Incremental Graph Computations: Doable and Undoable

Citation for published version:
https://doi.org/10.1145/3035918.3035944

Digital Object Identifier (DOI):
10.1145/3035918.3035944

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
SIGMOD '17 Proceedings of the 2017 ACM International Conference on Management of Data

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Incremental Graph Computations: Doable and Undoable

Wenfei Fan¹,² Chunming Hu² Chao Tian¹,²
¹University of Edinburgh  ²Beihang University
{wenfei@inf, chao.tian@}ed.ac.uk, hucm@buaa.edu.cn

ABSTRACT
The incremental problem for a class \( Q \) of graph queries aims to compute, given a query \( Q \in Q \) and \( G \) as input, changes \( \Delta G \) to \( G \) as output \( Q(G) \) and updates \( \Delta O \) to \( O(G) \) such that \( Q(G) + \Delta G = Q(G) \pm \Delta O \). It is called bounded if its cost is determined by the sizes of \( \Delta G \) and \( \Delta O \). It is to reduce the query size to \( \Delta G \) and \( \Delta O \). No matter how desirable, our first results are negative: for common graph queries such as graph traversal, connectivity, keyword search and pattern matching, their incremental problems are unbounded.

In light of the negative results, we propose two characterizations for the effectiveness of incremental computation: (a) localizable, if its cost is determined by the sizes of \( \Delta G \) and \( \Delta O \). We refer to \( T \) as an incremental algorithm for \( Q \) if the cost of \( T \) to \( \Delta G \) and \( \Delta O \) is small and is much less costly to compute than \( Q(G) \pm \Delta G \). Moreover, real-life graphs are constantly changing. It is often too costly to recompute \( Q(G) \pm \Delta G \) starting from scratch in response to frequent \( \Delta G \).

The need for incremental computations is evident. Real-life graphs \( G \) are often big, e.g., the social graph of Facebook has billions of nodes and trillions of edges [23]. Graph queries are expensive, e.g., subgraph isomorphism is \( \text{NP} \)-complete (cf. [35]). Moreover, real-life graphs are constantly changing. It is often too costly to recompute \( Q(G) \pm \Delta G \) starting from scratch in response to frequent \( \Delta G \).

These highlight the need for incremental algorithms \( T \): we use a batch algorithm \( T \) to compute \( Q(G) \) once, and then employ incremental \( T \) to compute changes \( \Delta O \) to \( Q(G) \) in response to \( \Delta G \). The rationale behind this is that in the real world, changes are typically small, e.g., less than 5% on the entire Web in a week [34]. When \( \Delta G \) is small, \( \Delta O \) is often small, and is much less costly to compute than \( Q(G) \pm \Delta G \), by making use of previous computation \( Q(G) \). In addition, incremental computations are crucial to parallel query processing [18, 21] that partitions a big graph, partially evaluates queries on the fragments at different processors, and conducts iterative computations incrementally to reduce the cost.

When \( \Delta G \) is small and \( G \) is big, can we guarantee that it is more efficient to compute \( \Delta O \) with \( T \) than to recompute \( Q(G) \pm \Delta G \) with \( T \)? A traditional characterization is by means of a notion of boundedness proposed in [44] and extended to graphs in [17, 38]. It measures the cost of \( T \) in \(|\text{CHANGED}| = |\Delta G| + |\Delta O| \), the size of the changes in the input and output. We say that \( T \) is bounded if its cost can be expressed as a polynomial function of \(|\text{CHANGED}| \) and \(|Q| \). The incremental problem for \( Q \) is bounded if there exists a bounded \( T \) for \( Q \), and is unbounded otherwise.

Bounded \( T \) allows us to reduce the incremental computations on big graphs to small graphs. Its cost is determined by \(|\text{CHANGED}| \) and query size \(|Q| \), rather than by the size \(|G| \) of the entire \( G \). In the real world, \(|Q| \) is typically small; moreover, \(|\text{CHANGED}| \) represents the updating cost that is inherent to the incremental problem itself, and is often much smaller than \(|G| \). Hence bounded \( T \) warrants efficient incremental computation no matter how big \( G \) is.

Undoable. No matter how desirable, we show that the incremental problem for \( Q \) is unbounded when \( Q \) ranges over graph traversal (RPQ), regular path queries, strongly connected components (SCC) and keyword search (KWS). The negative results hold when \( \Delta G \) consists of a single edge dele-
tion or insertion. Add to it the unboundedness of graph pattern matching via subgraph isomorphism (ISO) [17]. For these common queries, a bounded incremental algorithm is beyond reach. That is, by the standard of boundedness, incremental graph algorithms seem not very helpful.

**Doable.** The situation is not so hopeless. The boundedness of [17, 38, 44] is often too strong to evaluate incremental algorithms. To characterize the effectiveness of real-life incremental algorithms, we propose two alternative measures.

1. **Localizable computations.** We say that the incremental problem for \( Q \) is localizable if there exists an incremental algorithm \( T_Q \) such that for any \( Q \in Q, G \) and \( \Delta G \), its cost is determined by \( |Q| \) and the \( d_Q \)-neighbors of nodes in \( \Delta G \), where \( d_Q \) is decided by \( |Q| \) only. In practice, \( Q \) is typically small, and so is \( d_Q \). Hence it allows us to reduce the computations on (big) \( G \) to small \( d_Q \)-neighbors of \( \Delta G \).

   We show that the incremental problems for KWS and ISO are localizable, although they are unbounded.

2. **Relative boundedness.** We often want to incrementalize a batch algorithm \( T \) for \( Q \). For a query \( Q \in Q \) and a graph \( G \), we denote by \( G(T, Q) \) the part of data in \( G \) inspected by \( T \) when computing \( Q(G) \). Given updates \( \Delta G \) to \( G \), denote by \( \text{AFF} \) the difference between \( (G \oplus \Delta G)(T, Q) \) and \( G(T, Q) \).

   An incremental algorithm \( T_Q \) for \( Q \) is bounded relative to \( T \) if its cost is a polynomial in \( |\Delta G|, |Q| \) and \( |\text{AFF}| \). Intuitively, \( \text{AFF} \) indicates the necessary cost for incrementalizing \( T \), and \( T_Q \) incurs this minimum cost, not measured in \( |G| \).

   We show that RPQ and SCC are relatively bounded, i.e., it is possible to incrementalize their popular batch algorithms \( T \) and minimize unnecessary recomputation of \( T \).

**Contributions.** The paper studies the effectiveness of incremental graph computations, and provides the following.

1. **Impossibility results.** We show that no bounded incremental algorithms exist for RPQ, SCC, and KWS (Section 3). We establish these impossibility results either by elementary proofs or by reductions from incremental graph problems that are already known unbounded. To the best of our knowledge, this work gives the first proofs by reductions for unbounded graph incremental computations.

2. **New characterizations.** We characterize localizable incremental computations and relative boundedness in Sections 4 and 5, respectively. We show that the incremental computations above are either localizable (KWS and ISO) or relatively bounded (RPQ and SCC). That is, while these incremental computations are unbounded, they can still be effectively conducted with performance guarantees.

3. **Incremental algorithms.** As a proof of concept, we develop localized incremental algorithms for KWS and ISO (Section 4), and bounded incremental algorithms for RPQ and SCC relative to their batch algorithms (Section 5). We also develop optimization techniques for processing batch updates. These extend the small library of existing incremental graph algorithms that have performance guarantees.

4. **Experimental study.** We evaluate the algorithms using real-life and synthetic graphs (Section 6). We find that (a) our localizable and relatively bounded incremental algorithms for KWS, RPQ, SCC and ISO are effective. They outperform their batch counterparts even when \( |\Delta G| \) is up to 30%, 35%, 25% and 25% of \(|G|\), respectively, and are on average 4.9, 6.2, 2.9 and 3.7 times faster when \( |\Delta G| \) accounts for 10% of \(|G|\). (b) They scale well with \(|G|\). For instance, they take 28, 100, 19 and 225 seconds, respectively, on \( G \) with 50 million nodes and 100 million edges, under 5% updates, as opposed to 197, 1172, 144 and 2386 seconds by batch algorithms. (c) Our optimization strategies are effective: they improve the performance by 1.6 times on average.

**Related work.** We categorize the related work as follows.

**Bounded incremental algorithms.** Proposed in [44], the notion was studied for graph algorithms in [17, 38, 39]. A number of incremental algorithms have been developed for graphs [12, 17, 26, 28, 32, 38–42, 46] (see [16] for a survey). However, their costs are typically studied in terms of *amortized* analysis for averaged operation time of a sequence of unit updates to \( G \), not in the size of changes that is inherent to the incremental problem itself. To the best of our knowledge, bounded algorithms are only in place for the shortest path problems, single-source or all pairs, with positive lengths [38, 39]. It is known that the incremental problem is unbounded for subgraph isomorphism ISO [17], and for single-source reachability to all vertices [38].

As the notion of boundedness is often too strong, a weaker standard was introduced in [17], based on a notion of affected area \( \text{AFF} \). Intuitively, \( \text{AFF} \) covers not only changes \( \Delta O \), but also data that is necessarily checked to detect \( \Delta O \) by all incremental algorithms for \( Q \), encoded in auxiliary structures. An incremental algorithm is *semi-bounded* [17] if (a) its cost can be expressed as a polynomial in \( |\text{AFF}|, |Q| \) and \( |\Delta G| \), and (b) the size of the auxiliary structure is bounded by a polynomial in \(|G|\). The incremental problem for graph simulation is shown semi-bounded [17].

This work differs from the prior work in the following. (a) We establish new unboundedness results for RPQ, SCC and KWS, and a new form of reductions as proof techniques. (b) We propose measures for the effectiveness of incremental graph algorithms. In contrast to [17, 38, 39], localizable algorithms are characterized by \( d_Q \)-neighbors of \( \Delta G \) instead of \( \Delta O \) or \( \text{AFF} \). Relative boundedness is defined in terms of the affected area \( \text{AFF} \) relative to a specific algorithm \( T \), as opposed to \( \text{AFF} \) for all incremental algorithms for \( Q \) (semi-boundedness). (c) We develop incremental algorithms for RPQ, SCC, KWS and ISO with performance guarantees under the new measures, although they are unbounded.

**Locality of graph computations.** There have been batch algorithms that capitalize on the data locality of queries, for (parallel) subgraph isomorphism (*e.g.*, [19, 20]). Incoop [9], a generic MapReduce framework for incremental computations, also makes use of the locality of previously computed results in its scheduling algorithm to prevent straggling. To the best of our knowledge, the study of localizable incremental algorithms is the first effort to characterize the effectiveness of incremental algorithms in terms of locality.

**Relative boundedness.** There has also been work on incrementalizing batch algorithms, notably self-adjusting computations [5, 10]. The idea is to track the dependencies between data and function calls as a dynamic dependency graph [6], upon which functions that are affected by the changes in the input can be identified and recomputed. Memorization [36] is used to record and reuse the results of function calls when possible. It is a general-purpose, language-centric technique for programs to automatically respond to modifications to their data. In contrast, relative boundedness is to charac-
terize whether it is feasible to incrementalize a given batch algorithm \( T \) with cost measured in the size of affected area AFF inspected by \( T \), not in terms of function calls.

**View maintenance.** Related is also view maintenance for updating materialized views, which has been studied for relational data [14, 24, 25], object-oriented databases [31], and semi-structured data modeled as graphs [4, 46]. Various methods have been proposed, e.g., an algebraic approach of [11] for XML views and the use of key constraints [24] for relations. However, few of them have provable performance guarantees, and fewer can be applied to graph queries. In particular, the techniques of [4, 46] are developed for views specified as selection paths, and do not apply to graph queries studied in this paper. In contrast, we study the boundedness of incremental graph problems and provide algorithms that are localizable or relatively bounded.

