iTurboGraph: Scaling and Automating Incremental Graph Analytics

Seongyun Ko[†] Taesung Lee[†] Kijae Hong[†] Wonseok Lee[†]
In Seo[†] Jiwon Seo[‡] Wook-Shin Han[†§]
POSTECH, Korea[†], Hanyang University, Korea[‡]
{syko, tslee, kjhong, wslee, iseo, wshan}@dblab.postech.ac.kr[†], seojiwon@hanyang.ac.kr[‡]

ABSTRACT

With the rise of streaming data for dynamic graphs, large-scale graph analytics meets a new requirement of *Incremental Computation* because the larger the graph, the higher the cost for updating the analytics results by re-execution. A dynamic graph consists of an initial graph G and graph mutation updates ΔG of edge insertions or deletions. Given a query Q, its results $Q(G)$, and updates for ΔG to G, incremental graph analytics computes updates ΔQ such that $Q(G \cup \Delta G) = Q(G) \cup \Delta Q$ where $\cup$ is a union operator.

In this paper, we consider the problem of large-scale incremental neighbor-centric graph analytics (*NGA*). We solve the limitations of previous systems: lack of usability due to the difficulties in programming incremental algorithms for *NGA* and limited scalability and efficiency due to the overheads in maintaining intermediate results for graph traversals in *NGA*. First, we propose a domain-specific language, L_{NGA}, and develop its compiler for intuitive programming of *NGA*, automatic query incrementalization, and query optimizations. Second, we define *Graph Streaming Algebra* as a theoretical foundation for scalable processing of incremental *NGA*. We introduce a concept of *Nested Graph Windows* and model graph traversals as the generation of walk streams. Lastly, we present a system *iTurboGraph*, which efficiently processes incremental *NGA* for large graphs. Comprehensive experiments show that it effectively avoids costly re-executions and efficiently updates the analytics results with reduced IO and computations.

ACM Reference Format:
Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. iTurboGraph: Scaling and Automating Incremental Graph Analytics. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457243

1 INTRODUCTION

Real-world graphs are enormous, and they evolve constantly over time [3, 18, 26, 65]. Modern applications require processing of such dynamic graphs in a timely manner [19, 41, 51, 56, 65, 76]. As many of the existing graph analytics systems focus on processing *one-shot* analytics over the static graphs, which do not change over time [27, 39, 63, 71, 81, 88], they can exhibit significant overhead for dynamic graphs due to frequent re-execution of the analytics for each change to the graphs [12, 51, 52, 77].

Table 1: Comparisons of the state-of-the-art systems.

System	Performance		Usability	
	Scalability	Efficiency	Automatic Query Incrementalization	Neighbor-centric Graph Analytics
Differential Dataflow [52]	X	Δ	O	O
GraphInc [12]	X	Δ	O	Δ
GraphBolt [51]	X	Δ	X	X
KickStarter [77]	X	O	X	X
iTurboGraph	O	O	O	O

To avoid costly re-executions under changes to the graphs, incremental graph analytics systems have been studied [12, 51, 52, 77, 84]. Note that while there are a great many works on dynamic graph store [11, 19, 20, 35, 41, 49], they do not provide query processing using incremental computations.

Existing systems fail to process incremental graph analytics for large-scale graphs, since they have to maintain the intermediate results of the previous computations. For example, Differential Dataflow [52] maintains the join results for generated messages and graph traversals (i.e., joined edges). Similarly, GraphInc [12] relies on maintaining all vertex states and generated messages during the previous execution into disk. The enormous intermediate results can also lead to poor efficiency due to the high IO costs.

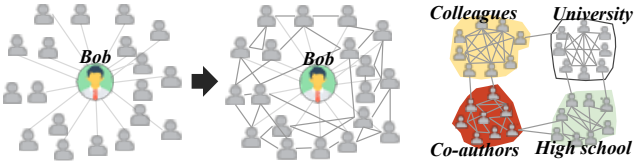
Automatic query incrementalization must be supported. The manual approaches for query incrementalization are error-prone and hard to reason about the correctness [30, 31, 69]. GraphBolt [51] and KickStarter [77] require users to program the incremental computation logic using the provided APIs. The user should implement various functions for updating graph mutations, checking changes in vertex attribute values, and propagating the changes.

We consider targeting *Neighbor-centric* graph analytics (*NGA*) [39, 47, 60, 80]. *NGA* is a generalization of vertex-centric graph analytics in that a user can program the computations over the multi-hop neighbors around each vertex. Since the vertex-centric systems, such as Pregel [82], do not directly support the *NGA*, users need to encode neighbors of vertices into messages as a typical workaround. It is known that such a method limits the scalability and efficiency due to the enormous intermediate results sizes [39, 60, 68, 87]. Table 1 compares the existing systems with *iTurboGraph*. The symbols {O, Δ, X} represent their comparative advantages.

[§]Corresponding author

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457243



(a) *Bob*'s one-hop (left) and two-hop (right) networks. (b) *Communities*.

Figure 1: An example of a social network.

We present a system *iTurboGraph* which consists of a language, compiler, and runtime engine for incremental *NGA*. *iTurboGraph* supports a domain-specific language, namely L_{NGA}, as an intuitive programming interface for *NGA*. For a given L_{NGA} program, its compiler automatically generates one-shot and incremental query plans consisting of *Graph Streaming Algebra* operations.

We propose *Graph Streaming Algebra* (*GSA*) as an extension of the streaming processing models [4, 5] for scalable and efficient processing of *NGA*. We regard the graph data on disk as streams and model the graph traversal for *NGA* as enumeration of walks, which avoids maintaining intermediate results. For efficient enumeration, we introduce the *Nested Graph Windows*. We use *GSA* as an intermediate representation of a query. We derive the incrementalization rules for *GSA* and generate incremental query plans.

We develop a runtime engine and present key optimizations for scalable and efficient execution of incremental *NGA*. The compiler applies query optimizations to query plans, and the runtime engine applies some run-time optimizations. We also present our dynamic graph store, which exploits a *delta*-based maintenance strategy for graph mutations and changes in vertex attributes values.

We conduct extensive experiments and demonstrate that incremental graph analytics effectively reduces the amounts of IO and computations by avoiding costly re-executions. We show that *iTurboGraph* efficiently processes the incremental graph analytics with large-scale graphs for vertex- and neighbor-centric analytics.

2 OVERVIEW

Neighbor-centric Graph Analytics (*NGA*). As a generalization of vertex-centric analytics, *NGA* performs computations within multi-hop neighborhood of each vertex. Therefore, *NGA* covers the vertex-centric analytics. Figure 1 (a) at left shows one-hop network of *Bob* in a social network. The vertex-centric analytics describes computations over one-hop neighbors around each vertex such as PageRank for ranking [32] and finding connected components [82].

NGA can be used to investigate the local structures around each vertex. For example, the triangles in a social graph show the cohesive relationship between friends; *friends of friends tend to be friends themselves* [78]. These local structures are basic components and used for downstream analytics tasks. For example, local clustering coefficient counts the number of triangles around each vertex and assesses the connectivity of its neighborhoods [33]. As in Figure 1 (a) at left, when considering only the one-hop neighbor relationship,

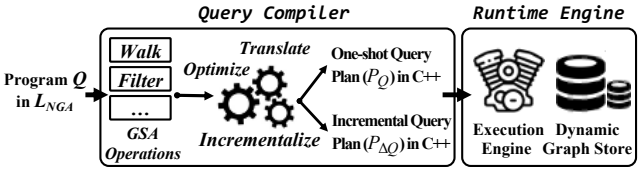


Figure 2: Overview of *iTurboGraph*.

```
1. Compute(u, recv_msgs):
2.  For v in u.nbrs()
3.   If superstep() == 0
4.    If (u.vid < v.vid)  // u1 < u2
5.     Message msg(u.vid);
6.     sendMessage(v, msg);
7.   Else If supertsep() == 1
8.    If (u.vid < v.vid)  // u2 < u3
9.     For msg in recv_msgs
10.     msg.append(u.vid);
11.    sendMessage(v, msg);
12.  Else If superstep() == 2
13.   For msg in recv_msgs
14.    If (v.vid == msg[0]) cnts++;
```

```
1. Traverse(u1) {
2.  For u2 in u1.nbrs Where (u1 < u2)
3.   For u3 in u2.nbrs Where (u2 < u3)
4.    For u4 in u3.nbrs Where (u4 == u1)
5.     cnts.Accumulate(1);
6. }
```

Figure 3: Triangle Counting (TC) program in the vertex-centric programming model (left) and in L_{NGA} (right).

it is hard to assess the characteristics of the network. Figure 1 (a) at right additionally shows the connections between *Bob*'s friends (*Bob*'s two-hop network). Figure 1 (b) shows community detection results using the local clustering coefficient [22]. The discovered communities show the cohesive structures and can be used for feed recommendation and link prediction [70, 74].

iTurboGraph. *iTurboGraph* consists of the query compiler and distributed runtime engine (Figure 2). The query compiler takes as an input an analytics program Q in L_{NGA} and generates a one-shot and incremental query plans: P_Q and $P_{\Delta Q}$. The runtime engine executes P_Q or $P_{\Delta Q}$ over the dynamic graph store.

L_{NGA} is designed to ease the difficulty of programming *NGA*; for example, its nested for-loop with constraints simplifies writing k-hop neighbor traversals. Figure 3 shows triangle counting algorithm implemented in vertex-centric model (left) and in L_{NGA} (right). The vertex-centric model requires encoding and decoding two different kinds of messages over three supersteps. However, in L_{NGA} the 3-hop traversal is intuitively expressed with the nested for-loops.

Given Q, the compiler generates P_Q and $P_{\Delta Q}$. It first compiles Q into P_Q of *GSA* operations. By applying the query incrementalization rules to P_Q, it automatically generates $P_{\Delta Q}$. Then, it applies a set of optimizations. For example, finding common neighbors of some vertices with the nested for-loops are frequently used, which our compiler converts to a for-loop exploiting a multi-way intersection over the adjacency lists of the vertices. The compiler also incorporates the techniques similar to those in relational databases such as join reordering and semi-join reductions [46, 73].

The execution engine runs either P_Q or $P_{\Delta Q}$ performing several runtime optimizations. The scheduling is particularly important for efficient processing of $P_{\Delta Q}$. For example, while $P_{\Delta Q}$ consists of multiple sub-queries (Section 5.1), processing them individually may result in poor performance due to repeated IO. The execution engine jointly schedules the sub-queries to avoid the repeated IO.

The dynamic graph store supports the efficient query execution of the engine. To avoid costly in-place updates to the graph data on disk, we maintain the changes in vertex attribute values or graph mutations as *deltas*. To prevent the deltas from growing indefinitely, we merge them based on our cost-based maintenance strategy.

3 L_{NGA}

We propose a domain-specific language, L_{NGA}, an imperative programming interface for *NGA*. In L_{NGA}, a user can express the graph traversals within multi-hop neighborhoods around each vertex using nested for-loops. We explain its semantics and syntax.

