Lazy Graph Processing in Haskell

Philip Dexter Yu David Liu Kenneth Chiu

Department of Computer Science
SUNY Binghamton
Binghamton NY 13902, USA
{pdexter1,davidl,kchiu}@binghamton.edu

Abstract

This paper presents a Haskell library for graph processing: DELTAGRAPH. One unique feature of this system is that intentions to perform graph updates can be memoized *in-graph* in a decentralized fashion, and the propagation of these intentions within the graph can be decoupled from the realization of the updates. As a result, DELTAGRAPH can respond to updates in constant time and work elegantly with parallelism support. We build a Twitter-like application on top of DELTAGRAPH to demonstrate its effectiveness and explore parallelism and opportunistic computing optimizations.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; E.1 [*Data Structures*]: Graphs and networks

Keywords Functional Data Structures, Graph Programming

1. Introduction

In this paper we take on the problem of graph processing in Haskell. From social networks to gene regulatory networks, graphs are often at the epicenter of modern computing. Many real-world graph applications are update-intensive and highly dynamic [11, 24]. For example, during a breaking news event, a graph of information dissemination via Twitter might change very quickly.

We introduce DELTAGRAPH to provide language support for efficient graph processing with intensive updates. Unlike existing graph support where an update to graph node, edge, or topology is realized immediately—a strategy we henceforth call *eager graphs*—each *intention* to perform an update in DELTAGRAPH is memoized in-graph in an order-preserving manner and gradually propagated through the topological graph structure. Compared with eager graphs, DELTAGRAPH has the following attractive traits:

- $O(1)$ **Reactiveness to Updates**: Potentially expensive operations such as changes in topology or whole-graph mapping—*e.g.*, reformatting the birth date of all users in Twitter—do not require an immediate traversal of the graph. They can be as cheap as an $O(1)$ operation to memoize the update intention.
- $O(1)$ **Local Propagation Steps**: In DELTAGRAPH, each update propagation step is *local*: it only involves a graph node and its immediate neighbors in the topological structure. This feature is useful for long-running systems with non-uniform workloads—propagation can happen when workloads are low. This is aligned in philosophy with opportunistic computing [8] and cycle stealing [22].
- **Pre-natal optimization**: In update-intensive systems, a recently updated node is often updated again. DELTAGRAPH optimizes for this case by removing update intentions that would "counteract" each other *before* such intentions are realized. It can further compose multiple related update intentions into one.
- **Parallelism Friendliness**: Two updates can be realized in parallel when they do not target nodes in the same neighborhood in the graph topological structure. Under the same condition, parallelism can be achieved between two update propagations as well as one update realization and one update propagation.

DELTAGRAPH is implemented as an open-source Haskell library. Our evaluation is built over a real-world scale-free Twitter graph with 1 million nodes and 5 million edges. Our experiments explore different forms of in-graph lookups (single key lookups vs. clique lookups), different ratios of updates and lookups, and different assumptions of data correlation among updates. We further demonstrate the impact of combining DELTAGRAPH with opportunistic computing under realistic assumptions of arrival distribution and the impact of combining DELTAGRAPH with parallelism. Our experiments show DELTAGRAPH can achieve a speedup of 50% over eager graphs in update-intensive settings, and a further speedup of 30% when utilizing parallelization for propagation.

Design Challenges Graph support in pure functional languages is an active research area [21, 13, 12, 15, 18, 19, 25, 17, 6], but explicit support for graph updates is under explored. The design of our system has faced some unique challenges.

The first challenge is to come up with a graph representation which allows for efficient propagation and realization. In functional languages, two graph representations are commonly used: (1) *node/edge graphs*: the canonical graph-theoretic representation where a graph is a set of nodes along with a set of edges connecting them; (2) *inductive graphs* (*e.g.*, [12, 25]): graph overlays on inductive trees. DELTAGRAPH is based on inductive graphs to allow the propagation of updates to follow the inductive structure as an efficient tree traversal. In contrast, propagation in node/edge graphs would require a lookup for every step of propagation, which leads to inefficiencies. Inductive graphs introduce their own share of subtlety. For example, when a node in the inductive graph is deleted, all of its child subtrees become detached from the rest of the inductive structure. Reattaching these detached subtrees in a fashion consistent with lazy propagation is a design issue of DELTAGRAPH.