## 2. INCREMENTAL COMPUTATIONS

We first present graph queries studied in this paper, and then formulate their incremental problems.

We start with basic notations.

We consider directed graphs \( G \) represented as \((V,E,l)\), where (1) \( V \) is a finite set of nodes; (2) \( E \subseteq V \times V \) is a set of edges in which \((v,v')\) denotes an edge from \( v \) to \( v' \), and (3) each node \( v \) in \( V \) carries \( l(v) \), indicating its label and content, as found in social networks and property graphs.

If \((v,w)\) is an edge in \( E \), we refer to node \( w \) as a successor of \( v \), and to node \( v \) as a predecessor of \( w \).

Graph \( G_s = (V_s,E_s,l_s) \) is a subgraph of \( G \) if \( V_s \subseteq V \), \( E_s \subseteq E \), and for each node \( v \in V_s \), \( l_s(v) = l(v) \).

Subgraph \( G_s \) is induced by \( V_s \) if \( E_s \) consists of all the edges in \( G \) such that their endpoints are both in \( V_s \).

### 2.1 Graph Queries

We study the following four classes of graph queries.

**RPQ.** Consider directed graphs \( G = (V,E,l) \) over a finite alphabet \( \Sigma \) of labels defined on the nodes in \( V \). A path \( p \) from \( v_0 \) to \( v_n \) in \( G \) is a list \((v_0,\ldots,v_n)\), where for \( i = 0, n + 1 \), \((v_i,v_{i+1})\) is an edge in \( G \). The length of path \( p \) is \( n \).

A regular path query \( Q \) is a regular expression as follows:

\[
Q := \epsilon | a | Q \cdot Q | Q + Q | \neg Q.
\]

Here \( \epsilon \) denotes an empty path; \( a \) is a label from \( \Sigma \); \( + \) and \( \cdot \) are concatenation and union operators, respectively; and \( \neg \) indicates zero or more occurrences of \( Q \).

We use \( L(Q) \) to denote the regular language defined by \( Q \), i.e., the set of all strings that can be parsed by \( Q \). For a path \( p = (v_0,\ldots,v_n) \) in \( G \), we use \( l(p) \) to denote the labels \( l(v_0) \cdot \cdots \cdot l(v_n) \) of the nodes on \( p \). A match of \( Q \) in \( G \) is a pair \((v,w)\) of nodes such that there exists a path \( p \) from \( v \) to \( w \) having \( l(p) \in L(Q) \). RPQ is stated as follows.

- **Input:** A directed graph \( G \) and a regular path query \( Q \).
- **Output:** The set \( Q(G) \) of all matches of \( Q \) in \( G \).

It takes \( O(|V| |E| |Q|^2 \log^2 |Q|) \) time to compute \( Q(G) \) by using NFA (nondeterministic finite automaton) [29, 33], where \(|Q|\) is the number of occurrences of labels from \( \Sigma \) in \( Q \) [29].

**SCC.** A subgraph \( G_s \) of a directed graph \( G \) is a strongly connected component of \( G \) if it is (a) strongly connected, i.e., for any pair \((v,v')\) of nodes in \( G_s \), there is a path from \( v \) to \( v' \) and vice versa, and (b) maximum, i.e., adding any node or edge to \( G_s \) makes it no longer strongly connected.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q(G) )</td>
<td>the answers to query ( Q ) in graph ( G )</td>
</tr>
<tr>
<td>( \Delta G )</td>
<td>updates to graph ( G ) (edge insertions, deletions)</td>
</tr>
<tr>
<td>( G \oplus \Delta G )</td>
<td>the graph obtained by updating ( G ) with ( \Delta G )</td>
</tr>
<tr>
<td>( \Delta G )</td>
<td>updates to old output ( Q(G) ) in response to ( \Delta G )</td>
</tr>
<tr>
<td>( T )</td>
<td>an incremental algorithm for ( \mathcal{Q} )</td>
</tr>
<tr>
<td>AFF</td>
<td>changes to the area inspected by a batch algorithm ( T )</td>
</tr>
<tr>
<td>( \text{dist}(s,t) )</td>
<td>the shortest distance from node ( s ) to ( t )</td>
</tr>
<tr>
<td>( G_s(v) )</td>
<td>the ( d )-neighbor of node ( v ) in ( G_s )</td>
</tr>
</tbody>
</table>

**Table 1: Notations**

We use \( SCC(G) \) to denote the set of all strongly connected components of \( G \). The SCC problem is stated as follows.

- **Input:** A directed graph \( G \).
- **Output:** \( SCC(G) \).

It is known that SCC is in \( O(|V| + |E|) \) time [43].

**KWS.** We consider keyword search with distinct roots in the same setting of [37]. A keyword query \( Q \) is of the form \((k_1,\ldots,k_m)\), where each \( k_i \) is a keyword. Given a directed graph \( G \) and a bound \( b \), a match to \( Q \) in \( G \) at node \( r \) is a tree \( T(r,p_1,\ldots,p_m) \) such that \( (a) \) \( T \) is a subgraph of \( G \), and \( r \) is the root of \( T \); \((b) \) for each \( i \in [1,m] \), \( p_i \) is a node in \( T \) such that \( l(p_i) = k_i \), i.e., it matches keyword \( k_i \); \((c) \) \( \text{dist}(r,p_i) \leq b \), and \((d) \) the sum \( \Sigma_{i \in [1,m]} \text{dist}(r,p_i) \) is the smallest among all such trees. Here for a pair \((r,s)\) of nodes, \( \text{dist}(r,s) \) denotes the shortest distance from \( r \) to \( s \), i.e., the length of a shortest path from \( r \) to \( s \). KWS is as follows.

- **Input:** A directed graph \( G \), a keyword query \( Q = (k_1,\ldots,k_m) \), and a positive integer \( b \).
- **Output:** The set \( KWS(Q,G,b) \) of all matches to \( Q \) at node \( r \) in \( G \) within \( b \) hops, for \( r \) ranging over all nodes in \( G \).

It can be computed in \( O(m(|V| \log |V| + |E|)) \) time (cf. [45]).

**ISO.** A pattern query \( Q \) is a graph \((V_Q,E_Q,l_Q)\), in which \( V_Q \) and \( E_Q \) are the set of pattern nodes and directed edges, respectively, and each node \( u \) in \( V_Q \) has a label \( l_Q(u) \).

A match of \( Q \) in \( G \) is a subgraph \( G_s \) of \( G \) that is isomorphic to \( Q \), i.e., there exists a bijective function \( h \) from \( V_Q \) to the set of nodes of \( G_s \) such that \( (a) \) for each node \( u \in V_Q \), \( l_Q(u) = l(h(u)) \), and \( (b) \) \( u, u' \) is an edge in \( Q \) iff \( (h(u),h(u')) \) is an edge in \( G_s \). The answer \( ISO(Q,G) \) to \( Q \) in \( G \) is the set of all matches of \( Q \) in \( G \). ISO is stated as follows.

- **Input:** A directed graph \( G \) and a pattern \( Q \).
- **Output:** The set \( ISO(Q,G) \) of all matches of \( Q \) in \( G \).

It is NP-complete to decide whether \( ISO(Q,G) \) is empty (cf. [35]).

### 2.2 Incremental Query Answering

We next formalize incremental computation problems.

**Updates.** We consider w.l.o.g. the following unit updates:

- **edge insertion:** \((\text{insert} e)\), possibly with new nodes, and
- **edge deletion:** \((\text{delete} e)\).

A batch update \( \Delta G \) to graph \( G \) is a sequence of unit updates.

**Incremental problem.** For a class \( \mathcal{Q} \) of graph queries, the incremental problem is stated as follows.

- **Input:** Graph \( G \), query \( Q \in \mathcal{Q} \), old output \( Q(G) \), and updates \( \Delta G \) to the input graph \( G \).
- **Output:** Updates \( \Delta \) to the output such that

\[
Q(G \oplus \Delta G) = Q(G) \oplus \Delta \).
\]

We study the problem for RPQ, SCC, KWS and ISO.

The notations of this paper are summarized in Table 1.

### 3. BOUNDED PROBLEMS: UNDOABLE

This section shows the following impossibility results.
Theorem 1: The incremental problem is unbounded for
- regular path queries (RPQ),
- strongly connected components (SCC), and
- keyword search (KWS),
even under a unit edge deletion and a unit edge deletion. □

Together with the unboundedness of ISO [17], Theorem 1 tells us that it is impossible to find bounded incremental algorithms for all the graph query classes presented in Section 2. The negative results are rather robust: the incremental problems are already unbounded under unit updates. Before we give a proof, we first review the notion of boundedness of [17,38], and introduce a form of Δ-reductions.

Boundedness. An incremental algorithm $T_\Delta$ for a graph query class $Q$ is bounded if its cost can be expressed as a polynomial of $|\text{CHANGED}|$ and $|Q|$, where $|\text{CHANGED}| = |\Delta G| + |\Delta O|$. Following [17,38], we require $T_\Delta$ to be locally persistent. Such $T_\Delta$ may use (a) auxiliary structures associated with each node $v$ of $G$, to keep track of intermediate results at $v$, and (b) pointers to its successors and predecessors. However, no global auxiliary information is allowed, such as pointers to nodes other than its neighbors; similarly for edges. The algorithm starts an update from the nodes or edges involved in $\Delta G$, and traverses $G$ following the edges of $G$. The choice of which edge to follow depends only on the information accumulated in the current processing of $G$ since global information from prior passes is not maintained.

Reductions. We now introduce Δ-reduction. Consider two classes of graph queries $Q_1$ and $Q_2$. For $i \in \{1, 2\}$, we represent an instance of (the computational problem for) $Q_i$ as $I_i = (Q_i, G_i)$, where $Q_i \in Q_i$ and $G_i$ is a graph.

A Δ-reduction from $Q_1$ to $Q_2$ is a triple $(f, f_1, f_2)$ of functions such that for each instance $I_i = (Q_i, G_i)$ of $Q_i$,
1. $f(I_1)$ is an instance $I_2 = (Q_2, G_2)$ of $Q_2$; and
2. for all updates $\Delta G_i$ to $G_i$,
   a. $f_1(\Delta G_1)$ computes updates $\Delta G_2$ to $G_2$; and
   b. $f_2(\Delta O_1)$ computes $\Delta O_1$, where $\Delta O_i$ denotes updates to $Q_i(G_i)$ in response to $\Delta G_i$, for $i \in \{1, 2\}$, in polynomial-time (PTIME) in $|\Delta G_i| + |\Delta O_i|$ and $|Q_i|$.

Intuitively, $f$ maps the instances of $Q_1$ to $Q_2$; $f_1$ maps input updates $\Delta G_1$ to $\Delta G_2$, and $f_2$ maps output updates $\Delta O_2$ back to $O_1$, both in PTIME in the size of $Q_1$ and changes in the input and output of instance $(Q_1, G_1)$, where $(Q_2, G_2)$ corresponds to $(Q_1, G_1)$ via function $f$. Hence if $Q_2$ has a bounded incremental algorithm, then so does $Q_1$. Equivalently, if $Q_1$ is unbounded, neither is $Q_2$. That is, Δ-reduction preserves boundedness (see Appendix for a proof).

Lemma 2: If there exists a Δ-reduction from $Q_1$ to $Q_2$ and the incremental problem for $Q_2$ is bounded, then the incremental problem for $Q_1$ is also bounded. □

Proof of Theorem 1. Based on Δ-reduction, we outline a proof, which reveals the challenges to the development of incremental algorithms. The proofs for RPQ, SCC and KWS are nontrivial (see Appendix and [2]). For each query class, we need to give two proofs: one under a unit edge deletion, and the other under a unit insertion. Indeed, a problem may be unbounded under deletions (resp. insertions) but be bounded under insertions (resp. deletions). An example is SSRP, the single-source reachability problem to all vertices. It is to decide, given a graph $G$ and a node $v_0$ in $G$, whether there exists a path from $v_0$ to $v_1$ for all nodes $v_1$ in $G$. It is known that SSRP is unbounded under unit edge deletions but bounded under unit edge insertions [38].