Figure 4 shows the type definitions and execution semantics of L_{NGA} programs. The program first defines two types – vertex

Vertex ($attribute1 : type1$, $attribute2 : type2$, ...)
GlobalVariable ($attribute3 : type3$, $attribute4 : type4$, ...)

foreach *vertex* $u \in V$ **do**
  | INITIALIZE(u);
while *ActiveVertexExist()* **do**
  foreach *vertex* $u \in V_{active}$ **do**
    | TRAVERSE(u);
  GlobalBarrier(); // the synchronization point in BSP model.
  foreach *vertex* $u \in V_{accm}$ **do**
    | UPDATE(u);

Figure 4: Type definitions and execution semantics of L_{NGA} Programs.

type and global variable type. The program also implements three user-defined functions (UDFs): INITIALIZE, TRAVERSE, and UPDATE. INITIALIZE sets initial values to vertex attributes; this function is executed once before any other UDFs are invoked. After the initialization, the execution engine iteratively invokes TRAVERSE and UPDATE until no active vertex exists. TRAVERSE performs graph traversals starting from vertex u in the active vertex set V_{active} and value aggregations to neighbors' attributes of *accumulator types*. The global barrier after TRAVERSE is the global synchronization point in the Bulk Synchronous Parallel (BSP) model [75, 82]. For accumulators updated in TRAVERSE, UPDATE is invoked for their corresponding vertices (V_{accm}) in Figure 4 is the set containing those vertices) to update the attribute values using the data stored in the accumulators. UPDATE performs updates to vertex attribute values and vertex activations for the next superstep.

L_{NGA} supports primitive, accumulator, and composite data types. We provide five primitive data types: {*bool*, *int*, *long*, *float*, *double*}. Accumulator types are defined as *Accm<primitive type*, *operation>*, where *primitive type* is one of the five primitive types and *operation* is any operator in *Abelian monoid* [43], such as SUM and MIN. Because accumulators are intended to store data temporarily during TRAVERSE and UPDATE, their values are initialized as the identity element of the operator at the beginning of each superstep. In order to facilitate various machine learning tasks, we also support composite types, such as *Array*; *Array<type*, *size>*.

L_{NGA} supports the vertex type and global variable type. A vertex holds the data for each vertex, and the global variable stores the data shared for all vertices. For example in Figure 5, the top shows the type definitions for PR and TC. The vertex and global variable type is denoted as Vertex and GlobalVariable, respectively. L_{NGA} pre-defines the essential vertex data: id (*id:long*), activation status (*active*: *bool*), degrees (*out_/in_/degree*: *int*), and adjacency lists (*out_/in_/nbrs*: *Set<id*: *long>*). For the pre-defined data, users

```
1.  Vertex (id, active, out_nbrs,
            out_degree, rank: float,
            sum: Accm<float, SUM>)
2.
3.  Initialize (u):
4.   u.rank = 1;
5.   u.active = true;
6.
7.  Traverse (u):
8.   Let val = u.rank / u.out_degree;
9.   For v in u.out_nbrs
10.   v.sum.Accumulate(val);
11.
12. Update (u):
13.  Let val = 0.15/V + 0.85*u.sum;
14.  If (Abs(val - u.rank) > 0.001)
15.   u.rank = val;
16.   u.active = true;
```

```
1.  Vertex         (id, active, nbrs)
2.  GlobalVariable (cnts: Accm<long, SUM>)
3.
4.  Initialize (u1):
5.   u1.active = true;
6.
7.  Traverse (u1):
8.   For u2 in u1.nbrs Where (u1 < u2)
9.    For u3 in u2.nbrs Where (u2 < u3)
10.    For u4 in u3.nbrs Where (u4 == u1)
11.     cnts.Accumulate(1);
12.
13. Update (u1):
```

Figure 5: Programs in L_{NGA}: PageRank (PR) at left and Triangle Counting (TC) at right.

specify only the names of the data to be used as shown in Figure 5. For the rest, users need to define the data with name and type.

L_{NGA} provides five imperative statements used to describe application logic: LET for binding an expression to a variable; ASSIGN for assigning the value of an expression to a variable; ACCUMULATE for accumulating the value of an expression to a variable; FOR for iterating over a set or range; and IF-ELSE for a branch. Table 2 summarizes them; the second column shows the syntax for the statements. *arith_expr* (or *bool_expr*) represents arithmetic (or boolean) expression. *set_var* is a set type variable, such as the neighbors of a vertex u; $u.nbrs$. We explain the third column in Section 4.4.

Users can easily program both vertex-centric and neighbor-centric analytics using L_{NGA}. Figure 5 (left) shows an implementation of PR. INITIALIZE(u) sets the initial values to attributes *rank* and *active* of u; the latter is used for vertex activation. TRAVERSE(u) performs graph traversals to one-hop neighbors of u. It iterates neighbors v of u (Line 9) and updates their accumulator attribute *sum* (Line 10). Here, it accumulates the value of *val* using the accumulation function SUM. UPDATE(u) updates the *rank* of u using the accumulated value at *sum* and performs vertex activation.

Figure 5 (right) implements TC. INITIALIZE activates all vertices in the graph. TRAVERSE($u1$) performs multi-hop graph traversals around $u1$ using three nested for-loops; from $u1$ to $u2$, $u3$, and $u4$ (Line 8-10). The ordering constraint ($u1 < u2 < u3$) is used to ensure that the same triangle is counted exactly once. For each triangle found, it accumulates one to *cnts* (Line 11). UPDATE is empty and we run TC only for a single superstep. Remark that the implementation in the vertex-centric model (Figure 3) requires multiple supersteps of encoding and decoding messages to express the traversals around multi-hop neighbors around each vertex.

L_{NGA} is an extension of the Pregel model [50, 51] which supports the BSP model. The BSP model runs a user program along a succession of supersteps (i.e. iterations) until convergence, which

Table 2: Statements in L_{NGA} and their corresponding algebra expressions where S represents the output stream of an outer query and T indicates the output stream for data modification operators.

Statements	Syntax	Algebraic Expression
LET	Let *var* = *arith_expr*;	Bind the variable *var* to the expression *arith_expr*
ASSIGN	*id.attr* = *arith_expr*;	$T \leftarrow_{id,attr} \pi_{x.id,arith\_expr \text{ as } attr}(S)$
ACCUMULATE	*id.attr.Accumulate*(*arith_expr*);	$T \uplus_{id,Accm(attr)} \pi_{x.id,arith\_expr \text{ as } attr}(S)$
	attr.Accumulate(*arith_expr*);	$T \uplus_{Accm(attr)} \pi_{arith\_expr \text{ as } attr}(S)$
FOR	For (*var*) in (*set_var*) Where (*bool_expr*) { ... }	$S\ \alpha\ (\sigma_{bool\_expr}\ (\rho_{var/Key(S')}\ (S')))$ where S' is a stream for *set_var*
IF-ELSE	If (*bool_expr*) { ... } Else { ... }	$\sigma_{bool\_expr}(S) \cup \sigma_{\neg bool\_expr}(S)$

can be viewed as a fixed-point computation. Thus, many representative graph algorithms can be expressed with this model. However, unlike Differential Dataflow, Pregel as well as *iTurboGraph* does not support arbitrary nested iterations, which can be expressed with nested fixed-point computation operators [52, 55]. Some complex graph algorithms could require arbitrary nested iterations for their efficient execution. It is also intended that *iTurboGraph* restricts the aggregation to be Abelian monoid to enable scalable and automatic incremental *NGA*. Supporting the nested iterations and general aggregation is beyond the scope. As future work, for the nested iterations, we consider supporting either the *master.compute*() API in [66] or cyclic computation graphs in [52, 55]. We also consider supporting user-defined aggregation APIs for more general aggregation and its incremental maintenance similarly in [57].

4 GRAPH STREAMING ALGEBRA

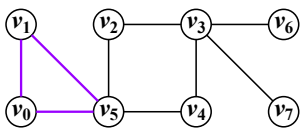
For scalable processing of *NGA*, we extend streaming processing models [4, 5] and define *Graph Streaming Algebra* (*GSA*). We model the graph stored on disk as streams and the graph traversals for *NGA* as enumeration of *walks* over streams. To efficiently enumerate the walks with a fixed amount of memory, we introduce the concept of *Nested Graph Windows* and two stream operators, namely *Window Seek* and *Window Join*.

GSA models a graph as a directed graph $G(V, E)$ where V and E represent the set of vertices and edges, respectively. Each vertex can have attributes. An undirected graph is modeled as a directed graph with pairs of edges in both directions. We model the graph as a simple graph where duplicates are not allowed. A walk of length k is a sequence of $(k+1)$ vertices in V, $(u_1, ..., u_{k+1})$, connected by the edges in E; (u_i, u_j) in E for some i and j such that $(1 \le i < j \le k+1)$.

4.1 Graph as Stream

A stream S is a sequence of tuples. Each tuple r_m has one or more columns of data and multiplicity $m \in$ {-1, 1}. By allowing the multiplicity to be positively or negatively valued, we can represent the insertion and deletion under the same data model, which is widely used and well understood [10, 29, 40]. For example, a graph mutation, such as insertion or deletion of an edge e, can be modeled as e_1 or e_{-1}. We denote a tuple r_1 simply by r.

We model the graph data stored on disk as streams. As we model the graph as a simple graph where duplicates are not allowed, the multiplicity of a vertex and edge tuple is either 1 or -1. Then, like [39], we regard a sequence of vertex tuples and a sequence of edge tuples as a vertex stream (*vs*) and an edge stream (*es*), respectively. Here, a vertex tuple consists of a vertex ID and its attribute values. An edge tuple consists of a source and destination vertex IDs. Then, we define a graph stream (*gs*) as a pair of *vs* and *es*; $gs = (vs, es)$. Conceptually, a graph stream $gs = (vs, es)$ represents a subgraph stored on disk. While we store the graph partitioned as in [39], we use a single partition for ease of explanation.



(a) Graph G_0.

vs	*es*
<v_0, 2>	<v_0, v_1>
<v_1, 2>	<v_0, v_5>
…	<v_1, v_0>
	<v_1, v_5>
	…

(b) vs and es for G_0.

Figure 6: A datagraph G_0 and its graph stream $gs = (vs, es)$.

Figure 6 shows an example input graph G_0 (left) and the corresponding graph stream (right). We consider attribute values of V to be a vertex stream. For example, Figure 6 (b) shows the vs of the tuples; each with the vertex ID and *degree* attribute value. The first and second tuples in vs represent $v_0.degree = 2$ and $v_1.degree = 2$, respectively. We regard the edge lists as the edge stream es, as shown in Figure 6 (b). The vs and es are sorted by the vertex ID.

4.2 Nested Graph Window

Graph traversals can incur high IO costs due to random access and repeated [38, 45, 86]. For example, nested for-loops in L_{NGA} express DFS-like traversals. To avoid the random access and repeated IO, we adopt the *Nested Graph Windows* in [39]. When used with graph streams, it performs sequential scans over them and reduces the repeated IO by enumerating as many walks as possible in memory once a vertex is loaded. It also enables enumeration of walks with a fixed memory for scalable processing of *NGA*.

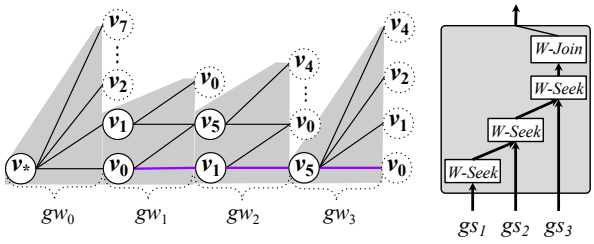
In *Nested Graph Window*, we define a window over a stream S as a subsequence of S with a fixed size sz and denote the window as $\{r_m \in S\}_{sz}$. A graph window (gw) is defined over a graph stream $gs = (vs, es)$ as a pair of a vertex window (vw) over vs and edge window (ew) over es. Conceptually, $gw = (vw, ew)$ represents a subgraph loaded in memory of vertices in vw and edges in ew.