The second challenge is to preserve the chronological order of graph updates. For example, imagine a graph is first applied with an update to add 2 to each node value (a whole-graph mapping up-

$$
\begin{array}{lll}
e & ::= \texttt{x} \mid f \mid e\ e & \textit{expression} \\
& \mid \textbf{atom}\ \texttt{nk}\ e & \\
& \mid \textbf{insertNode}\ \texttt{nk}\ e\ e & \\
& \mid \textbf{insertEdge}\ \texttt{ek}\ \texttt{nk}\ \texttt{nk}\ e & \\
& \mid \textbf{deleteNode}\ \texttt{nk}\ e & \\
& \mid \textbf{deleteEdge}\ \texttt{ek}\ e & \\
& \mid \textbf{merge}\ e\ \texttt{nk}\ \texttt{ek}\ e & \\
& \mid \textbf{mapNodes}\ e\ e & \\
& \mid \textbf{foldNodes}\ e\ e\ e & \\
& \mid \textbf{lookupNode}\ \texttt{nk}\ e & \\
& \mid \textbf{lookupEdge}\ \texttt{ek}\ e & \\
f & ::= \lambda \texttt{x}.e & \textit{function} \\
\texttt{nk} & & \textit{node key} \\
\texttt{ek} & & \textit{edge key} \\
\texttt{x}, \texttt{y} & & \textit{variable} \\
\delta & ::= \textbf{InsertNode}\ \texttt{nk}\ e & \textit{update intention} \\
& \mid \textbf{InsertEdge}\ \texttt{ek}\ \texttt{nk}\ \texttt{nk} & \\
& \mid \textbf{DeleteNode}\ \texttt{nk} & \\
& \mid \textbf{DeleteEdge}\ \texttt{ek} & \\
& \mid \textbf{MapNodes}\ e & \\
& \mid \textbf{Merge}\ \Gamma\ \texttt{nk}\ \texttt{ek} &
\end{array}
$$

Figure 1: DELTAGRAPH: Expressions and Values

date), and is then subsequently applied with another update to multiply 3 to each node value. This results in a graph different from one produced by reversing the order of the two updates. DELTAGRAPH is carefully designed to maintain the order of these update intentions. Order preservation is particularly challenging when combined with propagation, where update intentions are memoized in a decentralized fashion. In a nutshell, the chronological order must be preserved across all propagation paths.

This paper makes the following contributions:

- A Haskell graph processing system with rapid support for updates and efficient in-graph optimizations.
- The development of a MINITWITTER application to evaluate the feasibility of DELTAGRAPH given different settings of application characteristics, parallelization, and opportunistic computing.

2. DELTAGRAPH Programming Model

The programming interface of DELTAGRAPH is simple: it provides primitives to support elementary graph operations just as other graph systems do. Indeed, the most unique aspect of DELTAGRAPH lies upon how the graph operations are implemented, not the interface itself. This gives DELTAGRAPH seamless interoperability with existing systems that support similar operations.