RPQ. We show that the incremental problem for RPQ is unbounded under a unit edge deletion by Δ-reduction from SSRP, whose incremental problem is unbounded under unit deletions. We show the unboundedness under unit edge insertions by giving an elementary proof. We construct an instance $(Q, G)$ of RPQ, and show by contradiction that there exists no bounded incremental algorithm that can correctly compute $Q(G \oplus \Delta G)$ in response to updates $\Delta G$ to $G$.

SCC. We prove the unboundedness of the case under a unit edge deletion also by Δ-reduction from SSRP. The case under unit edge updates is verified by contradiction. □

4. LOCALIZABLE COMPUTATIONS

Not all is lost. Despite Theorem 1, there exist efficient incremental algorithms for RPQ, SCC, KWS and ISO with performance guarantees under new characterizations for the effectiveness of incremental algorithms. In this section we introduce one of the standards, namely, localizable incremental computations. We first present the notion (Section 4.1). We then show that the incremental problems for KWS and ISO are localizable (Section 4.2 and Appendix, respectively).

4.1 Locality of Incremental Computations

We start with a few notations. (a) In a graph $G$, we say that a node $v'$ is within $d$ hops of $v$ if $\text{dist}(v, v') \leq d$ by taking $G$ as an undirected graph. (b) We denote by $V_d(v)$ the set of all nodes in $G$ that are within $d$ hops of $v$. (c) The $d$-neighbor $G_d(v)$ of $v$ is the subgraph of $G$ induced by $V_d(v)$, in which the set of edges is denoted by $E_d(v)$.

Consider a graph query class $Q$. An incremental algorithm $T_\Delta$ for $Q$ is localizable if its cost is determined only by $|Q|$ and the sizes of the $d_Q$-neighbors of those nodes on the edges of $\Delta G$, where $d_Q$ is determined by the query size $|Q|$.

The incremental problem for $Q$ is called localizable if there exists a localizable incremental algorithm for $Q$.

Intuitively, if $T_\Delta$ is localizable, it can compute $\Delta O$ by inspecting only $G_{d_Q}(v)$, i.e., nodes within $d_Q$ hops of nodes $v$ in $\Delta G$. In practice, $G_{d_Q}(v)$ is often small. Indeed, (a) $Q$ is typically small; e.g., 98% of real-life pattern queries have radius 1, and 1.8% have radius 2 [22]; hence so is $d_Q$; and (b) real-life graphs are often sparse; for instance, the average node degree is 14.3 in social graphs [13]. Hence, $T_\Delta$ can reduce the computations on possibly big $G$ to small $G_{d_Q}(v)$.

The main results of this section are as follows.

Theorem 3: The incremental problem is localizable for KWS and ISO under batch updates. □

That is, while the incremental problems for KWS and ISO are unbounded, we can still effectively conduct their incremental computations by making big graphs “small”.

As a constructive proof of Theorem 3, we next develop localizable incremental algorithms for KWS. The incremental algorithms for ISO are similar and are outlined in Appendix.
Algorithm: IncKWS$^+$

Input: A graph $G$ with $\text{kdist}(\cdot)$, keyword query $Q$ and bound $b$, matches $Q(G)$, and an edge $(v, w)$ to be inserted.

Output: The updated matches $Q(G \oplus \Delta G)$ and $\text{kdist}$ lists.

1. for each $k_i$ in $Q$ with
2. \hspace{1em} $\text{kdist}(w)[k_i].\text{dist} < \min(\text{kdist}(v)[k_i].\text{dist} - 1, b)$ do
3. \hspace{2em} $\text{kdist}(v)[k_i].\text{dist} := \text{kdist}(w)[k_i].\text{dist} + 1$
4. \hspace{2em} $\text{kdist}(v)[k_i].\text{next} := w$; queue $q_i := \text{nil}$; enqueue($v$);
5. while $q_i$ is not empty do
6. \hspace{2em} node $u := q_i$; dequeue($v$);
7. \hspace{2em} for each predecessor $u'$ of $u$ such that
8. \hspace{4em} $\text{kdist}(u')[k_i].\text{dist} < \min(\text{kdist}(u')'[k_i].\text{dist} - 1, b)$ do
9. \hspace{6em} $\text{kdist}(u')[k_i].\text{dist} := \text{kdist}(u')[k_i].\text{dist} + 1$
10. \hspace{6em} $\text{kdist}(u')[k_i].\text{next} := u$; $q_i$.enqueue($u'$);
11. replace ($u, u'_1$) with ($u, u'_2$) in all the matches of $Q(G)$ or add matches to $Q(G \oplus \Delta G)$ by including ($u, u'_2$);
12. return $Q(G \oplus \Delta G)$ (including revised $Q(G)$) and $\text{kdist}$.

Figure 1: Algorithm IncKWS$^+$

4.2 Localizable Algorithms for KWS

We first provide localizable algorithms for KWS under unit edge insertions and deletions. We then develop a localizable incremental algorithm for KWS to process batch updates.

Data structures. We start with an auxiliary structure. Recall that a KWS query consists of a list $Q$ of keywords and an integer bound $b$. For each node $v$ in graph $G$, we maintain a keyword-distance list $\text{kdist}(v)$. Its entries are of the form (keyword, dist, next), where dist is the shortest distance from $v$ to a node labeled keyword in $Q$, and next indicates the node on this shortest path next to $v$. A single shortest path is selected with a predefined order in case of a tie. Hence each root uniquely determines a match if it exists. Such keyword-distance lists are obtained after the execution of a batch algorithm. Indeed, existing batch approaches [8,27,30] for KWS traverse $G$ to find shortest paths from nodes to others matching keywords in $Q$. While they vary in search and indexing strategies, they all maintain something like $\text{kdist}(\cdot)$.

1. Unit insertions. Inserting an edge to graph $G$ may shorten the shortest distances from nodes to those matching keywords in $Q$, which is reflected as changes to dist and next in the keyword-distance lists on $G$. Based on this, we present an incremental algorithm, referred to as IncKWS$^+$ and shown in Fig. 1, to process unit edge insertions.

Given $\Delta G$ consisting of insert$(v, w)$, IncKWS$^+$ inspects whether it inclicts any change to shortest paths of existing matches; if so, it propagates the change, revises $\text{kdist}(v)$ entries for affected nodes $v$ and updates the matches accordingly. It proceeds until no more revision is needed. The search is confined in the $b$-neighbors of nodes in $\Delta G$, and hence localizable, where $b$ is the bound in the KWS query.

More specifically, IncKWS$^+$ first checks whether $(v, w)$ is on a shorter path within the bound $b$ from $v$ to nodes labeled $k_i$ in $Q$. If so, $\text{kdist}(v)$ is updated by adjusting dist and next (lines 2-3). IncKWS$^+$ then propagates the change to the ancestors of $v$ if their $\text{kdist}$ entries are no longer valid (lines 4-8). An FIFO (first-in-first-out) queue $q_i$ is used to control the propagation, following BFS (breadth-first-search). Each time when a node $u$ is dequeued from $q_i$, the predecessors of $u$ are inspected to check whether $u$ triggers updated shortest path from them within bound $b$, followed by updating their $\text{kdist}$ entries when needed (lines 6-8). These predecessors may be inserted into queue $q_i$ for further checking (line 8).

Figure 2: Example graph and matches of KWS

After revising the data structures, IncKWS$^+$ computes $Q(G \oplus \Delta G)$ based on the changes to next in $\text{kdist}(\cdot)$’s (lines 9-10), either by replacing some edges in existing matches, or by including new matches not in $Q(G)$. Note that all such affected edges are inside the $2b$-neighbors of $\Delta G$.

Example 1: Figure 2 gives a graph $G$ (with all solid edges and dotted $e_2, e_5$). Consider $Q = (a, d)$ and bound 2. Two trees $T_b$ and $T_d$ in $Q(G)$ are shown in Fig. 2 (solid edges).

When edge $e_1$ is added to $G$, denote by $G_1$ the graph after the insertion. IncKWS$^+$ finds that the shortest distance from $b_2$ to nodes matching $d$ in $G_1$ is reduced to 1 from 2. Thus it updates the entries in $\text{kdist}(b_2)[d]$ and propagates the change to $b_2$’s predecessors. The propagation stops at $c_2$ since the distance from it to $d$ nodes reaches bound 2. The values of (dist, next) in $\text{kdist}$ lists on $G$ are updated as follows.

Then IncKWS$^+$ revises $T_{b_2}$ by replacing the path starting with edge $(b_2, b_4)$ by $(b_2, d_1)$ to get $T_b$ in $Q(G_1)$, and a new match $T_{c_2}$ (solid edges in Fig. 2) is added to $Q(G_1)$.

Correctness & complexity. IncKWS$^+$ updates $\text{kdist}(\cdot)$’s correctly: it revises only entries in which dist values are decreased, and checks all affected entries by propagating the changes. From this the correctness of IncKWS$^+$ follows.

IncKWS$^+$ is in $O(m(|V_h(w)| + |E_h(w)|) + |V_h(w)| |E_2b_b(w)|)$ time. Updating $\text{kdist}(\cdot)$’s takes $O(m(|V_h(w)| + |E_h(w)|))$ time in total (lines 1-8), where $m$ is the number of keywords in $Q$. Observe the following: (a) each node with updated $\text{kdist}$ is verified at most $m$ times to check all the keywords in $Q$; and (b) only the data in $G_h(w)$ is inspected since change propagation stops as soon as the shortest distance exceeds $b$, i.e., $\text{kdist}(\cdot)$’s are partially updated for matches within bound $b$. Updating $Q(G)$ (lines 9-10) takes $O(|V_h(w)| |E_2b_b(w)|)$ time since the roots of the affected matches are within $b$ hops of $w$, and their edges to be adjusted are at most $2b$ hops away from $w$. Therefore, algorithm IncKWS$^+$ is localizable.

2. Unit deletions. The incremental algorithm for processing unit delete$(v, w)$ is shown in Fig. 3, denoted by IncKWS$^-$. In contrast to edge insertions, some shortest distances in $\text{kdist}$ lists may be increased by delete$(v, w)$. The main idea of IncKWS$^-$ is to identify those entries in $\text{kdist}(\cdot)$’s that are affected by $\Delta G$, and compute changes to dist and next. Similar to IncKWS$^+$, updating $\text{kdist}(\cdot)$’s is confined within the $b$-neighbors of $\Delta G$ by inspecting only those distances no longer than bound $b$. The identification and computation are separated into two phases in IncKWS$^-$. After consulting whether $(v, w)$ is on a shortest path from $v$ to some node labeled keyword $k_i$ within bound $b$, IncKWS$^-$ propagates the change to $v$’s predecessors if needed with the help of a stack $a_i$, and each predecessor that may have
Algorithm: lnKWS−

Input: G with kdist(·), Q, b, Q(G) as in lnKWS+, and delete(v, w).
Output: The updated matches Q(G ⊕ ΔG) and kdist lists.
1. for each ki in Q with w = kdist(v)[ki].next and kdist(w)[ki] < b do
   2. queue qi := nil; stack ai := nil; a, push(v); mark v affected;
   3. while ai is not empty do
      4. node u := a.pop();
      5. for each predecessor u′ of u that u = kdist(u′)[ki].next and kdist(u′)[ki] ≤ b do
         6. a, push(u′); mark u′ affected;
      7. for each affected node u do
         8. compute dist and next for kdist(u)[ki] based on those u’s successors that are not affected;
         9. q, insert(u, kdist(u)[ki].dist);
      10. while qi is not empty do
          11. (u, d) := q, pull.min();
          12. for each predecessor u′ of u with d < min(kdist(u′)[ki].dist − 1, b) do
             13. kdist(u′)[ki].dist := d + 1; kdist(u′)[ki].next := u;
             14. q, decrease(u′, kdist(u′)[ki].dist);
          15. for each u′′ and u′′′ involved in a changed kdist(u′′)[ki].next do
             16. replace (u′′, u′′′) with (u′′, u′′′) in all the matches of Q(G) or remove matches from Q(G) by excluding (u′′, u′′′);
          17. return Q(G ⊕ ΔG) (updated Q(G) above) and kdist(·);

Figure 3: Algorithm lnKWS−

an updated shortest path to nodes matching ki is marked affected w.r.t. ki (lines 1-6). The propagation is similar to that of lnKWS+, by inspecting next values, and is conducted in the b-neighbors of v. Then the potential kdist entries for those affected nodes are computed based on their successors that are not affected w.r.t. ki (line 8), and affected nodes with their potential dist values (as keys) are inserted into priority queue q (line 9) to compute exact dist values later. Indeed, the exact values of dist and next may depend on the affected successors, whose values also need to be determined.