In order to enumerate walks of length k with a fixed amount of memory, we divide the memory space into k areas. Then we use the i-th area to load the graph window gw_i for $i \in [1, k]$. We refer to the tuple of $k+1$ graph windows $(gw_0, ..., gw_k)$ as Nested Graph Windows and denote it as ngw_k. Note that the first graph window gw_0 is defined as a virtual subgraph consisting of a virtual vertex v_* and the edges from v_* to the active vertices V_{active}. Given $ngw_{i-1} = (gw_0, ..., gw_{i-1})$ for $i \ge 1$, gw_i is recursively defined as the subgraph connected with those represented by ngw_{i-1}.

Figure 7 (a) shows the nested graph windows with G_0 in Figure 6. The gw_0 has the virtual vertex v_* and the edges from v_* to $V_{active} = \{v_0, ..., v_7\}$. We assume that all the windows (gw_0 .. gw_3) are of the same size and each can hold two vertices and their edges. Given $ngw_0 = (gw_0)$, by loading v_0 and v_1 into gw_1, we can enumerate walks of length one (v_0, v_1), (v_0, v_5), (v_1, v_0), and (v_1, v_5) omitting v_*. Given $ngw_1 = (gw_0, gw_1)$, by loading v_1 and v_5 into gw_2, we enumerate walks of length two, such as (v_0, v_1, v_5) and (v_0, v_1, v_0). In this way, we recursively define ngw_i ($i \in [1, k]$) over k memory areas to enumerate the walks up to length k with a limited memory.

4.3 Operators

We model graph traversals as enumeration of walks by using a walk generation operator, *Walk* (ω). *Walk* is an n-ary operator ($n \ge 2$)



(a) $ngw_3 = (gw_0, gw_1, gw_2, gw_3)$. **(b) $\omega(gs_1, gs_2, gs_3)$.**

Figure 7: An example of nested graph windows (ngw_3) and *Walk* (ω) for Triangle Counting with G_0.

taking as inputs n graph streams. It has two types of suboperators, *Window Seek* and *Window Join*, as its children. The former, Windows Seek, is for loading the subgraph from a graph stream, and the latter, Window Join, is for graph traversals over the subgraphs in memory represented by ngw while enumerating walks.

Window Seek (W-Seek) loads another subgraph connected to those in memory. That is, given ngw_{i-1} ($i \geq 1$), W-Seek loads gw_i from gs_i. It optionally takes a constraint p_i applied to the data from gs_i. For example, one may put constraints on the vertices, such as the partial order constraint ($u_1 < u_2 < u_3$) in Triangle Counting (Figure 5). Formally, given $ngw_{i-1} = (gw_0, ..., gw_{i-1})$ for $i \geq 1$, W-Seek loads a $gw_i = (vw_i, ew_i)$ from $gs_i = (vs_i, es_i)$ such that $vw_i = \{u_i \in vs_i \mid p_i(u_0, \ldots, u_i)$ where $u_j \in vw_j$ for j in $[0, i-1]$ and $(u_l, u_i) \in ew_l$ for some $l \in [0, i-1]\}_{sz}$ and $ew_i = \{ e \in es_i \mid e.src \in vw_i\}_{sz'}$.

Window Join (W-Join) performs graph traversals over the subgraphs in memory represented by ngw_k. The output is a stream of walks $(u_1, ..., u_{k+1})$ where u_i is a vertex in the i-th graph window ($i \in [1, k]$). The operator optionally takes a constraint p' which filters the walks to enumerate. Formally, given $ngw_k = (gw_0, ..., gw_k)$, W-Join enumerates the walks $(u_1, ..., u_{k+1})$ where $u_i \in vw_i$ for $i \in [1, k]$, $(u_k, u_{k+1}) \in ew_k$, and $p'(u_1, ..., u_{k+1})$ evaluates to true.

Figures 7 shows an example graph traversal for Triangle Counting at G_0 in Figure 6; Figure 7 (a) shows the nested graph windows and the loaded vertices in each window; Figure 7 (b) shows the corresponding *Walk* operator taking three graph streams (gs_1, gs_2, gs_3) for a 3-hop traversal. In Figure 7 (a), a graph window gw_i corresponds to the graph stream gs_i ($i \in [1, 3]$). The gw_0 is defined by V_{active} included in vs_1 of gs_1 as the pre-defined attribute, *active*.

The three W-Seek operators in Figure 7 (b) are performed in the following way. Given ngw_0 with $V_{active} = \{v_0, ..., v_7\}$, we perform the first W-Seek over gs_1 and load v_0 and v_1 into gw_1. Then, we have $ngw_1 = (gw_0, gw_1)$. With ngw_1, we perform the second W-Seek over gs_2 and load v_1 and v_5 into gw_2 as they are connected to v_0 and v_1 in gw_1 and satisfy the constraint $u_1 < u_2$. With ngw_2, we similarly perform the last W-Seek over gs_3 and load v_5 into gw_3 as it is connected to v_1 in gw_2 and satisfy the constraint $u_2 < u_3$. Now we have ngw_3 all loaded, we now perform W-Join operation on ngw_3. It generates four walks (v_0, v_1, v_5, v_0), (v_0, v_1, v_5, v_1), (v_0, v_1, v_5, v_2), and (v_0, v_1, v_5, v_4). With the first one, we find the triangle $<v_0, v_1, v_5>$; the rests do not form a triangle.

We summarize the *GSA* operators in Table 3. Besides Window Seek and Join, we have Filter, Map, Union, and Difference commonly found in [4, 5]. We have two data modification operators

Table 3: Operators in Graph Streaming Algebra.

Input	Output	Name	Symbol
Window, Stream	*Window*	*Window Seek*	*W-Seek*
Window	*Stream*	*Window Join*	*W-Join*
Stream	*Stream*	*Walk*	ω
		Filter	σ
		Map	Π
		Union	$\cup$
		Difference	$\ominus$
		Assign	$\leftarrow$
		Accumulate	$\uplus$
Stream, Operator	*Stream*	*Apply*	α

Assign and Accumulate. They are *stateful* operators with the attribute values or aggregation values as the internal states. Assign ($\leftarrow_{id,attr}$) performs an update to the attribute $attr$ of a vertex with ID id by deleting the old value and inserting the new value. That is, Assign takes as input a stream of tuples each containing the old and new values of the attribute. Then, the operator outputs two tuples for each input tuple, one for deleting the old value and another for inserting the new value. Accumulate ($\uplus_{id,f(attr)}$) aggregates the values of the attribute $attr$ of the input tuples with a key column id using the accumulate function f of an *Abelian monoid*.

Instead of defining an operator for iterative executions such as the fixed-point operator [6], we embed the iterative execution logic of the BSP model within the execution engine. We use the existence of active vertices as the explicit termination condition, which has the same expressive power as a fixed-point iteration [6]. To support nested iterations, we need to define a fixed-point operator similar to [6, 52, 55] and maintain a computation graph of operators.

4.4 Compile L_{NGA} to *GSA*

We explain how to compile an L_{NGA} program Q to *GSA* query plans. We compile each UDF (e.g., Traverse) of Q into a query plan of *GSA*. We perform the compilation at a statement level by translating each L_{NGA} statement to the corresponding *GSA* expression. Then, we merge the expressions into a single *GSA* expression representing the UDF. We denote each plan of Initialize, Traverse, and Update in Q as $\mathbf{P}_{Q^{init}}$, $\mathbf{P}_{Q^{trav}}$, and $\mathbf{P}_{Q^{upd}}$, respectively.

We model the imperative statements of L_{NGA} as the *correlated sub-queries* [61, 62]. A correlated sub-query is a query nested in an outer query and is *parameterized* by each output tuple of the outer query. Table 2 (column 3) shows the translations of the statements to the corresponding sub-query expressions. For example, For is converted to a Filter operation which takes a stream and outputs tuples from it satisfying certain conditions including the condition in *Where*. The statement 'For (v) in ($u.nbrs$)' is parameterized by a vertex u, which takes an edge stream es and outputs the edges $e \in es$ such that ($e.src = u.id$). The rename operator $\rho_{(b_1,\ldots,b_n)/(a_1,\ldots,a_n)}$ is used to rename the column a_i to b_i ($i \in [1, n]$) as declared.

In order to merge the translated expressions and build a query plan of a UDF, we use the Apply operator commonly used for modeling the execution of the correlated sub-queries in relational algebra [21, 25, 62]. We extend it so that it takes as inputs a stream S (instead of a relation) and a *GSA* expression $expr(r)$. It outputs the evaluated results of the expression for each tuple r from S to the output stream. That is, for each $r_m \in S$, it outputs $<r, expr(r)>_m$; $S\ \alpha\ expr = \{<r, expr(r)>_m \mid r_m \in S\}$. The combined expression by the Apply represents the correlated execution of the *GSA* expressions. By exploiting the *query decorrelation* techniques [21, 25, 72], we remove the Apply operators and obtain a single *GSA* expression.

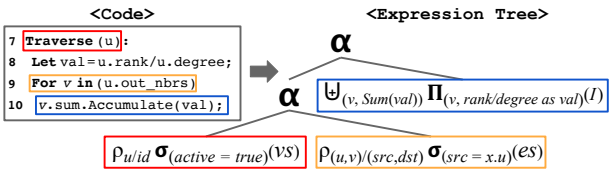


Figure 8: Compiling Traverse of Pagerank into a query plan. I is a relation with an empty tuple and no attributes [72].

We explain the compilation of TRAVERSE in PR, $\mathbf{P}_{pr^{trav}}$. Figure 8 shows its code and expression tree with the APPLY operator (α). At **Line 7**, as TRAVERSE(u) is invoked for each active vertex u in a vertex stream vs (i.e., $active = true$), we have $vs' = \rho_{u/id}(\sigma_{active=true}(vs))$. **Line 8** binds the expression $u.rank/u.degree$ to the local variable val. All followed references to val are replaced with the expression. **Line 9** shows the for-loop which iterates neighbors v of u. This is modeled using the APPLY operator as a correlated expression which selects e in an edge stream es such that $e.src = x.u$ for each x in vs'; $(vs' \; \alpha \; expr(x))$ where $expr(x) = \rho_{(u,v)/(src,dst)}(\sigma_{src=x.u}(es))$. Here, to remove the APPLY operator, we adapt the query decorrelation techniques [21, 25, 72]. As the result, the expression $(vs' \; \alpha \; expr(x))$ is converted, using *WALK* operator, to $\mathbf{P}_{\omega}(vs, es) = \omega_{(vs'.u=es'.u)}(vs', es')$ where $es' = \rho_{(u,v)/(src,dst)}(es)$. **Line 10** accumulates the value of val to sum of each neighbor v. It corresponds to ACCUMULATE ($\uplus$) on sum with the tuples emitted by MAP (Π) for each tuple from the *WALK*. Finally, the whole expression tree is converted to $\uplus_{(v,\, Sum(val))}\, \Pi_{(v,\, rank/degree\ as\ val)}\, \mathbf{P}_{\omega}(vs,\, es)$.