2.1 Concrete Syntax

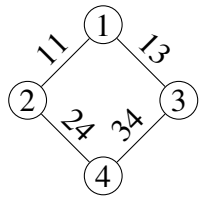
We define the syntax of DELTAGRAPH in Figure 1. Graph nodes and edges are uniquely identified through *node keys* (`nk`) and *edge keys* (`ek`). In DELTAGRAPH, graphs are first-class values. They are either atomic graphs or graphs constructed through applying operations on atomic graphs. The **atom** `nk` *e* expression creates a singleton graph where `nk` is the node key and *e* is the node payload. The graph operations we support are

- **insertNode** `nk` *e* e', insert a node with key `nk` and payload *e* into graph e'
- **insertEdge** `ek` `nk` `nk`$'$ *e*, insert an edge with key `ek` and two attaching nodes `nk` and `nk`$'$ into graph *e*
- **deleteNode** `nk` *e*, delete the node with key `nk` from graph *e*
- **deleteEdge** `ek` *e*, delete the edge with key `ek` from graph *e*
- **merge** *e* `nk` `ek` e', merge graphs *e* and e' through a new inductive edge with key `ek`; start the merge at the node with key `nk` in graph *e*
- **mapNodes** *e* e', map all nodes in graph e' with mapping function *e*
- **foldNodes** *e* e' e'', perform a fold over the graph e'' with *e* as the folding function and e' as the base folding value
- **lookupNode** `nk` *e*, perform a lookup for the node with key `nk` in graph *e*
- **lookupEdge** `ek` *e*, perform a lookup for the edge with key `ek` in graph *e*

Figure 1 is a faithful specification of our library API, except for two simplifications. First, here we only consider undirected edges and thus treat the two insertion operations **insertEdge** `ek` `nk` $\texttt{nk}_0$ and **insertEdge** `ek` $\texttt{nk}_0$ `nk` as congruent. Second, we omit edge payload from the abstract syntax. As a result, we do not include the grammar for the expressions **mapEdges** and **foldEdges**.

Example 1 (Graph construction)**.** In DELTAGRAPH, graphs are constructed through a sequence of node and edge insertions. For instance, the following is a source program in DELTAGRAPH which constructs the graph illustrated to the right of it. For illustration purposes, we liberally use integers for node/edge keys and node payloads, typeset as **node keys**, *edge keys*, and payloads.

$$
\begin{array}{ll}
\texttt{let} & x_1 = \textbf{atom}\ \mathbf{1}\ 10 \\
& x_2 = \textbf{insertNode}\ \mathbf{2}\ 20\ x_1 \\
& x_3 = \textbf{insertNode}\ \mathbf{3}\ 30\ x_2 \\
& x_4 = \textbf{insertNode}\ \mathbf{4}\ 40\ x_3 \\
& x_5 = \textbf{insertEdge}\ \textit{12}\ \mathbf{1}\ \mathbf{2}\ x_4 \\
& x_6 = \textbf{insertEdge}\ \textit{13}\ \mathbf{1}\ \mathbf{3}\ x_5 \\
& x_7 = \textbf{insertEdge}\ \textit{34}\ \mathbf{3}\ \mathbf{4}\ x_6 \\
\texttt{in} & \textbf{insertEdge}\ \textit{24}\ \mathbf{2}\ \mathbf{4}\ x_7
\end{array}
$$



Example 2 (Payload update)**.** Node payloads are updated through the **mapNodes** operation:

$$
\begin{array}{ll}
\texttt{let}\ x_1 = \ldots & \\
\texttt{in}\ \ \textbf{mapNodes}\ (\lambda n\ e \rightarrow & \texttt{if}\ (n > \mathbf{2}) \wedge (e > 1) \\
& \texttt{then}\ (e + 400) \\
& \texttt{else}\ e)\ x_1
\end{array}
$$

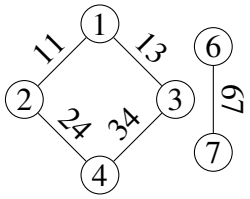
This expression applies a payload increase of 400 to every node in the graph x_1 with a key greater than **2** and a payload greater than 1. As this example suggests, DELTAGRAPH resorts to graph mapping for payload updates.