The exact values of dist and next are computed in the second phase (lines 10-14). For node u with minimum dist that is removed from q, lnKWS− checks whether it leads to a new shortest path within bound b originated from predecessor u′ of u (lines 11-12). If so, values in kdist(u′)[ki] are updated, and the key of u′ of q is decreased (lines 13-14).

The process continues until q becomes empty. Matches in Q(G) are updated using the latest kdist lists (lines 15-16).

Example 2: Recall Q, G1 and Q(G1) from Example 1. Suppose that e2 is now removed from G1. This makes the shortest path from c2 to e2 in T2b split, and lnKWS− marks node e2 affected with keyword a. Since the shortest distance from successor b2 of e2 to nodes matching a equals the bound 2, lnKWS− concludes that node e2 cannot be the root of a match, and removes Tc2 of Example 1 from Q(G1).

Correctness & complexity. The correctness of lnKWS− is verified just like for lnKWS+, except that the exact values of kdist(·) may depend on multiple affected successors of v. lnKWS− runs in O(m|Vw| log |Vw| + |Ew|) + |Vw||Ew| time. For updating matches in addition to the cost for computing changes to kdist(·)’s. Its first phase (lines 1-9) takes O(m|Vw|[log|Vw| + |Ew|]) + |Vw||Ew| time, including O(|Vw||Ew|) for updating matches in addition to the cost for computing changes to kdist(·)’s. Its first phase (lines 1-9) takes O(m|Vw| log |Vw| + |Ew|) time since only the affected shortest paths of length bounded by b are identified. The second phase (lines 10-14) takes O(m|Vw| log |Vw| + |Ew|) time, the same as computing b-bounded shortest path from affected nodes to m sinks, i.e., nodes labeled a keyword from Q.

(3) Batch updates. We next give an incremental algorithm, denoted by lnKWS (not shown), to process batch updates ΔG = (ΔG+, ΔG−), where ΔG+ and ΔG− denote edge insertions and deletions, respectively. We assume w.l.o.g. that there exist no delete e in ΔG− and insert e in ΔG− for the same edge e, which can be easily detected.

Given batch updates ΔG, lnKWS inspects whether each unit edge deletion and insertion causes any change to existing matches, i.e., whether some of existing shortest paths become invalid and new shortest paths have to be generated; if so, it propagates the changes and updates the affected keyword-distance lists. The algorithm updates the same entry at most once even if it is affected by multiple updates in ΔG, by interleaving different change propagation with a global data structure to accommodate the effects of different unit updates. It works in three phases, as outlined below.

(a) lnKWS first identifies the affected nodes w.r.t. each keyword ki in Q due to ΔG− within the b-neighbors of ΔG−, and computes their potential dist and next values, using the same strategy of lnKWS−. Here all the affected nodes w.r.t. ki and their potential dist values are inserted into a single priority queue q to further compute exact values.

(b) The algorithm then checks whether each insert(v, w) leads to the creation of a shorter path within bound b when neither v nor w is affected w.r.t. ki by ΔG+. Insertions with affected nodes are not considered since dist value at w may no longer be correct due to ΔG−, or this edge has already been inspected to compute potential dist value for node v. If so, dist and next values are updated for kdist(v).

Unlike lnKWS+ that propagates this change to ancestors of v directly, it inserts node v and the updated dist value into queue q to interleave insert(v, w) with other updates in ΔG.

(c) After these, lnKWS computes exact next and dist values of kdist(·)’s, in the same way as we do in lnKWS− by making use of queue q. Note that all potential changes to kdist(·)’s caused by ΔG, including both deletions and insertions, are collected into the same q; in this way the algorithm guarantees that the exact value, i.e., shortest distance, is decided at most once for each entry affected. Matches in Q(G) are updated accordingly within the 2b-neighbors of ΔG at last.

Example 3: Consider Q and G of Example 1, and batch updates ΔG that insert edges e1, e3, e4 and delete e2 and e5.

Given these, lnKWS first identifies the affected nodes c1 and c2 w.r.t. a, and finds that the potential value of the corresponding dist exceeds the bound 2. Then it processes insertions; e.g., the insertion of e3 leads to decreased shortest distance from b2 to a nodes, and the change is propagated to c2 for computing the exact value of kdist(e2)[a], i.e., lnKWS interleave insert e3 and delete e2 to decide the exact shortest distance from c2 to a nodes. The other updates are handled similarly. Based on these, it replaces the two branches of T2b with (b2, a1) and (b2, d1), respectively, and adds match T4b in Fig. 2. A new match Tc2 is also generated, where path (c2, b2, a2) in Tc2 of Example 1 is replaced by (c2, b2, a1).

Correctness & complexity. For the correctness of lnKWS observe the following. (a) Each node that is affected w.r.t. keyword ki by any unit update in ΔG is inspected. (b) The dist values for these nodes are monotonically increasing and correctly computed, similar to its counterpart in lnKWS−. lnKWS is in O(m(|Vw|(ΔG) log |Vw| + |Ew|) + |Vw|(|ΔG|Ew|ΔG|)) time, where Vw(ΔG) (resp. Ew(ΔG))
denote the nodes (resp. edges) of the union of b-neighbors of nodes in $\Delta G$. Note that the final $k$-dist value of each affected node w.r.t. any keyword $k_i$ is determined once by using the global priority queue $q$. The complexity analysis is similar to that of IncKWS, except that here the 2b-neighbors of all the nodes involved in $\Delta G$ are possibly accessed.

Since the costs of IncKWS, IncKWS' and IncKWS are determined by $m$ and the size of 2b-neighbors of nodes involved in $\Delta G$ for a given bound $b$, they are all localizable.

**Remark**: Although the incremental algorithms for KWS are developed for a constant $b$, they can be readily extended to cope with $b$ that varies. More specifically, when change propagation stops at node $v$ due to bound $b$, we can annotate $v$ as a “breakpoint” w.r.t. $b$, and the set of all such breakpoints is stored as a “snapshot” of graph $G$ w.r.t. $b$. When given a larger $b'$, the snapshot is firstly restored and each breakpoint is regarded as a unit update to $G$, i.e., as input to the incremental algorithm with $b'$ in addition to $\Delta G$, from where the change propagation continues. In this way, KWS queries with different $b$ values can be answered using the same data structure, i.e., key-based distance-list that is consistently updated. Indeed, we only need to store the snapshot of $G$ w.r.t. the maximum $b$ that is encountered.

### 5. RELATIVE BOUNDEDNESS

We next introduce relative boundedness, another alternative characterization for the effectiveness of incremental computations. We first formalize the notion in Section 5.1. We then develop relatively bounded incremental algorithms for RPQ and SCC in Sections 5.2 and 5.3, respectively.

#### 5.1 Relative Boundedness

Consider a batch algorithm $T$ for a query class $Q$ that is proven effective and being widely used in practice. For a query $Q \in Q$ and a graph $G$, we denote by $G(T, Q)$ the data inspected by $T$ when computing $Q(G)$, including data in $G$ and possibly auxiliary structures used by $T$. For updates $\Delta G$ to $G$, we denote by AFF the difference between $(G \oplus \Delta G)(T, Q)$ and $G(T, Q)$, i.e., the difference in the data inspected by $T$ for computing $Q(G \oplus \Delta G)$ and for $Q(G)$.

An incremental algorithm $T_\Delta$ for $Q$ is bounded relative to $T$ if its cost can be expressed as a polynomial function in $|\Delta G|$, $|Q|$ and $|AFF|$ for all $Q \in Q$, graph $G$ and updates $\Delta G$. Note that the changes $\Delta Q$ to $Q(G)$ are included in AFF.

Intuitively, we only incrementalize batch algorithms $T$’s that have been verified effective. As batch algorithms have been studied for decades for graphs, a number of such algorithms are in place. When incrementalizing such algorithms, relative boundedness is to characterize the effectiveness of the incrementalization, i.e., whether it minimizes unnecessary recomputation in response to updates $\Delta G$. It suffices to develop $T_\Delta$ bounded relative to one of such $T$’s.

Note that for a class $Q$ of graph queries, one can find localizable incremental algorithms only if $Q$ has the data locality, i.e., to determine whether $v$ is in the answer $Q(G)$ to a query $Q$, it suffices to inspect the 2d-neighbor of $v$. However, many graph queries do not have the data locality, e.g., RPQ and SCC. For such queries, we can explore relatively bounded incremental algorithms. Moreover, even when $Q$ has the data locality, we want to find incremental algorithms that are both localizable and bounded relative to a practical batch algorithm $M_Q$ and intersection graph of $M_Q$, $G$ algorithm of $Q$. Such algorithms are particularly needed for large queries $Q$ (i.e., when diameter $d_Q$ of $Q$ is large).

We should remark that there are other alternative effectiveness characterizations for incremental graph algorithms, e.g., a classification in terms of incremental complexity. We focus on localizability and relative boundedness in this paper since they are easy to verify and use in practice.

The main results of this section are as follows.

**Theorem 4**: There are bounded incremental algorithms for RPQ and SCC relative to their batch counterparts.

As a proof, we present relatively bounded algorithms for RPQ and SCC. As will be seen in Section 6, these algorithms are effective although none of the query classes is bounded.

#### 5.2 Incrementalization for RPQ

We start with RPQ. Given a regular path query $Q$ and a graph $G$, it is to compute the set $Q(G)$ of matches of $Q$ in $G$, i.e., pairs $(v, w)$ of nodes in $G$ such that $v$ can reach $w$ by following a path in the regular language defined by $Q$.

We incrementalize a batch algorithm RPQ_{NFA} [29, 33] for RPQ. We first review RPQ_{NFA} and identify its AFF. We then give a bounded incremental algorithm relative to RPQ_{NFA}.

**Batch algorithm.** Algorithm RPQ_{NFA} consists of two phases. Given $Q$ and $G$, it first translates $Q$ into an NFA $M_Q$ [29], and then computes $Q(G)$ by traversing $G$ guided by $M_Q$ [33]. Its time complexity is $O(|V||E|Q|^2 \log^2 |Q|)$.

More specifically, $M_Q = (S, \Sigma, \delta, s_0, F)$, where $S$ is a finite set of states, $\Sigma$ is the alphabet, $\delta$ is the transition function that maps $S \times \Sigma$ to the set of subsets of $S$, $s_0 \in S$ is the initial state, and $F \subseteq S$ is the set of accepting states. There are other methods for constructing NFA, e.g., the one based on partial derivatives [7]. We adopt the algorithm of [29] since it constructs smaller NFA than [7] and takes less time.