Next, we briefly explain the compilation of $\mathbf{P}_{tc^{trav}}$ for Triangle Counting (Figure 5). **Line 8-10** show the three nested for-loops. For **Line 8**, we have $\sigma_{u1<u2}\,(\rho_{(u1,u2)/(src,dst)}\,(\sigma_{src=x.u1}\,(es_1)))$ where $\sigma_{u1<u2}$ corresponds to the *Where* condition. With the remaining for-loops, the APPLY operator is recursively applied. Then, by decorrelation, we have $\mathbf{P}_{\omega}(vs_1, es_1, es_2, es_3) = \omega_{(u1<u2 \text{ AND } u2<u3 \text{ AND } u4=u1)}\,(\rho_{u1/id}\,(\sigma_{active=true}\,(vs_1)), \rho_{(u1,u2)/(src,dst)}\,(es_1), \rho_{(u2,u3)/(src,dst)}\,(es_2), \rho_{(u3,u4)/(src,dst)}\,(es_3))$. Note that P_{ω} does not have vs_2 and vs_3 as its operands as they are not required to express the query. By merging the expression for ACCUMULATE at **Line 11** with P_{ω}, we have $\mathbf{P}_{tc^{trav}}(vs_1, es_1, es_2, es_3) = P_{\omega} \; \alpha \; \uplus_{Sum(cnts)}\,(\Pi_{1\text{ as cnts}}(S))$. Finally, we have $\mathbf{P}_{tc^{trav}} = \uplus_{Sum(cnts)}\, \Pi_{1\text{ as cnts}}\, \mathbf{P}_{\omega}(vs_1, es_1, es_2, es_3)$

5 INCREMENTAL GRAPH ANALYTICS

By representing an L_{NGA} program in *GSA*, we can automatically derive its incremental query and systemically incorporate several optimizations for its efficient execution. We describe query incrementalization, execution, optimizations, and dynamic graph store.

5.1 Query Incrementalization

As tuples in a stream have positive or negative multiplicities, graph mutations and updates in attribute values can be represented with *delta* streams. Delta stream ΔS of stream S is a stream representing updates to S. For example, updates to vertex attribute values are expressed with two tuples in the delta stream; one for deletion of the previous value and the other for insertion of the new value.

The goal of incremental computations is to efficiently update the previous analytics results without costly re-executions on the entire graph. We define the incremental query ΔQ of query Q as follows: For a *GSA* query Q of N graph streams, $Q(gs_1, ..., gs_N)$, its incremental query ΔQ given the delta graph streams $(\Delta gs_1, ..., \Delta gs_N)$ is defined by the following equation: $Q(gs_1 \cup \Delta gs_1, ..., gs_N \cup \Delta gs_N) = Q(gs_1, ..., gs_N) \cup \Delta Q\,(gs_1, ..., gs_N, \Delta gs_1, ..., \Delta gs_N)$ where $\cup$ is a union operation. Based on the above definition of the incremental query, we can derive *incrementalization rules* for *GSA* operators (Table 4). Because the derivations of the rules are straightforward, we omit their proofs. Note that as we directly incrementalize the Walk operator (ω) rather than the individual W-SEEK and W-JOIN operators, they do not appear in Table 4.

Table 4: *GSA* incrementalization rules; $(s_1, ..., s_n)$ denote n streams; notations of the operators are in Table 3.

① $\Delta(\sigma(s_1)) = \sigma(\Delta s_1)$	② $\Delta(\Pi(s_1)) = \Pi(\Delta s_1)$
③ $\Delta(s_1 \cup s_2) = \Delta s_1 \cup \Delta s_2$	④ $\Delta(s_1 \ominus s_2) = \Delta s_1 \ominus \Delta s_2$
⑤ $\Delta(\leftarrow (s_1)) = \leftarrow (\Delta s_1)$	⑥ $\Delta(\uplus(s_1)) = \uplus(\Delta s_1)$
⑦ $\Delta(\omega(s_1, ..., s_n)) = \omega(\Delta s_1, s_2, ..., s_n) \cup \omega(s'_1, \Delta s_2, ..., s_n)$ $\cup \ldots \cup \omega(s'_1, s'_2, ..., \Delta s_n)$ where $s'_i = s_i \cup \Delta s_i$	

By applying the incrementalization rules to the one-shot query plan P_Q, we obtain its incremental query plan $P_{\Delta Q}$ of *GSA* over (delta) graph streams. For examples, consider the compiled expressions of TRAVERSE in PR and TC explained in Section 4.4. For PR, we apply the Rules ⑥, ②, and ⑦ in Table 4 at $\mathbf{P}_{pr^{trav}}$ to derive $\mathbf{P}_{\Delta pr^{trav}} = \Delta(\mathbf{P}_{pr^{trav}})$. By the Rules ⑥ and ②, we have $\mathbf{P}_{\Delta pr^{trav}} = \uplus_{(v,Sum(val))}\, \Pi_{(v,\, rank/degree\ as\ val)}\, (\Delta P_{\omega})$. By the Rule ⑦, we have $\Delta P_{\omega} = \Delta\,(\omega(vs_1, es_1)) = \omega(\Delta vs_1, es_1) \cup \omega(vs'_1, \Delta es_1)$. For TC, $\mathbf{P}_{\Delta tc^{trav}} = \Delta(\mathbf{P}_{tc^{trav}}) = \uplus_{Sum(cnts)}\, \Pi_{1\text{ as cnts}}\, (\Delta P_{\omega})$ where $\Delta P_{\omega} = \omega(\Delta vs_1, es_1, es_2, es_3) \cup \ldots \cup \omega(vs', es'_1, es'_2, \Delta es_3)$. As shown in Table 4, *GSA* is closed under incrementalization; for any P_Q of *GSA*, its $P_{\Delta Q}$ also consists of *GSA*. Therefore, both P_Q and $P_{\Delta Q}$ can be processed by the same query execution engine supporting *GSA*.

5.2 Incremental *NGA* in *iTURBOGRAPH*

In this section, we explain the execution mechanism of incremental *NGA* under the BSP model. We first introduce notations. We denote by G_t a snapshot of a graph at timestamp $t \in \mathbb{N} \cup \{0\}$. Graph mutation $\Delta G_t = G_t \ominus G_{t-1}$; ΔG_t consists of the tuples for the graph mutation operations occurring at timestamp t (insertion or deletion of edges). We denote a graph stream at timestamp t at superstep s by $gs_{t,s} = (vs_{t,s}, es_t)$ where $vs_{t,s}$ and es_t represent the corresponding vertex stream and edge stream, respectively. We further separately denote the vertex attribute values in $vs_{t,s}$ as $A^{accm}_{t,s}$ for accumulator types and as $A_{t,s}$ for non-accumulator types. For example in PR (Figure 5), the attributes *active* and *rank* are non-accumulator types, and the attribute *sum* is an accumulator type. We denote as $A_{t,s}(v)$ the attribute value of a vertex v.

Recall that for one-shot analytics at G_t under the BSP model, we execute the one-shot query Q at every superstep $s \geq 0$. Q takes as inputs the edge stream (es_t) and attribute values of non-accumulator type ($A_{t,s}$) and outputs the attribute values of non-accumulator type ($A_{t,s+1}$) for the next superstep. This is illustrated in Figure 9. Here, Q consists of the query plans of TRAVERSE and UPDATE. TRAVERSE performs the graph traversals enumerating walks and then updates the attribute values of accumulator types. That is, it takes $(es_t, A_{t,s})$ and outputs attribute values of accumulator types ($A^{accm}_{t,s}$). With $A^{accm}_{t,s}$, UPDATE updates the vertex attribute values and performs vertex activation for the next superstep. That is, it takes ($A_{t,s}$, $A^{accm}_{t,s}$) and outputs $A_{t,s+1}$. Our dynamic graph store maintains both $A^{accm}_{t,s}$ and $A_{t,s}$ for efficient incremental computations.

Now, we explain the execution of the incremental analytics for G_{t+1} with ΔG_{t+1}. The incremental query ΔQ considers the changes in the input of Q and computes the changes in the output of Q. That is, for each superstep s, ΔQ takes the delta streams Δes_{t+1} and $\Delta A_{t+1,s}$ as the additional inputs and outputs $\Delta A_{t+1,s+1}$. This is illustrated at the bottom of Figure 9. Here, ΔQ consists of ΔTRAVERSE and ΔUPDATE which represent the incrementalized TRAVERSE and UPDATE, respectively. For each superstep s, given $\Delta A_{t+1,s}$

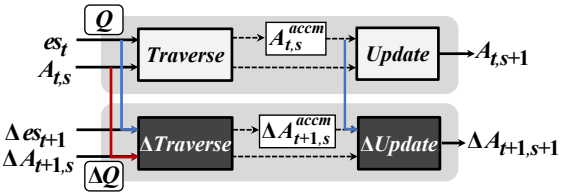


Figure 9: Execution of one-shot query Q (above) at G_t and incremental query ΔQ (below) at G_{t+1} at superstep s.

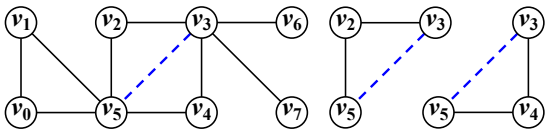
$= (A_{t+1,s} \ominus A_{t,s})$ and Δes_{t+1}, ΔTraverse performs the graph traversals to enumerate the walks consisting of the vertices either with modified attribute values or edge mutations. With the enumerated walks, it incrementally updates the attribute values of accumulator types, which results in $\Delta A^{accm}_{t+1,s} = (A^{accm}_{t+1,s} \ominus A^{accm}_{t,s})$. Finally, ΔUpdate is executed for vertices v with any change in the attribute value; $A_{t+1,s}(v) \neq A_{t,s}(v)$ or $A^{accm}_{t+1,s}(v) \neq A^{accm}_{t,s}(v)$. It results in $\Delta A_{t+1,s+1} = (A_{t+1,s+1} \ominus A_{t,s+1})$. We add $\Delta A_{t+1,s+1}$ and $\Delta A^{accm}_{t+1,s}$ to the dynamic graph store.

5.3 Δ-walk Enumeration

As walk enumeration for ΔTraverse has a large impact on performance, we elaborate its execution and optimization. Processing of ΔTraverse needs to enumerate new or invalidated walks containing the vertices with modified attribute values or edge mutations. According to Rule ⑦ in Table 4, the incremental *Walk* with k streams is a union of k sub-queries ($\cup^k_{n=1}\ q_n$) of *Walk*s, each taking $k-1$ streams and one delta stream. Here, we build upon the well-understood existing work of delta query decomposition [7, 10, 29, 37]. We apply the technique for the incremental *NGA*, particularly for the enumeration of Δ-*walk*s, which we call Δ-walk Enumeration. At G_{t+1} and superstep s, a Δ-walk is a walk (u_1, ..., u_{k+1}) where, for some $i \in [1, k]$, there exists a vertex u_i such that $A_{t+1,s}(u_i) \neq A_{t,s}(u_i)$ or an edge $e \in \Delta G_{t+1}$ such that $e.src = u_i$. Note that the multiplicity of a Δ-walk is computed as 1 or -1 by the product of the multiplicities of the joined tuples. We denote as V_Δ the set of the starting vertices (u_1) of these Δ-walks.

