala, G. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 918–934, 2019

[10] D. Ediger, R. McColl, J. Riedy and D. A. Bader. STINGER: High performance data structure for streaming graphs In IEEE Conference on High Performance Extreme Computing, pp. 1-5, 2012

[11] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In OSDI'12.

[12] 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 Symp. on Operating Systems Design and Implementation (OSDI), pages 599-613, 2014.

[13] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In WWW, pages 505-514, 2013.

[14] W-S. Han, S. Lee, K. Park, J-H. Lee, and M-S. Kim, and J. Kim and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pages 77-85, 2013.

[15] A.F. Harshvardhan, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 27-38, 2014.

[16] G. He, H. Feng, C. Li, and H. Chen. Parallel simrank computation on large graphs with iterative aggregation. In KDD, pages 543-552, 2010.

[17] R. Jin, Z. Peng, W. Wu, F. Dragan, G. Agrawal, and B. Ren. Parallelizing pruned landmark labeling: dealing with dependencies in graph algorithms. In ACM ICS, Article 11, 1-13, 2020.

[18] 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.

[19] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu. Efficient Processing of Large Graphs via Input Reduction. In ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 245-257, May-June 2016.

[20] 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.

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

[22] J. Lember, D. Gasbarra, A. Koloydenko, and K. Kuljus. Estimation of Viterbi Path in Bayesian Hidden Markov Models. arXiv:1802.01630, pages 1-27, Feb. 2018.

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

[24] 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).

[25] P. Macko, D. Margo, and M. Seltzer. Performance Introspection of Graph Databases. In ACM International Systems and Storage Conferences (SYSTOR), 10 pages, 2013.

[26] 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, 2010.

[27] M. Mariappan and K. Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. EuroSys, pages 1–16, 2019.

[28] H. Maserrat and J. Pie. Neighbor Query Friendly Compression of Social Networks. In ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), 9 pages, 2010.

[29] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable Distributed Evaluation of batches of Iterative Graph Queries. In IEEE International Conference on Big Data, pages 349-358, Dec. 2019.

[30] 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, Dec. 2020.

[31] U. Meyer and P. Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. In Journal of Algorithms, 49:114-152, 2003.

[32] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. Science, 298.5594, pages 824-827, 2002.

[33] D. Murray, F. McSherry, R. Isaacs, M. Isard, P.Barham, and M. Abadi. Naiad: a timely dataflow system. In SOSP, pages 439–455, 2013.

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

[35] K. Najeebullah, K. U. Khan, W. Nawaz and Y-K. Lee. BiShard Parallel Processor: A Disk-Based Processing Engine for Billion-Scale Graphs. In International Journal of Multimedia and Ubiquitous Engineering, 9(2):199-212, 2014.

[36] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM SOSP*, pages 456-471, 2013.

[37] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-Nearest Neighbors in Uncertain Graphs. In Proc. of the VLDB Endowment, 2010.

[38] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. In VLDB, 7(13):1673-1676, 2014.

[39] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In Symp. on Database Prog. Languages, pages 1-10, 2015.

[40] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In 24th ACM Symposium on Operating Systems Principles (SOSP), pages 472-488, 2013.

[41] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In ACM PPoPP, pages 135-146, 2013.

[42] 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.

[43] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In SIGMOD, pages 417–430, 2016.

[44] 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. VLDB Endowment, 2015.

[45] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM (CACM), 33(8):103-111, 1990.

[46] 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, October 2014.

[47] 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, April 2017.

[48] K. Vora, R. Gupta, and G. Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 237-251, April 2017.

[49] K. Vora, G. Xu, and R. Gupta. Load the Edges You Need: A Generic I/O Optimization for Distributive Disk-based Graph Algorithms. USENIX Annual Technical Conference (ATC), pages 507-522, June 2016.

[50] J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo. "Fast Neighborhood Graph Search using Cartesian Concatenation," In International Conference on Computer Vision (ICCV), pages 2128-2135, 2013.

[51] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In Conference on Innovative Data Systems Research (CIDR), pages 3-6, 2013.

[52] F. Wenfei , L. Jianzhong, L. Jizhou , T. Zijing , W. Xin and W. Yinghui. Incremental Graph Pattern Matching. In ACM SIGMOD International Conference on Management of Data, pages 925-936, 2011.

[53] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai Seraph: an efficient, low-cost system for concurrent graph processing. In HPDC, 2014.

[54] S. Yang, X. Yan, B. Zong and A. Khan. Towards effective partition management for large graphs. In ACM SIGMOD International Conference on Management of Data, pages 517-528, 2012.

[55] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 194-204, 2015.

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

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

[58] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng and W. Ng. Quegel: A General-Purpose Query-Centric Framework for Querying Big Graphs. In Proc. VLDB Endowment, 9(7):564-575, 2016.

[59] 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 ATC, 2018.

[60] Y. Zhang, M.Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. GraphIt: a high-performance graph DSL. In PACM 2, OOPSLA, 2018.

[61] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In ACM ASPLOS, pages 608-621, 2018.

[62] P. Zhao and J. Han. On Graph Query Optimization in Large Networks In Proc. VLDB, VLDB Endowment, Vol. 3, 1-2, pages 340-351, 2010.

[63] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In 13th USENIX Conference on File and Storage Technologies (FAST), pages 45-58, 2015.

[64] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In USENIX Annual Technical Conference (ATC), pages 375-386, 2015.