After $M_Q$ is in place, the second phase starts, traversing the intersection graph $G_I = (V_I, E_I, I_1)$ of $G$ and $M_Q$ [33]. Here $V_I = V \times S$, $I_1(v, s) = I(v)$, $E_I \subseteq V_I \times V_I$ and $((v, s), (v', s'))$ is in $E_I$ if and only if $(v, v') \in E$ and $s' \in \delta(s, I(v'))$. Each node $v$ in $G$ is marked with a set [v, mark] of markings, where $[v, mark]$ is a set of states $s$ in $S$, indicating that there exists a path $p$ from $u$ to $v$ in $G$ such that $(u, s_0)$ reaches $v$ following the corresponding path $p_t$ of $p$ in $G_I$. When node $v$ is visited in state $s$, only the successor $v'$ of $v$ with $\delta(s, I(v')) = 0$ are inspected. The markings prevent a node from being visited more than once in the same state. It includes $(u, v)$ in $Q(G)$ if $[v, mark] \cap F = \emptyset$, i.e., there exist state $s \in [v, mark]$ and a path $p_t$ from $(u, s_0)$ to $(v, s)$ such that $I_1(p_t) \in L(Q)$.

**Example 4**: Consider an RPQ query $Q = \cdot (b \cdot a + c) \cdot \cdot$ over the graph $G$ of Fig. 2. Its NFA $M_Q$ and a fragment of the intersection graph $G_I$ of $G$ and $M_Q$ are shown in Fig. 4 (excluding dotted edge $((b_2, s_2), (a_1, s_1))$). RPQ_{NFA} traverses $G_I$ and marks the nodes in $G$ with states of $M_Q$. Note that there exist paths from $(c_1, s_0)$ to $(c_2, s_3)$ and from $(c_2, s_0)$ to $(c_2, s_3)$ in $G_I$; thus the accepting state...
Algorithm: IncRPQ

Input: A graph $G$ with $pmark_e()$, regular path query $Q$ and $NFA$ $M_Q$, matches $Q(G)$, and batch updates $(\Delta G^+, \Delta G^-)$.

Output: The updated matches $Q(G \oplus \Delta G)$ and markings $pmark_e()$.

1. set $aff_i := \text{identAff}(G, pmark_e(), \Delta G^-)$; queue $q := nil$
2. for each $(v, u, s)$ in $aff_i$ do
3. update dist, mpre for $v.pmark(u)[s]$ based on its cpre;
4. $q.insert((v, u, s), v.pmark(u)[s])$;
5. for each edge insertion of $(v, w)$ in $\Delta G^+$ do
6. if edge $(v, w)$ leads to a smaller $w.pmark(u)[s].dist$ for node $u$ and state $s$ and $(v, u, s)$ is not in $aff_i$ then
7. update dist, mpre, cpre for $v.pmark(u)[s]$;
8. $q.insert((v, w, u), v.pmark(u)[s])$;
9. update $pmark_e()$ based on queue $q$ and NFA $M_Q$;
10. update $Q(G)$ to get $Q(G \oplus \Delta G)$;
11. return $Q(G \oplus \Delta G)$ and $pmark_e()$;

Figure 5: Algorithm IncRPQ

$s_3$ is included in markings $c_2.pmark(c_1)$ and $c_2.pmark(c_2)$. Therefore, $(c_1, c_2)$ and $(c_2, c_2)$ are returned by $RPQ_{NFA}$.

 Auxiliary structures. The marking $v.pmark(u)$ is of the form (state, dist, cpre, mpre), where (a) dist is the shortest distance from $(u, s_0)$ to $(v, state)$ in $G_1$, (b) $(v', s')$ is contained in $v.pmark(u)[s].cpre$ if there exists an entry in $v'.pmark(u)$ for state $s'$ such that $s \in \delta(s', l(v))$ and $(v', v)$ is in $G$, i.e., $v.pmark(u)[s].cpre$ stores predecessors of node $(v, s)$ in $G_1$ that are on a path starting from $(u, s_0)$; and (c) $(v', s')$ is in $v.pmark(u)[s].mpre$ if $v'.pmark(u)[s].dist + 1 = v.pmark(u)[s].dist$, i.e., mpre keeps track of those predecessors on shortest paths. The auxiliary information is computed by $RPQ_{NFA}$ without increasing its complexity.

 Characterization of AFF. We identify AFF, i.e., the difference between $G(RPQ_{NFA}, Q)$ and $(G \oplus \Delta G)(RPQ_{NFA}, Q)$, as changes to the markings. Indeed, the markings are the data that $RPQ_{NFA}$ necessarily inspects, since updates to markings trigger different behaviors of $RPQ_{NFA}$ when computing $Q(G \oplus \Delta G)$ and $Q(G)$. For instance, a change to dist in $v.pmark(u)[s]$ indicates that $(v, s)$ is reached in BFS through a different path from $(u, s_0)$ and state $s$ is included in $v.pmark(u)$ in $RPQ_{NFA}$ at a different level of the BFS tree.

 Incremental algorithm. Based on markings, we develop incremental algorithms that are bounded relative to $RPQ_{NFA}$. The boundedness is accomplished by updating markings only when there exists a corresponding difference between the data inspected by $RPQ_{NFA}$. For unit edge deletions and insertions, the algorithms are similar to their counterparts for KWS (Section 4.2), guided by changes to dist. Below we just present an algorithm for processing batch updates.

The algorithm is denoted as IncRPQ and shown in Fig. 5. It first invokes procedure $\text{identAff}$ (not shown) to identify a set $aff_i$ of $(v, u, s)$ triples, where $v.pmark(u)[s].dist$ is no longer valid due to edge deletions (line 1). Similar to how IncKWS identifies affected entries of keyword-distance lists (Section 4.2), $\text{identAff}$ checks the values of mpre and cpre in markings. For example, if $v.pmark(u)[s].mpre$ becomes empty, it checks whether $(v, s)$ is in $v.pmark(u)[s'].mpre$ for each successor $v'$ of $v$ and $s' \in \delta(s', l(v'))$. If so, $(v, s)$ is removed, and $\text{identAff}$ continues to check the successors of $v'$. IncRPQ then updates the corresponding (potential) dist values of triple in $aff_i$ based on the current cpre, i.e., the remaining candidate predecessors after removing affected entries. These triples with dist values are inserted into priority queue $q$ (lines 2-4) for deciding exact markings later on.

Thereafter, IncRPQ processes insertions in $\Delta G^+$ by checking whether they yield smaller dist values in some markings (lines 5-6), and update them accordingly (line 7). Again, the updated triples are added to queue $q$ (line 8). IncRPQ determines exact markings based on queue $q$ (line 9) following a monotonically increasing order of updated dist, similar to IncKWS, while NFA $M_Q$ is used to guide the propagation. By grouping updated triples in queue $q$, the algorithm reduces redundant computations when processing $\Delta G$.

Finally, given the updated markings, $Q(G \oplus \Delta G)$ is computed by taking new pairs of nodes marked with accepting states in $F$ and removing invalid ones from $Q(G)$ (line 10).

Example 5: Recall batch updates $\Delta G$ to $G$ from Example 3. These inflict the deletion of $(c_2, s_2), (b_1, s_1)$ and insertion of $(b_2, s_2), (a_1, s_1))$ to the intersection graph $G_1$ of Example 4. IncRPQ first finds that triple $(b_1, c_2, s_2)$ is affected by the deletion. The change is propagated to the descendants of $(b_2, s_2)$ in $G_1$, and potential values of $(\text{dist, mpre})$ for affected entries are computed. After these, it decides exact values after processing insertions; some are shown below.

<table>
<thead>
<tr>
<th>IncRPQ</th>
<th>before updates</th>
<th>after updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2.pmark(c_1)$</td>
<td>$c_2(c_1, s_1)$</td>
<td>$c_2(c_1, s_1)$</td>
</tr>
<tr>
<td>$c_2.pmark(c_2)$</td>
<td>$c_2(c_2, s_1)$</td>
<td>$c_2(c_2, s_1)$</td>
</tr>
<tr>
<td>$c_1.pmark(c_1)$</td>
<td>$c_1(c_1, s_1)$</td>
<td>$c_1(c_1, s_1)$</td>
</tr>
</tbody>
</table>

Note that although the previous path from $(c_2, s_0)$ to $(c_2, s_2)$ is split, accepting state $s_3$ remains in marking $c_2.pmark(c_2)$ since another path connecting these two nodes in $G_1$ is formed as a result of insertions. Indeed, IncRPQ combines the processes for $\text{delete}(c_2, b_1)$ and $\text{insert}(b_2, a_1)$ to compute exact value of $c_2.pmark(c_2)[s_3]$. Based on these, it adds $(c_2, c_1)$ and $(c_1, c_1)$ to obtain $Q(G \oplus \Delta G)$, as accepting state $s_3$ is included in the corresponding markings.

Correctness & complexity. One can verify that IncRPQ correctly updates markings by induction on the number of changed entries. IncRPQ is in $O(|AFF| \log |AFF|)$ time. Indeed, (a) affected triples are added to set $aff_i$, and queue $q$ at most once by BFS traversal; (b) each of procedure $\text{identAff}$ (line 1), computing potential values (lines 2-4) and processing edge insertions (lines 5-8) takes $O(|AFF|)$ time by using $M_Q$ and cpre, where to compute potential values, $O(|AFF|)$ predecessors are processed directly via cpre, without inspecting the entire neighbors; and (c) computing the latest values of markings (line 9) needs $O(|AFF| \log |AFF|)$ time by using heaps for queue $q$, just like fixing dist values for affected nodes in IncKWS (Section 4.2). Note that $|Q|$ is counted in |AFF|. All these steps have costs bounded by a function of |AFF|. Hence IncRPQ is bounded relative to $RPQ_{NFA}$.

5.3 Incrementalization for SCC

We next investigate the incremental problem for SCC. Given a graph $G$, it is to compute $SCC(G)$, i.e., the set of all strongly connected components in $G$. In the sequel we abbreviate a strongly connected component as an SCC.

We incrementalize Tarjan’s algorithm [43] for SCC. We refer to the batch algorithm as Tarjan. Below we first review the basic idea of Tarjan, and identify its AFF.

Batch algorithm. Tarjan traverses a directed graph $G$ via repeated DFS (depth-first search) to generate a spanning forest $\mathcal{F}$, such that each SCC corresponds to a subtree of a
Figure 6: DFS forest of $G$ and contracted graphs

tree $T$ in $F$ with a designated root. It reduces SCC to finding the roots of corresponding subtrees in $F$.

More specifically, each node $v$ in $G$ is assigned a unique integer $v.num$, denoting the order of $v$ visited during the traversal. The edges of $G$ fall into four classes by DFS: (a) tree arcs that lead to nodes not yet discovered during the traversal; (b) fronds that run from descendants to their ancestors in a tree; (c) reverse fronds that are from ancestors to descendants in a tree; and (d) cross-links that run from one subtree to another. In addition, $v.lowlink$ is maintained, representing the smallest $num$ of the node that is in the same SCC as $v$ and is reachable by traversing zero or more tree arcs followed by at most one frond or cross-link. It determines whether $v$ is the root of the subtree corresponding to an SCC by checking whether $v.lowlink = v.num$, and if so, generates the SCC accordingly. It uses a stack to store nodes that have been reached during DFS but have not been placed in an SCC. A node remains on the stack if and only if there exists a path in $G$ from it to some node earlier on the stack.

Example 6: Figure 6 depicts the DFS forest $F$ obtained by applying Tarjan on graph $G$ of Fig. 2. Each node is annotated with its $(num, lowlink)$. There are four SCC's. The corresponding contracted graph $G_c$ (see below) is also shown in Fig. 6 (solid edges), where node $i$ refers to SCC in $G$.