For example, consider Triangle Counting (TC) query in Figure 5. Figure 10 (a) shows the graph G_1 with an insertion of an edge $e = (v_3, v_5)$, and Figure 10 (b) shows the newly found triangles after ΔG_1. Recall that we performed the graph traversals from all active vertices for one-shot analytics of TC. However, for incremental TC, the fixed traversal order used for the one-shot analytics can be sub-optimal, and the traversals starting from any vertex other than v_2 or v_3 are wasteful as they find no triangle. To this end, we propose *traversal reordering* and *neighbor pruning*.

We first explain *traversal reordering* as a basic join order optimization [7, 37, 46] which we apply for incremental *NGA*. For $\mathbf{P}_{\Delta tc^{trav}}$, consider a sub-query $q_4 = \omega(vs', es'_1, es'_2, \Delta es_3)$ which finds the triangle $<v_3, v_4, v_5>$. Figure 11 (a) shows the query plan of q_4 using the original traversal order ($u_1 \rightarrow u_2 \rightarrow u_3$). Until the W-Seek over Δes_3, it performs total re-execution of W-Seeks with (vs', es'_1, es'_2) for all active vertices, which is wasteful. Similar to the join reordering in relational databases, we solve this problem by reordering the graph traversals. That is, we reorder the graph traversal so that we perform W-Seek for the delta vertex or edge stream as early in the plan as possible. Figure 11 (b) shows its reordered plan with a new traversal order, ($u_1 \rightarrow u_3 \rightarrow u_2$). With the



(a) Graph G_1 with $\Delta G_1 = \{(v_3, v_5)\}$. (b) New triangles in G_1.

Figure 10: A running example of incremental graph analytics for Triangle Counting (TC) query.

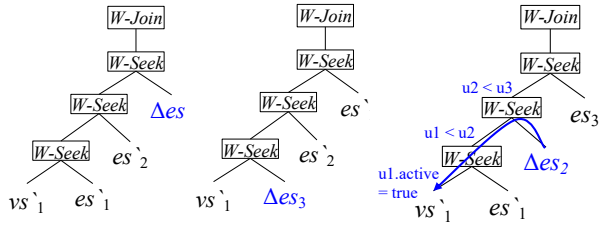
reordered plan, by traversing first from v_3 only to v_5 via $e \in \Delta es_3$ and then to v_4 ($< v_5$ and $> v_3$), we find the new triangle while avoiding unnecessary graph traversals. Note that we fix the starting vertex to benefit from the user heuristics and selection conditions (e.g., *active*). This is also beneficial system-wide; the memory space for the data of the starting vertices can be shared by all sub-queries.

Next, we explain *neighbor pruning*. For $\mathbf{P}_{\Delta tc^{trav}}$, consider a sub-query $q_3 = \omega(vs', es'_1, \Delta es_2, es_3)$ which finds the new triangle $<v_2, v_3, v_5>$. Observe that the new triangle is found by starting the traversal only from v_2 which is a one-hop neighbor of v_3 which gets directly affected by ΔG_1. That is, by restricting the starting vertices for Δ-walk enumeration to be neighboring vertices of the vertices with either modified attribute values or edge connections, we can prune out unnecessary graph traversals.

For neighbor pruning in the example for q_3, we perform the Multi-Source BFS (MS-BFS) from the vertices affected by ΔG_1. Figure 11 (c) shows the plan of q_3 with the selection conditions exploited during the MS-BFS. We start the MS-BFS from the vertices $\{u_3 \mid <u_2, u_3>_m \in \Delta es\} = \{v_3, v_5\}$. At Depth 1, we backward traverse from v_3 and v_5 via e to each other. We follow only from v_5 to v_3 exploiting the selection condition, u_2 ($= v_3$) $< u_3$ ($= v_5$). At Depth 2, we traverse from v_3 to v_2 ($< v_3$) in the same way and find v_2 as a candidate for V_Δ of q_3. When processing q_3, we enumerate the walks consisting of only the vertices visited during the MS-BFS and find the new triangle $<v_2, v_3, v_5>$.

We explain the detailed process of the MS-BFS for neighbor pruning. Considering a sub-query with Δes as one of its operands, we perform backward MS-BFS up to at most Depth k starting from $X^0 = \{e.dst \mid e_m \in \Delta es\}$. At depth $i \geq 1$, we perform backward traversal from X^{i-1} and visit $X^i = \{e.src \mid e.dst \in X^{i-1} \wedge e \in es^i \wedge Pred_{e_i}(e) \wedge Pred_{v_i}(e.src)\}$. Here, es^i is Δes if $i = 0$, and es' otherwise. Also, $Pred_{e_i}$ and $Pred_{v_i}$ represent the selection condition on the corresponding edges and vertices, respectively. For a sub-query with Δvs, we perform the MS-BFS up to at most Depth $(k-1)$ starting from $\{v \mid <v, value>_m \in \Delta vs\}$. The rest is similar. Then, when we enumerate Δ-walks, we follow only the vertices visited during the MS-BFS. The MS-BFS programs are generated by the compiler.

Lastly, we propose a technique called *seek/window sharing*. While we have four sub-queries of *Walk* for $\mathbf{P}_{\Delta tc^{trav}}$, processing them individually may lead to poor performance and a low degree of parallelism due to repeated IOs. By exploiting the batch-processing technique, we achieve a sharing of the IO and the memory spaces for the graph windows. Suppose that we are given k sub-queries of *Walk*s, $\{q_n\}^k_{n=1}$. We merge the IO requests by the i-th W-Seek ($1 \leq i \leq k$) of all sub-queries and annotate each IO request with the relevant sub-query IDs. Then, when the IO is served, we compute all of the annotated sub-queries. In this way, we avoid repeatedly loading the same data and increase the CPU utilization.



(a) q_4. (b) reordered q_4. (c) q_3.

Figure 11: Examples of traversal reordering for $q_4 = \omega(vs'_1, es'_1, \Delta es_2, es_3)$ and neighbor pruning for $q_3 = \omega(vs'_1, es_1, \Delta es_2, es_3)$ at TC. The schema for tuples from each stream are: vs_1 = <u1, active>, es_1 = <u1, u2>, es_2 = <u2, u3>, es_3 = <u3, u4>.

5.4 Incremental Accumulate

With the enumerated Δ-walks, we need to incrementally update the accumulators used in Traverse. For example, consider an accumulator of a vertex where the accumulation function is f. Then its incremental query plan in *GSA* is expressed as $\uplus(\Pi(\Delta P_\omega))$ where $\Pi(\Delta P_\omega)$ emits a tuple $<v, val>_m$ for the new (m=1) or deleted (m=-1) Δ-walk from ΔP_ω. We separately denote new and deleted Δ-walks generated by ΔP_ω as $\Delta^+ P_\omega$ and $\Delta^- P_\omega$, respectively. Then we have $\uplus_{v,f(val)}(\Pi_{v,val}(\Delta P_\omega)) = \uplus_{v,f(val)}(\Pi_{v,val}(\Delta^+ P_\omega \ominus \Delta^- P_\omega))$.

For efficient incremental Accumulate, we exploit the algebraic properties of the accumulation functions which form either the *Abelian group* or *Abelian monoid*. The Abelian group has a commutative and associative addition operation $f: M \times M \rightarrow M$ for a domain M with an identity element $\mathbf{1} \in M$ such that $f(x, \mathbf{1}) = f(\mathbf{1}, x) = x$ for any $x \in M$. Examples are Sum and Product. Since it has an inverse $g: M \rightarrow M$ such that $f(x, g(x)) = f(g(x), x) = \mathbf{1}$ for any $x \in M$, it can easily be incrementally maintained under deletions; the accumulation of x can be offset by the accumulation of $g(x)$. However, Abelian monoids such as Min or Max does not have the inverse. Therefore, under deletions, they require recomputations.

When f belongs to an Abelian group, we do not perform recomputations (e.g., Sum for Pagerank and Triangle Counting) even under deletions by rewriting the incremental query plan using the inverse of f. We rewrite the plan $\uplus_{v,f(val)}(\Pi_{v,val}(\Delta P_\omega)) = \uplus_{v,f(val)}(\Pi_{v,val}(\Delta^+ P_\omega \ominus \Delta^- P_\omega))$ as $\uplus_{v,f(val)}(\Pi_{v,val}(\Delta^+ P_\omega) \cup \Pi_{v,g(val)}(\Delta^- P_\omega))$ where we take the inverse g on val for deletions.

For PageRank, the f is Sum and its inverse is $g(x) = -x$ for all x. The insertion or deletion of an edge (u, v) leads to the insertion of a tuple (v, val) or $(v, -val)$ to be aggregated by f. For the insertion, the value of val is added to the previous aggregation result. For the deletion, the value of val is subtracted by adding its inverse. The update in the value of val is processed by subtracting its old value and adding its new value into the previous aggregation result.

When f belongs to Abelian monoid and $\Delta^- P_\omega = \emptyset$, we have $\uplus_{v,f(val)}(\Pi_{v,val}(\Delta^+ P_\omega))$. In such a case, it is processed in the same way as that of the Abelian group. When $\Delta^- P_\omega \neq \emptyset$, we fall back to recomputation of Accumulate for the vertices (V_{aff}) affected by the Δ-walks in $\Delta^- P_\omega$; $V_{aff} = \{ v \mid (v, val) \in \Pi_{v,val}(\Delta^- P_\omega)\}$. To find the candidate starting vertices V_{re} for recomputation, we perform a backward MS-BFS from V_{aff}. Then, we perform *Walk* with $(V_{re} \wedge V_{active})$ as the starting vertices and Accumulate to recompute the accumulator attribute values of V_{aff}.

Since performing recomputation for every deletion can be costly, we take further advantage of the algebraic properties of individual accumulators to avoid recomputations when possible. For example with Min, if a deleted value is larger than the accumulated value, it requires no recomputation. Moreover, by maintaining the number of supportive tuples for the accumulated value, we can avoid redundant recomputations [28]. Consider the example of Min({1, 2, 5, 1}) = 1. If we maintain the number of supportive tuples, 2, for the minimum value, 1, we can avoid the unnecessary recomputation. Because we express the query with formally defined *GSA*, we can incorporate more advanced and systemic techniques, such as query rewriting based on program analysis [16, 17] and maintenance of various aggregations [57, 79], which are beyond our scope.

5.5 Dynamic Graph Store

Our dynamic graph store consists of the vertex store for vertex attribute values and the edge store for edge connections. While we need to support updates for vertex attribute values and for edge connections, in-place updates of the graph on disk are costly; the *IO amplification* problem [54] and reorganization due to page under-/overflow. To avoid such costly in-place updates, we propose a *delta*-based maintenance strategy. Due to the page constraint, we focus on explaining the maintenance of non-accumulator attributes ($A_{t,s}$). Accumulator attributes are maintained similarly.