$$
\begin{array}{rll}
\Gamma & ::= & \langle \overline{\texttt{ek} \mapsto \texttt{nk}}; e; \Pi; \hat{\Pi}; \Delta; \hat{\Delta} \rangle & \textit{delta graph} \\
\Pi & ::= & \overline{\texttt{ek} \mapsto \Gamma} & \textit{inductive edges} \\
\hat{\Pi} & ::= & \overline{\texttt{ek} \mapsto \texttt{nk}} & \textit{back/cross edges} \\
\Delta & ::= & \overline{\delta} & \textit{delta list} \\
\hat{\Delta} & ::= & \overline{\Gamma} & \textit{disconnected components}
\end{array}
$$

Figure 2: DELTAGRAPH: Run-time Values

Example 3 (Forest)**.** DELTAGRAPH can support forests:

$$
\begin{array}{ll}
\texttt{let} & x_1 = \mathbf{atom}\ \mathbf{1}\ 10 \\
& x_2 = \mathbf{insertNode}\ \mathbf{2}\ 20\ x_1 \\
& x_3 = \mathbf{insertNode}\ \mathbf{3}\ 30\ x_2 \\
& x_4 = \mathbf{insertNode}\ \mathbf{4}\ 40\ x_3 \\
& x_5 = \mathbf{insertEdge}\ \mathit{12}\ \mathbf{1}\ \mathbf{2}\ x_4 \\
& x_6 = \mathbf{insertEdge}\ \mathit{13}\ \mathbf{1}\ \mathbf{3}\ x_5 \\
& x_7 = \mathbf{insertEdge}\ \mathit{34}\ \mathbf{3}\ \mathbf{4}\ x_6 \\
& x_8 = \mathbf{insertEdge}\ \mathit{24}\ \mathbf{2}\ \mathbf{4}\ x_7 \\
& y_1 = \mathbf{insertNode}\ \mathbf{6}\ 60\ x_8 \\
& y_2 = \mathbf{insertNode}\ \mathbf{7}\ 70\ y_1 \\
\texttt{in} & \mathbf{insertEdge}\ \mathit{67}\ \mathbf{6}\ \mathbf{7}\ y_2
\end{array}
$$



Notations For the rest of the presentation, we use the following notations. Notation $\overline{q}^{1..n}$ is a shorthand form for the sequence $q_1, q_2, \ldots, q_n$, where q can be any meta-level element. We further call q_1 the head and q_n as the tail. When the length of the sequence does not matter, we further shorthand it as $\overline{q}$. We use ϵ to represent an empty sequence. Given two sequences ι and ι', notation ι, ι' represents their concatenation. When the sequence does not contain duplicates, and the order among its elements does not matter, we liberally treat it as a set, and apply common operators such as $\in$, $\subseteq$, $\setminus$, and $\cup$. In our implementation, we use an efficient double-ended queue for all sequences which require $O(1)$ access to the head and the tail.

We call a sequence in the form of $[q_1 \mapsto q_1', \ldots, q_n \mapsto q_n']$ a *mapping*, defined only when q_1, q_2, $\ldots$, q_n are distinct. Given mapping μ, we denote $\mu(q_i)$ for q_i' for any $i = 1..n$, and $dom(\mu)$ as $\{q_1, \ldots, q_n\}$ and $\mathbf{ran}(\mu)$ as $\{q_1', \ldots, q_n'\}$. We overload $\mu \setminus q$ as restricting μ to the domain of $dom(\mu) - \{q\}$. For two mappings μ and μ', $\mu \uplus \mu' = (\mu \setminus \mu'), \mu'$ if $\forall q \in dom(\mu), q \notin dom(\mu')$ or $\mu(q) = \mu'(q)$. The operator is otherwise undefined.

3. Dynamic Behavior Overview

In this section we introduce the dynamic behavior of DELTAGRAPH. We will cover run-time representation, graph lookup and fold, and update realization. We delay propagation support to a section of its own: Section 4.