Auxiliary structures. To incrementalize Tarjan, we maintain the values of num and lowlink after traversing $G$, and annotate the edges with the type that they fall into. Besides, a contracted graph $G_c$ is constructed by contracting each SCC into a single node. The graph $G_c$ maintains a counter for the number of cross-links from one node to another. Each node $v$ in $G_c$ has a topological rank $r(v)$, initially the order of the SCC to which $v$ corresponds in the output sequence of Tarjan. Indeed, the topological sorting of SCC's is a byproduct of Tarjan as nodes of each SCC are popped from the stack recursively. These can be obtained by slightly revising Tarjan without increasing its complexity or changing its logic.

It is shown that $r(v) > r(v')$ if $(v, v')$ is a cross-link in $G_c$ [43], an invariant property on which we will capitalize.

Characterization of AFF. The affected area AFF includes the following: (a) changes to lowlink and num of nodes when computing SCC($G \oplus \Delta G$), since accurate lowlink and num values determine the correctness of Tarjan; (b) $v$'s successors for each node $v$ whose lowlink changes, since the lowlink value of $v$ is determined by comparing with lowlink or num of its successors; and (c) the neighbors of $v$ for each node $v$ whose num changes, since these neighbors are affected in this case and are necessarily checked by Tarjan.

We next give bounded incremental algorithms relative to Tarjan, under unit insertions, deletions, and batch updates.

(1) Unit insertions. Inserting an edge may result in combining two or more SCC's into a single one. This happens if

Algorithm: incSCC$^+$

Input: A graph $G$ with num($\cdot$), lowlink($\cdot$), contracted graph $G_c$, SCC($G \oplus \Delta G$) and an edge $(v, w)$ to be inserted.
Output: SCC($G \oplus \Delta G$) and updated num($\cdot$), lowlink($\cdot$) and $G_c$.

1. if $v$ and $w$ are within the same SCC (tree) $T$ then
2. $T := T \oplus \Delta G$; update num($\cdot$), lowlink($\cdot$) for $T$;
3. if $r(scc(v)) > r(scc(w))$ then update $G_c$;
4. if $r(scc(v)) < r(scc(w))$ then
5. $aff :=$ DFS$_G(G_c, w, r(scc(w))); aff :=$ DFS$_G(G_c, v, r(scc(w)));$
6. if Tarjan(aff, r, l) has non-singleton cycle $C$ then
7. merge the corresponding components of nodes in $C$;
8. update num($\cdot$), lowlink($\cdot$) for the new component;
9. else realocRank(aff, r, l);
10. return SCC($G \oplus \Delta G$) and updated num($\cdot$), lowlink($\cdot$), and $G_c$;

Figure 7: Algorithm incSCC$^+$

and only if a cycle is formed with the corresponding nodes of these SCC's in the contracted graph after the insertion.

Employing the contracted graph $G_c$, we propose incremental algorithm incSCC$^+$, shown in Fig. 7, to process unit insertion of edge $(v, w)$. Intuitively, incSCC$^+$ checks whether $(v, w)$ inflicts a cycle in $G_c$, and combines some of the SCC's in SCC($G$) when necessary to get SCC($G \oplus \Delta G$). It separates different types of $(v, w)$, and makes use of topological ranks based on the invariant property mentioned above. Relatively boundedness is guaranteed since every change to the rank of an SCC inspected by algorithm incSCC$^+$ corresponds to a change of lowlink or num, and thus is in AFF.

More specifically, if $v$ and $w$ are within the same SCC $T$, then nothing changes for the other SCC's. In this case, incSCC$^+$ only applies $\Delta G$ to $T$ and computes the changes to num and lowlink, by applying Tarjan on the changed parts (lines 1-2). Otherwise consider the topological ranks of SCC$(v)$ and SCC$(w)$ in $G_c$, where SCC$(v)$ (resp. SCC$(w)$) refers to the corresponding SCC node to which $v$ (resp. $w$) belongs.

(a) If $r(scc(v)) > r(scc(w))$, then no new SCC is generated, and incSCC$^+$ only updates the graph $G_c$ by inserting edge SCC$(v)$, SCC$(w)$ or increasing the counter of edges connecting their corresponding SCC's (line 3). As the order of topological ranks in $G_c$ is not affected in this case, it concludes that SCC$(G_c)$ is still acyclic and SCC($G \oplus \Delta G$) = SCC($G$).

(b) If $r(scc(w)) > r(scc(v))$, i.e., if the order of these two ranks becomes “incorrect”, incSCC$^+$ identifies the affected area aff and aff, two subgraphs of $G_c$ induced by nodes whose ranks are no longer valid, through a bi-directional search. It invokes procedure DFS to conduct a forward DFS traversal from $v$ to find nodes with topological ranks greater than that of $v$, followed by a backward traversal DFS from $v$ to find nodes having ranks less than that of $w$ (lines 4-5). If a cycle $C$ is formed in the affected area, the corresponding SCC's of the nodes in $C$ are merged into one to obtain SCC($G \oplus \Delta G$); this is followed by updating num and lowlink values in the new SCC (lines 6-8). Otherwise, although the output is unaffected, it reallocates the topological ranks of nodes in the affected area such that $r(v) > r(v')$ when $(v, v')$ is in $G_c$, using procedure realocRank (not shown) (line 9), i.e., the relationship of topological ranks still holds. Procedure realocRank sorts the previous ranks of those nodes in aff and aff, and reassigns them in an ascending order, first aff, and then aff. Indeed, nodes in aff, should have lower ranks than those in aff due to the edge insertion.

Example 7: Continuing with Example 6, consider insertion of edge $e_4 = (b_4, b_3)$ into $G$. Observe that the topological
ranks \( r(\text{scc}(b_2)) < r(\text{scc}(b_1)) \) in \( G_c \); thus \( \text{IncSCC}^+ \) identifies the affected area that consists of nodes 1 and 2 and forms a cycle. Then \( \text{scc}_1 \) and \( \text{scc}_2 \) are merged to get the output.

**Correctness & complexity.** The correctness of \( \text{IncSCC}^+ \) is warranted by the following properties: (a) \( \text{scc} \)'s are merged in response to an edge insertion if and only if they form a cycle in the contracted graph; and (b) the topological ranks \( r \) and \( \text{lowlink} \) warranted by the following properties: (a) \( \text{scc} \) the affected area that consists of nodes 1 and 2 and forms a cycle in the contracted graph with updated ranks, and their neighbors. The number of nodes visited does not exceed \( |\text{AFF}| \) since there must be changes to num and \( \text{lowlink} \) in the \( \text{scc} \)'s that they refer to. Cycle detection is done in \( O(|\text{AFF}|) \) time and rank reallocation takes \( O(|\text{AFF}| \log |\text{AFF}|) \) time via sorting by using heaps. Hence \( \text{IncSCC}^+ \) is bounded relative to Tarjan.

(2) **Unit deletions.** When edge \((v, w)\) is deleted from \( G \), an \( \text{scc} \) may be split into multiple ones. However, the output is unchanged if \( v \) still reaches \( w \) after deletion. We give an incremental algorithm for \( \text{SCC} \) under unit deletions, denoted by \( \text{IncSCC}^- \). Intuitively, it examines the reachability from \( v \) to \( w \) by using num and \( \text{lowlink} \) maintained, and computes new \( \text{scc} \)'s in \( \text{SCC}(G \oplus \Delta G) \) when \( v \) no longer reaches \( w \) in the same \( \text{scc} \). The reachability checking is done as a byproduct of change propagation to num and \( \text{lowlink} \), from which relatively boundedness is obtained. For the lack of space, we defer the details of \( \text{IncSCC}^- \) to [2].

(3) **Batch updates.** We now present algorithm \( \text{IncSCC} \) to process \( \Delta G = (\Delta G^+, \Delta G^-) \), provided in [2]. It handles multiple updates in groups instead of one by one, to reduce redundant cost. \( \text{IncSCC} \) consists of two steps.

(a) \( \text{IncSCC} \) first processes **intra-component** updates, where the endpoints of an updated edge are in the same \( \text{scc} \). All updates to the same \( \text{scc} \) are grouped and processed together. It starts with edge insertions, and adjusts values of num and \( \text{lowlink} \) following \( \text{IncSCC}^+ \). Inserted edges are processed following a descending order determined by the num values of their source nodes. Then, following the same processing order, \( \text{IncSCC}^- \) is invoked to handle deletions grouped together, to reduce redundant updates to num and \( \text{lowlink} \) values. After these, Tarjan is called on the affected \( \text{scc} \)'s at most once to generate new \( \text{scc} \)'s in \( \text{SCC}(G \oplus \Delta G) \).

(b) \( \text{IncSCC} \) then handles **inter-component** updates, for edge updates in which the endpoints fall in different \( \text{scc} \)'s. After updating \( G_c \) with deletions, forward and backward traversals are performed to find the affected areas for all inter-component insertions, similar to \( \text{IncSCC}^+ \). However, \( \text{IncSCC} \) stores these areas in a global structure \( \text{aff} \), and checks the existence of cycles formed by nodes from this global affected area, instead of processing unit updates one by one. Components are merged, and num(·) and \( \text{lowlink}(·) \) are revised, along the same lines as \( \text{IncSCC}^+ \) to get \( \text{SCC}(G \oplus \Delta G) \).

Finally, topological ranks are reallocated if needed, and \( \text{SCC}(G \oplus \Delta G) \) is returned (see [2] for details).

**Example 8:** Consider batch updates \( \Delta G \) of Example 3. The intra-component deletions of \( e_9 \) and \( e_5 \) are firstly handled. Since \( e_9 = (e_2, b_3) \) is a reverse frond in \( \text{scc}_2 \), \( \text{IncSCC} \) just deletes it from \( \text{scc}_2 \). Deletion of \( e_5 \) is processed as described in Example 9 (Appendix). Thereafter, the remaining three inter-component insertions in \( \Delta G \) are handled by retrieving the affected area on contracted graph \( G_c' \). Note that nodes 1 to 5 are covered by affected area \( \text{aff} \) that constitutes an \( \text{scc} \) in \( G_c' \), hence all the previous \( \text{scc} \)'s in \( \text{SCC}(G) \) except \( \text{scc}_4 \) are merged to obtain \( \text{SCC}(G \oplus \Delta G) \) in \( \text{IncSCC} \).

**Correctness & complexity.** The correctness of \( \text{IncSCC} \) follows from the correctness of \( \text{IncSCC}^+ \) and \( \text{IncSCC}^- \). \( \text{IncSCC} \) takes \( O(|\text{AFF}|(|\Delta G| + \log |\text{AFF}|)) \) time. Indeed, processing inter-component updates needs \( O(|\Delta G||\text{AFF}|) \) time since each update to the auxiliary structures in \( \text{AFF} \) is checked at most \( |\Delta G| \) times; and handling inter-component updates takes \( O(|\Delta G||\text{AFF}| + |\text{AFF}| \log |\text{AFF}|) \) time, where each node with updated ranks in \( G_c \) is accessed by at most \( |\Delta G| \) different bi-directional searches; the time for final rank reallocation is in \( O(|\text{AFF}| \log |\text{AFF}|) \) as all such nodes are collected in \( \text{aff} \). Thus \( \text{IncSCC} \) is bounded relative to Tarjan.

### 6. EXPERIMENTAL EVALUATION

Using real-life and synthetic data, we conducted three sets of experiments to evaluate the impacts of (1) the size \( |\Delta G| \) of batch updates; (2) the complexity of queries \( Q \) for KWS, RPQ and ISO (see below); and (3) the size \( |G| \) of graphs on our incremental algorithms, compared with their batch counterparts and some existing dynamic algorithms.