The vertex store maintains the changes in vertex attribute values as deltas. During the one-shot analytics with G_0, the vertex attribute values change across supersteps. For each superstep $s > 0$, we store only the changes against the previous superstep, $A_{0,s} \ominus A_{0,s-1}$. Specifically, we store the *after-image* $A_{0,s}(v)$ of vertices v such that $A_{0,s}(v) \neq A_{0,s-1}(v)$. During the incremental analytics with G_t ($t > 0$), the vertex attribute values change across both snapshots and supersteps. That is, we store the after-images of vertices v such that $A_{t,s}(v) \neq A_{t,s-1}(v)$ or $A_{t,s}(v) \neq A_{t-1,s}(v)$. Over the multiple snapshots, we store the deltas of the same superstep in a single file.

The vertex store needs to support efficient retrievals of vertex attribute values. For G_t, after the execution engine computes the superstep s with $A_{t,s}$ materialized as an array in memory, it issues the retrieval of $A_{t,s+1}$. Then, we load the deltas for the superstep $s + 1$ from the disks and update the values in the array.

We briefly explain our delta maintenance strategy which decides when to merge deltas based on our cost model. While maintaining the changes as deltas reduce the amount of disk writes, the deltas need to be read repeatedly to execute incremental queries. We incorporate this trade-off into our cost model to determine when to merge the deltas. For every superstep s with G_t, we compare the write cost of merging the deltas, $W^{(t,s)}_{merge}$, and the read cost of the deltas, $R^{(t,s)}_{delta}$. Let $X^{(t,s)}$ be a set of vertices whose after-image values need to be stored as deltas; $X^{(t,s)} = \{ v \mid (A_{t,s}(v) \neq A_{t,s-1}(v)) \vee (A_{t,s}(v) \neq A_{t-1,s}(v))\}$. We have $W^{(t,s)}_{merge} = |\cup_{\tau \leq t} X^{(\tau,s)}|$ and $R^{(t,s)}_{delta} = \Sigma_{0<\tau<t}(t - \tau) \times |X^{(\tau,s)}|$. Here, the read cost is computed considering the repeated reads of the deltas from their creation until now. We merge the deltas if $W^{(t,s)}_{merge} < R^{(t,s)}_{delta}$.

The edge store maintains G_0 and ΔG_t ($t > 0$) separately for each t. The insertion and deletion operations in ΔG_t are maintained in separate files, so that the execution engine is aware of the multiplicity of edge tuples. Each file stores the edges in the CSR-like

format and thus, the execution engine can access the initial graph and graph mutations identically. An issue is how to handle the deletions. That is, when the execution engine scans the edge streams of G_t, the edges deleted by ΔG_τ ($\tau \leq t$) should not be visible. However, eagerly applying the deletions to the data on disk is expensive. Our simple strategy is to maintain the deletions in ΔG_t in memory and to lazily mark the corresponding edges as deleted when their associated disk pages are loaded into the page buffer pool.

6 EXPERIMENTS

We present the evaluation of *iTurboGraph* (*iTbgpp*). We first evaluate the overall performance of *iTbgpp* against the state-of-the-art systems over real-world graphs to demonstrate the effectiveness of the incremental computations (Section 6.2). We observe some limitations of the competitor systems; they perform many redundant computations or do not scale due to enormous intermediate results. Then, we perform an in-depth study of the scalability of *iTbgpp* with varying graph size and the number of machines (Section 6.3). We observe the limited scalability of the competitor system which runs out of memory for small graphs due to the memory overhead of storing intermediate results. Lastly, we conduct a sensitivity analysis of *iTbgpp* (Section 6.4). Here, we study the performance characteristics of *iTbgpp* for various analysis algorithms by varying the workloads of graph mutations: the ratios of insertions to deletions and the batch size of graph mutations. We also demonstrate the effectiveness of several optimizations.

6.1 Experimental Setup

Competitors: We compare *iTbgpp* against GraphBolt (GrB) [51] and Differential Dataflow (DD) [52]. GrB extends the vertex-centric computation model and provides users with APIs to manually implement incremental graph algorithms. DD is based on the relational model and provides automatic incrementalization of user programs. DD supports general dataflow programs for various workloads, such as relational queries and datalog programs [64]. In contrast, *iTbgpp* is focused on *NGA* which is the main target of our paper. GrB and DD are in-memory systems over a single-machine and distributed environment, respectively. We do not compare with Kickstarter [77] since it supports some *monotonic* algorithms only.

Analysis Algorithms: We use six algorithms and group them based on their time complexities and performance characteristics. Group 1 consists of representative matrix-vector multiplication algorithms [32, 91]: PageRank (PR) and Label Propagation (LP). Group 2 includes representative algorithms for graph connectivity problems [82]: Weakly Connected Components (WCC) and Breadth-First Search (BFS). Group 1 and Group 2 are one-hop *NGA*. Group 3 consists of multi-hop *NGA* [39, 60]: Triangle Counting (TC) and Local Clustering Coefficient (LCC). We run Group 1 algorithms for 10 iterations and Group 2 algorithms until convergence. For BFS, we use the vertex with the largest degree as the root. For Group 1 algorithms, since DD does not support floating-point data types for attributes, we implement PR and LP using integer types in all evaluated systems by scaling the floating numbers to decimal numbers. We scale them by 1,000, which is equivalent to rounding the floating numbers down to three decimal places.

For GrB, we use the authors' implementations for incremental PR and LP with the modifications using the integer data types as explained above. We do not compare with GrB for Group 2 or Group 3 algorithms because their implementations are not open-sourced. The descriptions of their implementations in [51] were not comprehensive enough to implement the *incremental* algorithms.

For DD, we implement the Group 1 and 2 algorithms as described in [36] with all the optimizations described in the paper. For TC, we use the fastest implementation among the ones provided after performance comparison. We implement LCC.

Table 5: The statistics of datasets.

Dataset	|V|	|E|	Size
Twitter (TWT)	41.6 M	1.37 B	21.9 GB
GSH15 (GSH15)	988 M	33.9 B	542.4 GB
Clueweb12 (CW12)	6.3 B	66.8 B	1.06 TB
HyperLink (HL)	3.3 B	119 B	1.9 TB
RMAT_X ($25 \leq X \leq 36$)	2^{X-4}	2^X	512 MB ~ 1 TB

Datasets: We use four real-world graphs [1, 2, 42, 53] and synthetic graphs of various sizes (Table 5) for the evaluation. For extensive evaluation with large social graphs, we upscale the Twitter social graph (TWT) using a graph upscaling method, EvoGraph [59]. We denote its X times upscaled graph by TWT_X. Synthetic graphs are generated according to the RMAT model [13] using TrillionG [58].

Workloads: We generate the workloads of graph mutations in a similar way as [51]. Given a graph data, we sample 90% of the edges uniformly at random and use them as the initial graph G_0. For the insertion workloads Δ^+G_i ($i > 0$), we use the remaining 10% of the edges. For the deletion workloads Δ^-G_i ($i > 0$), we sample the edges in G_{i-1} uniformly at random. By default, we use $|\Delta^+G_i| : |\Delta^-G_i| = 75 : 25$ [9] and $|\Delta G_i| = 100$k. We report the execution times of one-shot query for G_0 and the average execution times of four consecutive incremental queries for G_i where $i \in [1, 4]$.

Running Environment: We use a cluster of 25 machines, interconnected by an InfiniBand QDR 4x network switch. Each machine has two Intel Xeon E5-2450 CPUs (each with 8 cores), 64GB RAM (32GB per a NUMA node), and a PCIe SSD.

6.2 Overall Performance

We begin by comparing the overall performance of the evaluated systems with the real-world graphs to demonstrate their effectiveness for incremental graph analytics. We compare *iTbgpp* against GraphBolt (GrB) on a single machine and Differential Dataflow (DD) on a cluster of 25 machines (Table 6 and Figure 12).

6.2.1 Single-Machine Evaluation. We first present our single-machine evaluation for *iTbgpp* and GrB. Table 6 shows the execution times at TWT. We omit the results with other graphs as GrB's scalability is restricted by the memory capacity of a single machine.

For one-shot queries, *iTbgpp* performs comparably to GrB even with disk IO by efficiently overlapping computation and IO.

For incremental queries of PR and LP, *iTbgpp* outperforms GrB, as it eliminates a larger amount of unnecessary computations. GrB performs unnecessary refinement computations as it considers only the transitive impacts of graph mutations along with the neighbor relationship starting from the inserted or deleted edges. However,

Table 6: Execution times at TWT on a single machine.

Time [sec]	PR		LP	
	One-shot	Incremental	One-shot	Incremental
GrB	59.9	54.5	133.5	109.5
iTbgpp	53.2	23.8	139.6	29.8

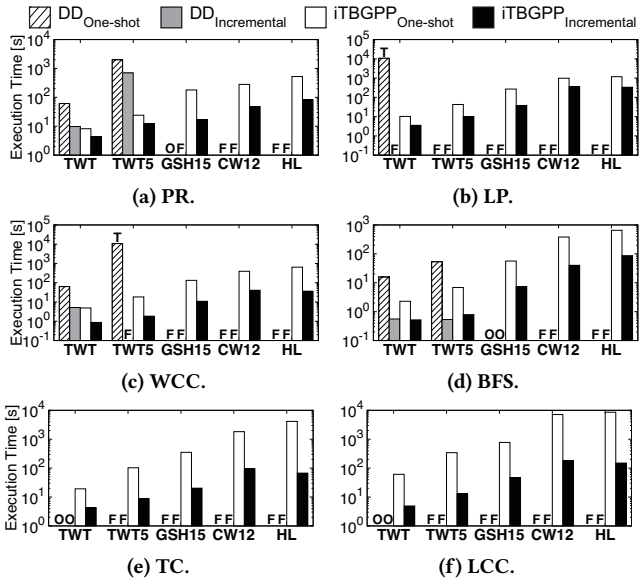


Figure 12: Execution times at real graphs on 25 machines.

iTBGPP additionally considers the actual changes of the vertex attribute values and does not perform computations if the value does not change from the previous snapshot.

While GrB maintains large arrays of vertex attributes for all supersteps in memory, *iTBGPP* stores only the changes in the attribute values as deltas on the secondary storage under memory pressure. That is, we trade off the memory overhead against the disk IOs. To prevent the deltas from growing indefinitely with high IO cost, we use our cost-based management strategy evaluated in Section 6.4.2.

6.2.2 Distributed Evaluation. We evaluate *iTBGPP* and DD on a cluster of 25 machines (Figure 12). DD is an in-memory system and maintains all intermediate results for incremental computations. We mark O, T, or F if the system fails: Out-Of-Memory (OOM), time-out (3 hours), or for other reasons, respectively.

DD's strategy of maintaining the intermediate results might be reasonable for some workloads with small intermediate result sizes. However, we observe that DD suffers from serious scalability problems for *NGA* where the joins can generate enormous intermediate results. For example, TC performs self-joins on the edge table for graph traversals, which leads to $O(|E|^2)$ intermediate results size. *iTBGPP* achieves scalable processing of incremental *NGA* by avoiding maintaining the intermediate results for such joins.

Group 1: Figures 12 (a) and (b) show the execution times for PR and LP, respectively. While *iTBGPP* scales to the largest graph, DD crashes due to the OOM error at a much smaller graph.