**Experimental setting.** We used the following datasets.

**Graphs.** We used two real-life graphs: (a) DBpedia, a knowledge graph [1] with 4.3 million nodes, 40.3 million edges and 495 labels; and (b) LiveJournal (liveJ in short), a social network [3] with 4.9 million nodes, 68.5 million edges and 100 labels. We also designed a generator to produce synthetic graphs \( G \), controlled by the number of nodes \( |V| \) (up to 50 million) and number of edges \( |E| \) (up to 100 million), with labels drawn from an alphabet \( \Sigma \) of 100 symbols.

**Updates \( \Delta G \) are randomly generated for real-life and synthetic data, controlled by size \( |\Delta G| \) and a ratio \( \rho \) of edge insertions to deletions. We use \( \rho = 1 \) unless stated otherwise, i.e., the size of the data graphs \( G \) remain stable.

**Query generators.** We randomly generated 30 queries of \( \text{KWS, RPQ} \) and \( \text{ISO} \), with labels drawn from the graphs. More specifically, (1) \( \text{KWS} \) queries are controlled by the number \( m \) of keywords and bound \( b \); (2) \( \text{RPQ} \) queries are controlled by the size (recall size \( |Q| \) of a regular path query from Section 2.1) and the numbers of occurrences of \( .+ \) and Kleene \( * \); and (3) \( \text{ISO} \) queries are controlled by the number of nodes \( |V_Q| \), the number of edges \( |E_Q| \) and the diameter \( d_Q \), i.e., the length of longest shortest path between any two nodes in \( Q \) when taken as an undirected graph.

**Algorithms.** We implemented the following algorithms, all in Java. (1) Incremental algorithms (a) \( \text{IncKWS} \) (Section 4.2), \( \text{IncRPQ} \) (Section 5.2), \( \text{IncSCC} \) (Section 5.3) and \( \text{IncISO} \) (see Appendix); (b) \( \text{IncKWS}_a, \text{IncRPQ}_a, \text{IncSCC}_a \) and \( \text{IncISO}_a \), which process unit updates in batch \( \Delta G \) one by one by calling their algorithms for unit updates developed in this work; (c) \( \text{DynSCC} \) which combines the incremental algorithm in [26] to process insertions and decremental algorithm in [32] for deletions. (2) Batch algorithms \( \text{BLINKS} \) [27] for \( \text{KWS, RPQ}\text{nfa} \) for \( \text{RPQ} \), Tarjan for \( \text{SCC} \), and \( \text{VF2} \) [15] for \( \text{ISO} \).

We did the experiments on an Amazon EC2 r3.4xlarge instance, powered by Intel Xeon processor with 2.3GHz, with
Experimental results. We next report our findings.

Exp-1: Impact of $|\Delta G|$. We first evaluated the impact of $|\Delta G|$ on the performance of IncKWS, IncRPQ, IncSCC and IncISO, compared with (a) their batch counterparts, and (b) incremental IncKWS, IncRPQ, IncSCC, and IncISO, and DynSCC for SCC. We conducted the experiments (a) on real-life graphs by varying $|\Delta G|$ from 2.2M to 17.6M in 2.2M increments over DBpedia and from 3.7M to 29.6M in 4M increments over liveJ, which account for 5% to 40% of each graph; and (b) synthetic $G$ with $|G| = 50M, 100M$ by varying $|\Delta G|$ from 7.5M to 60M in 7.5M increments, i.e., 5% to 40% of $|G|$, for SCC; the results for KWS, RPQ and ISO on synthetic graphs are consistent with their counterparts on real-life graphs, and hence are not reported here.

(1) KWS. Fixing $m = 3$ and $b = 2$, we report the performance of IncKWS on DBpedia and liveJ in Figures 8(a) and 8(e), respectively. We find the following. (a) IncKWS outperforms BLINKS from 6.3 times to 2.8 times over DBpedia, and from 7.3 times to 2 times over liveJ, when $|\Delta G|$ varies from 5% to 20% of $|G|$. In fact, IncKWS does better than BLINKS when $|\Delta G|$ is up to 35% and 30% of $|G|$, respectively. These verify the effectiveness of localizable incremental algorithm IncKWS. (b) IncKWS is from 1.6 to 2 and 1.3 to 1.7 times faster than IncKWS, in the same setting. This validates the effectiveness of our optimization strategies on batch updates. (c) The larger $|\Delta G|$ is, the slower IncKWS and IncKWS n are, as expected. However, when $|\Delta G|$ increases, the gap between the performance of IncKWS and IncKWS n gets larger, which is more evident on liveJ. That is, IncKWS scales better with $|\Delta G|$. In contrast, BLINKS is indifferent to $|\Delta G|$. (d) IncKWS is efficient: it takes 12 and 32 seconds over DBpedia and liveJ, respectively, when $|\Delta G|$ is 10% of $|G|$, as opposed to 61 and 146 seconds by BLINKS. (e) The ratio $\rho$ of insertions to deletions in $\Delta G$ has no impact on the performance of IncKWS, by varying $\rho$ while keeping $|\Delta G|$ unchanged (not shown).

(2) RPQ. We then evaluated the relatively bounded algorithm IncRPQ. Fixing $|Q| = 4$, Figures 8(b) and 8(f) show that (a) IncRPQ is from 8.6 to 3.2 times faster than RPQ on DBpedia, and from 12.7 to 4.1 times faster on liveJ, when $|\Delta G|$ varies from 5% to 20% of $|G|$. (b) IncRPQ consistently does better than IncRPQ n. The improvement is on average 2.3 times when $|\Delta G|$ is about 15% of $|G|$. (c) IncRPQ scales better with $|\Delta G|$ than IncRPQ n, especially when $|\Delta G|$ is large. (d) IncRPQ is insensitive to $\rho$.

(3) SCC. Figures 8(c), 8(g) and 8(i) report the performance for SCC over DBpedia, liveJ and synthetic graphs, respec-

Figure 8: Performance evaluation


We find the following. (a) IncSCC is from 8 to 1.5, 2.3 to 1.2, and 7.7 to 1.7 times faster than Tarjan over DBpedia, liveJ and synthetic graphs, respectively, when $|\Delta G|$ varies from 5% to 25% of $|G|$. These verify the effectiveness of incrementalizing batch algorithm Tarjan. It is from 1.7 to 2.6, 1.9 to 2.1, and 1.4 to 2.2 times faster than IncSCC, in the same setting. (b) IncSCC performs better than DynSCC. For instance, IncSCC is on average 2.1 times faster than DynSCC when $|\Delta G|$ varies from 5% to 15% of $|G|$ over synthetic graphs. In particular, DynSCC does not do well with small $|\Delta G|$ due to its additional cost for maintaining dynamic data structures even when the output remains stable. (c) IncSCC works better on DBpedia than on liveJ since there are large scc's in liveJ, which take up to 77% of $|G|$, and need to be split in response to $\Delta G$. (d) IncSCC is insensitive to $\rho$, similar to IncKWS and IncRPQ.

(4) ISO. Fixing $|Q| = (4, 6, 2)$, i.e., pattern queries with 4 nodes, 6 edges and diameter 2, we evaluated localizable IncISO. As shown in Figures 8(d) and 8(h) on DBpedia and liveJ, respectively, (a) IncISO behaves better than VF2 and IncISOo, when $|\Delta G|$ is no more than 25% of $|G|$; it is from 5.6 to 1.8 times faster than VF2 and from 2.4 to 2.6 times faster than IncISOo, respectively, for $|\Delta G|$ from 5% to 25% of $|G|$. (b) The gap between the performance of IncISO and IncISOo gets larger when $|\Delta G|$ grows. (c) IncISO and IncISOo take longer to process edge insertions than deletions for the same $|\Delta G|$. This is because matches to be removed can be directly identified and hence, IncISO is faster for deletions. We also find that IncISO is insensitive to $\rho$.

(5) Unit updates. Using the same set of queries, we also evaluated the performance of these algorithms on processing unit updates, which consist of either a unit insertion or a unit edge deletion. As expected, the improvements of incremental algorithms are substantial. More specifically, IncKWS, IncRPQ, IncSCC and IncISO outperform their batch counterparts by 89 times, 221 times, 37 times, and 393 times on average, respectively (not shown). Moreover, IncSCC is 5.7 times faster than DynSCC on average.

Exp-2: Query complexity. We next evaluated the impact of queries $Q$, by varying different parameters of $Q$. We focused on KWS, RPQ and ISO, as SCC has a constant query. We fixed $|\Delta G| = 4.4M$, i.e., 10% of $|G|$, and used DBpedia.

(1) KWS. We varied $(m, b)$ from (2, 1) to (6, 5) for KWS queries. As shown in Figure 8(j), (a) the larger $(m, b)$ is, the longer time is taken by all the algorithms, as expected. (b) IncKWS performs well on real-life queries. For queries with 4 keywords and bound 3, it takes 17 seconds over DBpedia, as opposed to 44 seconds by BLINKS. It works better on sparse DBpedia than on liveJ (not shown). (c) IncKWS outperforms the other algorithms, consistent with Fig. 8(a).

(2) RPQ. Varying $|Q|$ from 3 to 7, the results in Fig. 8(k) tell us the following. (a) IncRPQ is efficient: it returns answers within 190 seconds for all the queries, as opposed to 1080 seconds by RPQ$_{108s}$ and 326 seconds by IncRPQ$_d$. (b) The occurrences of Kleene * have little impact on all the algorithms, as the size of NFA $M_2$ only depends on the number of node labels in $Q$. (c) IncRPQ outperforms RPQ$_{108s}$ and IncRPQ$_d$ on all the queries; this is consistent with Fig. 8(b).

(3) ISO. Varying $|Q| = (|V_Q|, |E_Q|, \sigma_Q)$ from (3, 5, 1) to (7, 9, 5), we evaluated the impact of pattern queries. Figure 8(l) shows that all algorithms take longer over larger $|Q|$, as expected. However, (a) IncISO outperforms VF2 and IncISOo in all the cases, for the same reasons given above. (b) IncISO does well: it takes 290 seconds when $|Q| = (5, 7, 3)$, but VF2 and IncISOo take 1160 and 570 seconds, respectively.

Exp-3: Impact of $|G|$. We finally evaluated the impact of $|G|$ using synthetic graphs. Fixing $|\Delta G| = 15M$ and using the same set of queries tested in Exp-1, we varied $|G|$ with scale factors from 0.2 to 1. Figures 8(m), 8(n), 8(o) and 8(p) report the performance for KWS, RPQ, SCC and ISO, respectively. Observe the following. (a) All the incremental algorithms are less sensitive to $|G|$ compared with their batch counterparts. (b) Incremental algorithms scales well with $|G|$ and are feasible on large graphs.

Summary. From the experiments we find the following. (1) Incremental algorithms, either localizable or relatively bounded, are more effective than their batch counterparts in response to updates. When $|\Delta G|$ varies from 5% to 20% of $|G|$ for the three full-size graphs $G$, IncKWS, IncRPQ, IncSCC and IncISO outperform BLINKS, RPQ$_{108s}$, Tarjan and VF2 from 6.9 to 2.4 times, 11.6 to 2.8 times, 3.4 to 1.7 times, and 7.9 to 2 times on average, respectively. They outperform the batch algorithms even when $|\Delta G|$ is up to 30%, 35%, 25% and 25% of $|G|$, respectively. (2) Incremental algorithms scale well with $|G|$ and are feasible on real-life graphs when $|\Delta G|$ is small, as commonly found in practice. For instance, IncKWS, IncRPQ, IncSCC and IncISO take 9, 42, 7 and 113 seconds, respectively, when updates account for 5% of DBpedia, as opposed to 62, 355, 54 and 427 seconds by their batch counterparts. (3) Our optimization strategies for batch updates effectively improve the performance by 1.6 times on average.