For one-shot queries, DD shows slow execution times due to its high network IO cost for repartitioning the messages. At every superstep, DD processes the Join with Vertex and Edge tables and generates messages. To process Reduce with the messages for value aggregation, it repartitions the messages. Specifically for PR at TWT, the network transfer size of DD is 16.3 times larger than that of *iTBGPP*. *iTBGPP* reduces the network IO cost by performing partial pre-aggregation [39, 44] exploiting the algebraic properties of the accumulations in *GSA*. DD incurs disk swaps on TWT_5 due to its large memory usage for maintaining the intermediate results. For PR at TWT_5, it consumes memory of 2.1 TB in total over 25 machines, which is 24.4 times larger than the input graph size.

iTBGPP efficiently processes both the one-shot and incremental queries. The speedups by the incremental computations are on average 5.3 for PR and 4.1 for LP. The incremental computations accelerate the execution time with reduced disk IO. For example, at HL, compared to the one-shot query, the disk IO bytes for the incremental query are reduced to 16% for PR and 56% for LP.

Group 2: Figures 12 (c) and (d) show the execution times for WCC and BFS, respectively. Overall trends are similar to those for Group 1. DD shows limited efficiency and scalability. *iTBGPP* efficiently processes up to the largest graph without crashing. In *iTBGPP*, the speedups by the incremental computations are on average 11.3 for WCC and 10.2 for BFS. The speedups increase from 6.2 and 4.7 at TWT to 17.6 and 7.5 at HL for WCC and BFS, respectively.

For BFS at TWT_5, *iTBGPP* is slightly slower than DD by 0.12 seconds. This is due to the different design choices of the two systems. DD maintains the previously generated messages sorted as inputs to the Min aggregation, requiring 1.4 TB heap memory space which is 17.4 times larger than the input graph size. When the messages for the current minimum value are deleted, and the minimum value changes, the aggregation value is updated to the next larger one. *iTBGPP* needs to recompute the Min aggregation by generating the input messages in such cases, which takes 0.33 seconds at TWT_5. However, *iTBGPP*'s design choice pays off for scaling to the larger graphs, while DD fails due to the OOM error.

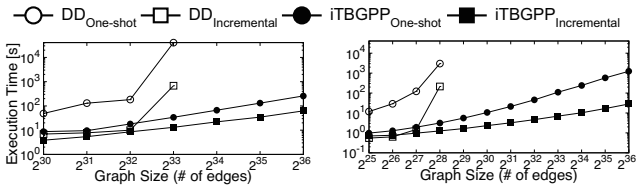
Group 3: Figures 12 (e) and (f) show the execution times for TC and LCC, respectively. DD cannot process the smallest TWT, incurring the OOM error. While DD maintains the intermediate results for faster incremental computations, the total size can reach up to $\Sigma_{v \in V} deg(v)^2$ where $deg(v)$ is the degree of a vertex v. Specifically, the square sum for TWT and HL are as large as 199 trillion and 36 quadrillion, respectively, which indicates that maintaining the entire intermediate results is not suited for *NGA*. *iTBGPP* processes up to the largest graph without crash.

The incremental computations reduces the required computations and IO. For example, at HL, compared to the one-shot query, the disk IO bytes for the incremental query are reduced by a factor of 75.5 for TC and 54 for LCC. The total CPU time measured by *clock_gettime()* is reduced by a factor of 94 for TC and 63.8 for LCC.

6.3 Scalability

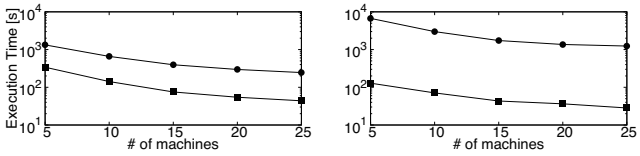
We evaluate the scalability of the systems for processing large graphs with limited hardware resources. We first vary the size of RMAT graphs with a cluster of 25 machines. We then vary the number of machines with a fixed-size graph. We use PR for Group 1 and TC for Group 3 and omit the results for Group 2 as it shows similar trends. For TC, we begin with a small graph $RMAT_{25}$.

6.3.1 Varying Graph Size. Figure 13 shows the execution times of PR and TC varying the sizes of *RMAT* graphs. Overall, we confirm our previous observations from Section 6.2. *iTBGPP* scales up to the largest graph efficiently without crashing. DD shows scalability problems by maintaining all intermediate results in memory. For PR and TC, DD suffers from the disk swaps with $RMAT_{33}$ and $RMAT_{28}$ where it requires 2.9 TB and 4.9 TB heap memory space which are 26 and 1349 times larger than the input graph sizes, respectively. It crashes due to the OOM error with $RMAT_{34}$ and $RMAT_{29}$.



(a) PR. (b) TC.

Figure 13: Execution times varying graph size.



(a) PR. (b) TC.

Figure 14: Execution times varying the number of machines.

The benefits of incremental computations increase as the graph size grows, which shows the needs for scalable processing of incremental computations for large-scale dynamic graphs. For PR and TC, the speedups by the incremental computations in *iTBGPP* are on average 2.8 and 12.5, respectively. For the largest graph $RMAT_{36}$, the speedups increase up to 4.1 and 43.9, respectively.

6.3.2 Varying the Number of Machines. Now, we show the scalability of *iTBGPP* for varying the number of machines while fixing the graph to $RMAT_{36}$. We omit the results of DD because it crashes due to the OOM error for all cases.

For PR, as the number of machines increases from five to 25, *iTBGPP* achieves 5.4 and 7.7 speedups for the one-shot and incremental queries, respectively. The super-linear speedup of the incremental query is because the incremental query accesses only some portions of the graph, and their data get fit in the memory of the cluster machines as more machines are added, reducing disk IO.

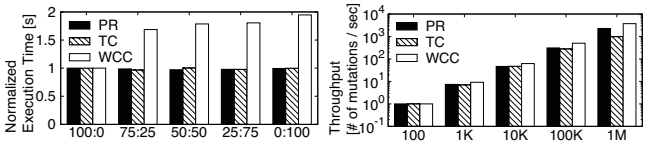
For TC, *iTBGPP* achieves 5.4 and 4.5 speedups for the one-shot and incremental queries, respectively. *iTBGPP* efficiently processes the incremental *NGA* even with a small number of machines without maintaining the intermediate results.

6.4 Sensitivity Analysis

We conduct the sensitivity analysis of the performance of *iTBGPP* with TWT_{25}. We study the performance characteristics of different algorithms by varying the workloads of graph mutations: 1) the ratios of the number of insertions to the number of deletions; 2) the number of graph mutations per snapshot. We also study the effects of the optimizations in *iTBGPP*.

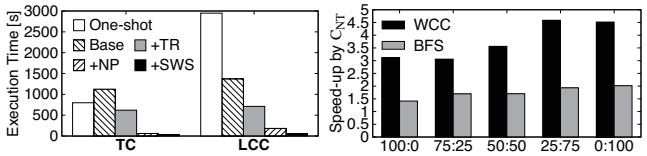
6.4.1 Varying Workloads. For a query, if the aggregate function belongs to the Abelian group, no recomputation is needed even under deletions. However, if the aggregate function belongs to the Abelian monoid, recomputations might be needed under deletions.

We vary the ratios of the number of insertions to the number of deletions, $|\Delta^+G| : |\Delta^-G|$ = {100:0, 75:25, 50:50, 25:75, 0:100}. Figure 15 (a) shows the execution times normalized by the execution time of the insertion-only workload. For PR and TC, no recomputation is needed since the aggregate function belongs to an Abelian group. Therefore, PR and TC show no differences for varying ratios. However, for WCC, it uses the MIN accumulator belonging to the



(a) Varying $|\Delta^+G| : |\Delta^-G|$. (b) Varying $|\Delta G|$.

Figure 15: Normalized execution times and throughputs varying the workloads.



(a) TC and LCC. (b) WCC and BFS.

Figure 16: Execution times of Group 3 and Speed-ups for Group 2 varying the optimizations.

Abelian monoid which may require recomputations under deletions. Therefore, as the amounts of deletions increase, the execution times increase as well. Similar results are observed in [51].

We evaluate the effects of batch size on *iTBGPP* in the two aspects: (1) the scalable processing with a large batch size without the out-of-memory problem and (2) the increased throughput for large batch sizes by sharing computations and IO within the batch.

We vary the number of graph mutations in each ΔG, $|\Delta G|$ = {100, 1k, 10k, 100k, 1M}. Running incremental computation for every graph mutation can be wasteful. By processing multiple graph mutations in a batch, *iTBGPP* can greatly increase the throughput of refreshing the analytics results. Figure 15 (b) shows the throughput normalized by the throughput with $|\Delta G|$ = 100. The throughput (# of mutations per second) is $|\Delta G|$ divided by the execution time. As the batch size increases from 100 to 1M, the throughput increases 43894, 26782, and 101303 times for PR, WCC, and TC, respectively.

6.4.2 Effects of Optimizations. We evaluate the effects of the optimizations in *iTBGPP*. The selected optimizations are traversal reordering (TR), neighbor pruning (NP), seek/window sharing (SWS), MIN with counting (CNT), and cost-based delta maintenance (COST).

First, we evaluate the effects of the optimizations (TR, NP, and SWS) for TC and LCC as examples of multi-hop *NGA*. Figure 16 (a) shows the execution times for varying the optimizations. We denote by BASE the method with all of the optimizations disabled.

For TC, although BASE performs incremental computations, it takes longer than the one-shot query. At BASE, the IO is bottlenecked due to the redundant and repeated IO for the four subqueries. While TR reduces the network IO by 49%, compared to BASE, it is only 29% faster than the one-shot query. By applying both TR and NP, we can avoid many of the redundant IO, and achieve 13.1 times speedup over the one-shot query. Finally, by applying SWS, which batch-processes the sub-queries by sharing their IO, *iTBGPP* achieves 28.9 times speedup. TR and NP effectively reduce the redundant computation and IO cost, and SWS further reduces the IO cost and increases the degree of parallelism by sharing the IO.

For LCC, even BASE is 2.2 times faster than the one-shot query. This is due to the significantly high CPU computation cost of LCC for the one-shot query. In the incremental query, the bottleneck is

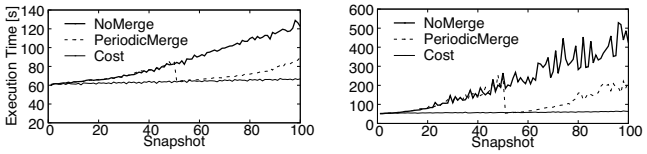


Figure 17: Execution times of the incremental queries for PR (left) and LP (right) varying the optimization.

changed to the IO, similar to TC. By applying all the optimizations, LCC achieves 52.7 times speedup.

Now, we evaluate the effect of the optimization (Cnt) for incremental accumulate for Min. It prunes out the redundant recomputations under deletions. We vary the workloads ratios for WCC and BFS. Figure 16 (b) shows the speedups achieved when applying the Cnt optimization while all other optimizations are enabled. At the insertion-only workload (100:0), the speedup is 2.4 for WCC and 1.4 for BFS. Note that even at the insertion-only workload, recomputations can occur. For example, suppose that a vertex v has converged at superstep s of G_t. After some insertions by ΔG_{t+1}, v can converge faster at superstep s' ($< s$) of G_{t+1}. Then, the previously sent updates from v to its neighbors at supersteps s'' ($s' < s'' \leq s$) need to be deleted. As the workloads include more deletions, the speedups increase up to 3.2 and 2 for WCC and BFS, respectively.