7. CONCLUSION

We have established undoable and doable results for incremental graph computations. We have shown that the incremental problems for RPQ, SCC and KWS are unbounded under unit updates. However, we have proposed alternative characterizations for the effectiveness of incremental graph computations, and shown that RPQ, SCC, KWS and ISO are either localizable or bounded relative to their batch counterparts, by providing incremental algorithms with corresponding performance guarantees. Our experimental results have verified that the incremental algorithms substantially outperform their batch counterparts and scale well with large graphs, justifying the effectiveness of the new standards.

One topic for future work is to classify graph queries commonly used in practice, characterize their incremental computations, and identify performance guarantees for their incremental algorithms when possible. Another topic is to identify practical conditions under which unbounded incremental problems become bounded or relatively bounded.

Acknowledgements. Fan and Tian are supported in part by ERC 652976, 973 Program 2014CB340902, NSFC 61133002 and 61421003, EPSRC EP/M02528/1, Shenzhen Peacock Program 110510003084361, Guangdong Innovative Research Team Program 2011D005, the Foundation for Innovative Research Groups of NSFC, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Tian is also supported in part by NSFC 61602023.
8. REFERENCES

functions put and output updates of the two instances are mapped by $I$ is unbounded under a unit edge deletion [38].

Given delete($v_i, v_j$) in $∆G_1$, function $f_i$ returns corresponding ($v'_i, v'_j$) to be deleted from $G_2$, i.e., $∆G_2 = f_i(∆G_1)$. Then the changes $∆O_2$ to $Q_2(G_2)$ consist of node pairs ($v'_i, v'_j$) removed. Clearly, $v'_i$ is no longer reachable from $v_i$ in $G_2$ and $v_j$ is not reachable from $v_j$ in $G_1$; hence $∆O_1$ is the set of such $r(v_i)$ changed from true to false, which can be computed by $f_o(∆O_2)$ directly. Thus, a one-to-one mapping between the changes of $I_1$ and $I_2$ is obtained via linear-time functions $f_i$ and $f_o$.

Putting these together, $(f_i, f_o)$ is a $∆$-reduction and RPQ is unbounded under a unit edge deletion by Lemma 2.

(2) Insertions. We next show that RPQ is unbounded under a unit edge insertion by contradiction. Consider graph $G$ shown in Fig. 9 (excluding dotted edges), which consists of two cycles $(v_1, v_2), \ldots, (v_{2n-1}, v_{2n}), (v_{2n}, v_1)$ and $(u_1, u_2), \ldots, (u_{2n-1}, u_{2n}), (u_{2n}, u_1)$, and an edge $(v_1, w)$. Each node $v_i$ in $G$ has label $α_i$ for $i \in [1, 2n]$, while $u_i$ is labeled $α_2$. Node $w$ is labeled $α_3$ that is distinct from $α_1$ and $α_2$. Query $Q$ is defined as $α_1(α_1\ast α_2\ast α_3)$. Denote by $∆_1$ the insertion of $(v_1, v_1)$, and by $∆_2$ the insertion of $(u_1, v_1)$. Let graph $G_1 = G \oplus ∆_1$, $G_2 = G \oplus ∆_2$, and $G_3 = G \oplus ∆_2$. One can verify that $Q(G_1) = Q(G_2) = \emptyset$, while $Q(G_3) = \{(v_i, w) | i \in [1, 2n]\}$.

Assume by contradiction that there exists a bounded incremental algorithm $T_o$ for RPQ under a unit edge insertion. Then $T_o(G, Q, Q(G), ∆_1)$ and $T_o(G, Q, Q(G), ∆_2)$ are both in $O(1)$ time since only a unit update is applied to $G$ and none of the outputs is affected for the fixed query $Q$. We next show that this leads to contradiction.

Let $T_o(G, ∆G)$ denote the sequence of nodes visited in executing $T_o(G, Q, Q(G), ∆G)$, referred to as its trace. Observe that $T_o(G, Q, Q(G), ∆_2)$ and $T_o(G_1, Q, Q(G_1), ∆_2)$ must behave differently as the outputs of these two are different, in which $T_o(G_1, Q, Q(G_1), ∆_2)$ computes $Q(G_2)$ exactly. This can happen only if $T_o(G, ∆_2)$ and $T_o(G_1, ∆_2)$ contain some node associated with different information in $G$ and $G_1$ as $T_o$ traverses the graph from the nodes involved in $∆_2$, i.e., $u_i$ or $v_i$. Since $G_1$ is obtained by applying $∆_1$ to $G$, these nodes must be included in $T_o(G, ∆_1)$ with information updated. Observe that if a node $v$ in $G$ is visited during the execution of a locally persistent algorithm $T_o$ to process $∆G$, then each node on some undirected path from the position of $∆G$ to $v$ is also inspected by $T_o$. Denote by $v_o$ the first node having different information in $T_o(G, ∆_2)$ and in $T_o(G_1, ∆_2)$. Then $T_o(G, ∆_1)$ and $T_o(G, ∆_2)$ include all the nodes on an undirected path from the position of $∆_1$ to that of $∆_2$ through $v_o$. However, the length of this path is $O(n)$, which contradicts the assumption that $T_o(G, Q, Q(G), ∆_1)$ and $T_o(G, Q, Q(G), ∆_2)$ both take constant time.

**Appendix: Proofs and Algorithms**

**Proof of Lemma 2**

Assume that there exists a bounded incremental algorithm $T_o$ for $Q_2$. We show that a bounded incremental algorithm $T_o$ for $Q_1$ can be built from $T_o$ and the $Δ$-reduction $(f, f_o)$ from $Q_1$ to $Q_2$. Given an instance $I_1 = (Q_1, G_1)$ of $Q_1$, we first compute a corresponding instance $f(I_1) = (Q_2, G_2)$ of $Q_2$. Then for each update $∆G_1$ to $G_1$, $T_o$ transforms it to $f(∆G_1)$ and invokes the bounded incremental algorithm $T_o$ on $G_2$. $Q_2$ and $f(∆G_1)$ to obtain $∆O_2$, i.e., the corresponding changes to $Q_2(G_2)$. Thereafter, it transforms the updates $∆O_2$ back to $Q_2$ leveraging function $f_o$. As $(f, f_o)$ is a $Δ$-reduction, it concludes that $f_o(∆O_2) = ∆O_1$, where $ΔO_1$ denotes the updates to $Q_1(G_1)$ in response to $∆G_1$ and $T_o$ takes $\text{PTIME in } |ΔG_1| + |ΔO_1|$ and $|Q_1|$ to compute $ΔO_1$, i.e., $T_o$ is a bounded incremental algorithm for $Q_1$. From this Lemma 2 follows.

**Proof of Theorem 1**

We give a proof for RPQ, and defer the proofs for SCC and KWS to [2] due to the lack of space.

**RPQ**. We consider first updates consisting of a unit edge deletion, and then the case of a unit edge insertion.

(1) Deletions. We prove the unboundedness of the incremental problem for RPQ under a unit edge deletion by $Δ$-reduction from the single source reachability problem to all vertices (SSRP). Given a graph $G = (V, E, l)$ and a node $v_s \in V$, SSRP is to decide whether node $v_i$ is reachable from $v_s$ for all $v_i \in V$. The answer is expressed as Boolean value $r(v_i)$ associated with $v_i$. The incremental problem for SSRP is unbounded under a unit edge deletion [38].

Given an instance $I_1$ of SSRP, i.e., a graph $G_1 = (V_1, E_1, l_1)$ and a distinguished node $v_s \in G_1$, we construct an instance $I_2$ of RPQ, i.e., a graph $G_2 = (V_2, E_2, l_2)$ and a regular path query $Q_2$, by using function $f$ such that the reachability $r(v_i)$ from $v_s$ to $v_i$ in $G_1$ changes in response to $∆G_1$ iff (if and only if) there exists a corresponding change in the output of $Q_2$ on $G_2$ in response to $∆G_2$, where input and output updates of the two instances are mapped by functions $f_i$ and $f_o$, respectively (see Section 3). More specifically, $G_2$ is constructed from $G_1$ with each node $v_i$ replaced by $v'_i$. All the edges in $G_1$ remain unchanged, i.e., $(v'_i, v'_j) \in E_2$ iff $(v_i, v_j) \in E_1$. Furthermore, $l_2(v'_i) = α_1$ when $v'_i = v_i$, and $l_2(v'_i) = α_2$ otherwise, where $v'_i$ corresponds to source node $v_i$ in $G_1$. Query $Q_2$ is defined as $α_1(α_2)$

Given delete($v_i, v_j$) in $∆G_1$, function $f_i$ returns corresponding ($v'_i, v'_j$) to be deleted from $G_2$, i.e., $∆G_2 = f_i(∆G_1)$. Then the changes $∆O_2$ to $Q_2(G_2)$ consist of node pairs ($v'_i, v'_j$) removed. Clearly, $v'_i$ is no longer reachable from $v'_i$ in $G_2$ and $v_j$ is not reachable from $v_s$ in $G_1$; hence $∆O_1$ is the set of such $r(v_i)$ changed from true to false, which can be computed by $f_o(∆O_2)$ directly. Thus, a one-to-one mapping between the changes of $I_1$ and $I_2$ is obtained via linear-time functions $f_i$ and $f_o$.

Putting these together, $(f_i, f_o)$ is a $Δ$-reduction and RPQ is unbounded under a unit edge deletion by Lemma 2.
subgraphs of \( G \) that are isomorphic to \( Q \). Observe that the deletion of an edge \( e \) may cause the removal of matches that include \( e \) from \( Q(G) \). Conversely, insertion of \( e = (v, w) \) may add new matches to \( Q(G) \) and all these matches are within \( G_{d_Q}(v) \) and \( G_{d_Q}(w) \), where \( d_Q \) is the length of the longest shortest path between any two nodes in \( Q \) when taken as undirected graph, i.e., the diameter of \( Q \).

Based on this, we outline a localizable incremental algorithm, denoted by IncISO, for ISO under batch updates (not shown). It works as follows. (1) Collect the set \( \Delta G^- \) of all edge deletions in \( \Delta G \). For each edge deletion of \( e \), remove those matches including \( e \) from \( Q(G) \), by inspecting the \( d_Q \)-neighbors of the two nodes on \( e \), where \( d_Q \) is the diameter of \( Q \). (2) For the rest of updates in \( \Delta G \), i.e., edge insertions \( \Delta G^+ \), extract the union of \( d_Q \)-neighbors of the nodes involved in these edge insertions, denoted by \( G_{d_Q}(\Delta G^+) \). (3) Invoke an existing batch algorithm (e.g., VF2 [15]) for ISO to compute \( Q(G_{d_Q}(\Delta G^+)) \) all together rather than one by one, and add those matches to \( Q(G) \) that are not in \( Q(G) \).

Obviously, the cost of IncISO can be expressed as a function of \( |Q| \) and \( |G_{d_Q}(\Delta G)| \), instead of the size \( |G| \) of the entire graph \( G \). In other words, IncISO is localizable, and hence so is ISO. Note that \( G_{d_Q}(\Delta G) \) also includes the \( d_Q \)-neighbors of nodes involved in edge deletions.

Putting this together with the algorithms presented in Sections 4.2, we complete the proof of Theorem 3.

In our experimental study, we compare IncISO with another algorithm IncISO_s, which applies the batch algorithm on \( d_Q \)-neighbor of each update one by one.

**Incrementalization for SCC (Section 5.3)**

*Example 9:* Consider deleting edge \( e_5 = (c_1, a_1) \) from \( G \) of Fig. 2, which is a frond in scc_3 (see Example 6). Since the lowlink value of \( c_1 \) increases to 3 and equals its num after deletion, procedure chkReach concludes that \( c_1 \) no longer reaches root \( a_1 \) of scc_3. In light of this, IncSCC computes new scc’s on affected scc_3 to update the output, i.e., scc_3 is split into three components. The contracted graph \( G' \) after the deletion is also shown in Fig. 6 (solid edges). \( \square \)