Finally, we evaluate the effects of the technique (Cost) for PR and LP. We test it against two other approaches: 1) NoMerge which does not merge the deltas, and 2) PeriodicMerge which merges the deltas periodically. We use 50 snapshots as the period. Figure 17 shows the execution times of the three approaches up to 100 snapshots with $|\Delta G_i| = 1$M for $i \in [1, 100]$. The execution time of NoMerge increases rapidly as the deltas keep accumulating. PeriodicMerge exhibits suboptimal performance. Until the 50-th snapshot, its execution time increases along with NoMerge. Our proposed cost-based strategy (Cost) achieves consistent execution times over many snapshots.

7 RELATED WORK

Graph Analytics Language. The vertex-centric programming model [50] is the most widely used one with various applications [18, 82]. In addition, several domain-specific languages for graph analytics have been proposed [15, 24, 34, 67, 85] being focused on parallelizing their processing. Green-Marl [34] focuses on efficient parallelization in shared-memory environments. GRAPE [24] aims at automatic parallelization of a provided sequential algorithm. In comparison, our proposed language, L_{NGA}, adopts the vertex-centric programming model for compatibility and extends it for simple and intuitive programming of *NGA* with nested for-loops.

Incremental Graph Analytics. Differential Dataflow [52] is built for automatic incrementalization on top of Naiad [55]. Naiad provides a powerful dataflow programming functionality with nested iterations allowing nested cycles in its computation graph, which can be used to compose multiple iterative programs. However, the scalability for incremental computations is limited due to the high space complexity of maintaining the enormous intermediate results. Kickstarter [77] exploits algorithmic insights for monotonic algorithms. GraphBolt [51] refines the previous computation results by exploiting a dependence graph. However, none of them provides automatic incrementalization nor supports for *NGA*. GraphFlow [37] has shown the effectiveness of the delta query decomposition technique of [10] for incremental pattern matching problems, and [7] has extended it to the worst-case optimality. While they focus on efficiency by keeping the data and indices in memory, *iTurboGraph* targets to scale to large graphs with a fixed memory using secondary storage devices. [23] studies the theoretical bounds of incremental graph computation problems. In comparison, *iTurboGraph* is a full system focusing on the scalable processing of the incremental *NGA* with a new incremental, distributed graph analytics engine, which avoids maintaining the enormous intermediate results. In addition, it automatically incrementalizes queries written in L_{NGA}.

Streaming Graph Processing. Many disk-based graph analytics systems have exploited streaming processing techniques [14, 39, 48, 63, 83, 87, 90]. However, they focus on the one-shot analytics over the static graphs and provide no algebraic foundations for their streaming graph processing. We propose Graph Streaming Algebra (*GSA*) as a theoretical foundation for scalable and efficient processing of *NGA* and automatic incrementalization over dynamic graphs. CQL [8] extends the relational algebra for stream processing. However, it does not consider nesting windows over streams, which *iTurboGraph* supports for scalable processing of incremental *NGA*.

Dynamic Graph Store. LiveGraph [89] is a transactional graph store with multi-version concurrency control but is not optimized for analytics workloads. For example, accessing vertex attribute values can require traversals over version chains. GraphOne [41] presents a hybrid graph store which efficiently ingests streaming graph mutation updates. However, it supports the graph mutations only but it does not handle the updates in the vertex attributes values necessary for efficient incremental graph computation. Aspen [19] develops a compressed tree data structure which can store streaming graphs with reduced memory usage. While it supports the updates in the vertex attributes values, it does not support graphs larger than the memory size. Our delta-based dynamic graph store is focused on analytics workloads and avoids costly in-place updates of the data on disk. *iTurboGraph* also exploits the cost-based delta-maintenance strategy of vertex attribute values for efficient processing of incremental *NGA*.

8 CONCLUSION

We have presented *iTurboGraph*, a language, compiler, and runtime engine for incremental neighbor-centric graph analytics (*NGA*). *iTurboGraph* provides an intuitive programming language for *NGA* called L_{NGA}. We have proposed Graph Streaming Algebra (*GSA*) as a theoretical foundation for scalable and efficient streaming processing of incremental *NGA* without maintaining the intermediate results of the graph traversals in *NGA*. Once the L_{NGA} programs are compiled into *GSA*, our compiler performs automatic query incrementalization using the algebraic properties of *GSA*. We have presented several key optimization techniques for efficient executions of the incremental *NGA*. Our dynamic graph store efficiently maintains the graph mutations and vertex attribute values. Through extensive experiments, we have shown that *iTurboGraph* efficiently scales to large-scale graphs for incremental *NGA*.

ACKNOWLEDGMENT

We appreciate the anonymous reviewers for the valuable comments. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-53.

REFERENCES

[1] 2002. *Laboratory for Web Algorithmics: gsh-2015.* http://law.di.unimi.it/webdata/gsh-2015/.

[2] 2012. *The lemur project: Clueweb12 web graph.* http://www.lemurproject.org/clueweb12.

[3] 2019. *Twitter reveals its daily active user numbers for the first time.* https://www.washingtonpost.com/technology/2019/02/07/twitter-reveals-its-daily-active-user-numbers-first-time/.

[4] 2020. Apache Flink. https://ci.apache.org/projects/flink/flink-docs-release-1.11/.

[5] 2020. Apache Spark Streaming. https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison-Wesley Reading.

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018).

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[9] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[10] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *ACM SIGMOD Record* 15, 2 (1986), 61–71.

[11] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[12] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*. 1–8.

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[14] Cheng Chen, Hejun Wu, Dyce Jing Zhao, Da Yan, and James Cheng. 2016. SGraph: A Distributed Streaming System for Processing Big Graphs. In *International Conference on Big Data Computing and Communications*. Springer, 285–294.

[15] Unnikrishnan Cheramangalath, Rupesh Nasre, and YN Srikant. 2015. Falcon: A graph manipulation language for heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2015), 1–27.

[16] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C Myers. 2014. Using Program Analysis to Improve Database Applications. *IEEE Data Eng. Bull.* 37, 1 (2014), 48–59.

[17] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.

[18] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[19] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.

[20] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[21] Mostafa Elhemali, César A Galindo-Legaria, Torsten Grabs, and Milind M Joshi. 2007. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 993–1004.

[22] Zeynep Ertem, Alexander Veremyev, and Sergiy Butenko. 2016. Detecting large cohesive subgroups with high clustering coefficients in social networks. *Social Networks* 46 (2016), 1–10.

[23] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[24] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1889–1892.

[25] César Galindo-Legaria and Milind Joshi. 2001. Orthogonal optimization of subqueries and aggregation. *ACM SIGMOD Record* 30, 2 (2001), 571–581.

[26] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. 2010. Walking in facebook: A case study of unbiased sampling of osns. In *2010 Proceedings IEEE Infocom*. Ieee, 1–9.

[27] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[28] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. 1992. Counting solutions to the View Maintenance Problem.. In *Workshop on deductive databases, JICSLP*. 185–194.

[29] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[30] Matthew A Hammer, Umut A Acar, and Yan Chen. 2009. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 25–37.

[31] Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. 2014. Adapton: Composable, demand-driven incremental computation. *ACM SIGPLAN Notices* 49, 6 (2014), 156–166.

[32] Taher Haveliwala. 1999. *Efficient computation of PageRank.* Technical Report. Stanford.

[33] Petter Holme and Beom Jun Kim. 2002. Growing scale-free networks with tunable clustering. *Physical review E* 65, 2 (2002), 026107.

[34] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 349–362.

[35] Wole Jaiyeoba and Kevin Skadron. 2019. Graphtinker: A high performance data structure for dynamic graph processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1030–1041.

[36] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. 2015. Graph analytics using vertica relational database. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1191–1200.

[37] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1695–1698.

[38] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*. 1231–1245.

[39] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++ A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data*. 395–410.

[40] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 87–98.

[41] Pradeep Kumar and H Howie Huang. 2019. Graphone: A data store for real-time analytics on evolving graphs. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 249–263.

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, 591–600.

[43] Serge Lang. 1967. Algebraic structures. (1967).

[44] P-A Larson. 2002. Data reduction by partial preaggregation. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 706–715.

[45] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. 2019. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 459–474.

[46] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[47] Jure Leskovec and Julian McAuley. 2012. Learning to discover social circles in ego networks. *Advances in neural information processing systems* 25 (2012), 539–547.

[48] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[49] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[50] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[51] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[52] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[53] R Meusel, O Lehmberg, C Bizer, and S Vigna. 2014. *Web data commons-hyperlink graphs.* http://webdatacommons.org/hyperlinkgraph/.

[54] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. 2017. Analyzing IO amplification in Linux file systems. *arXiv preprint arXiv:1707.08514* (2017).

[55] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[56] M Tamer Özsu. 2020. Streaming graph processing and analytics. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 1–1.

[57] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental maintenance for non-distributive aggregate functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 802–813.

[58] Himchan Park and Min-Soo Kim. 2017. TrillionG: A Trillion-scale Synthetic Graph Generator using a Recursive Vector Model. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 913–928.

[59] Himchan Park and Min-Soo Kim. 2018. EvoGraph: an effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2051–2059.

[60] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2014. NScale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1673–1676.

[61] Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: automatic inlining of scalar UDFs into SQL queries with Froid. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1810–1813.

[62] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[63] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.

[64] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog.. In *Datalog*. 56–67.

[65] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* (2019), 1–24.

[66] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 1–12.

[67] Jiwon Seo, Stephen Guo, and Monica S Lam. 2015. SociaLite: An efficient graph query language based on datalog. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1824–1837.

[68] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1906–1917.

[69] Ajeet Shankar and Rastislav Bodik. 2007. DITTO: Automatic incrementalization of data structure invariant checks (in Java). *ACM SIGPLAN Notices* 42, 6 (2007), 310–319.

[70] Ehsan Sherkat, Maseud Rahgozar, and Masoud Asadpour. 2015. Structural link prediction based on ant colony approach in social networks. *Physica A: Statistical Mechanics and its Applications* 419 (2015), 80–94.

[71] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[72] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 532–543.

[73] Konrad Stocker, Donald Kossmann, R Braumandi, and Alfons Kemper. 2001. Integrating semi-join-reducers into state-of-the-art query processors. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 575–584.

[74] Charalampos E Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. 2011. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining* 1, 2 (2011), 75–81.

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

[76] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–27.

[77] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[78] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[79] Richard Wesley and Fei Xu. 2016. Incremental computation of common windowed holistic aggregates. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1221–1232.

[80] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big graph analytics platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

[81] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. 1307–1317.

[82] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.

[83] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2017. Graphd: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Transactions on Parallel and Distributed Systems* 29, 1 (2017), 99–114.

[84] Timothy AK Zakian, Ludovic AR Capelli, and Zhenjiang Hu. 2019. Incrementalization of vertex-centric programs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1019–1029.

[85] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[86] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, and Zechao Shang. 2015. Divide & conquer: I/O efficient depth-first search. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 445–458.

[87] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. 2014. MOCgraph: Scalable distributed graph processing using message online computing. *Proceedings of the VLDB Endowment* 8, 4 (2014), 377–388.

[88] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*.

[89] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. [n.d.]. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 ([n. d.]).

[90] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 375–386.

[91] Xiaojin Zhu and Zoubin Ghahramani. 2002. Learning from labeled and unlabeled data with label propagation. (2002).