3.1 Run-time values

A DELTAGRAPH graph at run-time—which we also call a *delta graph* and represent with metavariable Γ—is a 6-tuple as shown in Figure 2. For succinctness, we use this tuple form which in our implementation is an algebraic data type. The components are

- a *root* key `nk`, the node key identifying the entry point of the graph.
- a *payload (expression) e*, for the root node
- *inductive edges* Π, a mapping from edge keys to (sub) delta graphs. This definition reveals the inductive nature of delta graphs. Each element $\texttt{ek} \mapsto \Gamma$ says that the current node (the one with the root key) is connected with a (sub)graph through edge `ek`.
- *back/cross edges* $\hat{\Pi}$, a mapping from edge keys to node keys. Each element $\texttt{ek} \mapsto \texttt{nk}$ says that the current node is connected to node `nk` through a back/cross edge.
- *delta list* Δ, a sequence of "memoized" update intentions.
- *disconnected components* $\hat{\Delta}$, a sequence of graphs disconnected from the current graph.

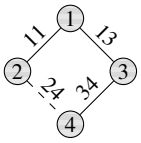
Other than the components common for representing inductive graphs, the unique components here are the delta list and disconnected components. Intuitively, a graph Γ with a delta list $\delta_1, \ldots, \delta_n$ is a "lazy" representation of a graph that would have been produced by applying a sequence of eager updates $\delta_1, \ldots, \delta_n$ to Γ, *in that order*. The disconnected components represent all graphs which have no current inductive attachment to the current graph. This offers both a natural representation of forests as well as a method of capturing the intention of merging a graph into the current graph. The graphs to be merged are introduced at run-time when a node is inserted into the graph without specifying an existing node to attach to or when a sub-graph becomes detached from the inductive graph due to node or edge deletion.

In the rest of the paper we explicitly distinguish between a logical graph—the logical structure in the mind of the programmer such as the figures in Example 1 and Example 3—and the run-time DELTAGRAPH graph. We use the following legend in our figures to distinguish between a logical graph node and a DELTAGRAPH graph node.

Further, we stylize inductive edges as solid lines and back edges as dotted lines. We take the convention of visualizing each delta graph as an inductive structure where the root is at the top, and leaves are toward the bottom.

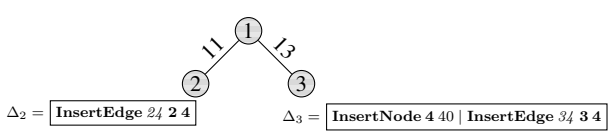
Example 4 (Run-time representations)**.** Due to lazy propagation, the same logical graph may have multiple run-time representations. As an example, the logical graph illustrated in Example 1 can be represented at run-time as the following, where all updates have been realized:

$$
\begin{array}{lll}
\langle \mathbf{1}; 10; & \mathit{13} \mapsto & \langle \mathbf{3}; 30; \mathit{34} \mapsto \langle \mathbf{4}; 40; \epsilon; \epsilon; \epsilon; \epsilon \rangle; \epsilon; \epsilon; \epsilon \rangle, \\
& \mathit{12} \mapsto & \langle \mathbf{2}; 20; \epsilon; \mathit{24} \mapsto \mathit{4}; \epsilon; \epsilon \rangle \\
& \epsilon; \epsilon \rangle &
\end{array}
$$



Or only some updates are realized:

$$
\begin{array}{lll}
\langle \mathbf{1}; 10; & \mathit{12} \mapsto & \langle \mathbf{2}; 20; \epsilon; \epsilon; \mathbf{InsertEdge}\ \mathit{24}\ \mathbf{2}\ \mathbf{4}; \epsilon \rangle, \\
& \mathit{13} \mapsto & \langle \mathbf{3}; 30; \epsilon; \epsilon; \\
& & \mathbf{InsertNode}\ \mathbf{4}\ 40, \mathbf{InsertEdge}\ \mathit{34}\ \mathbf{3}\ \mathbf{4}; \epsilon \rangle; \\
& \epsilon; \epsilon; \epsilon \rangle &
\end{array}
$$



Or none have been realized:

$$\langle \mathbf{1}; 10; \epsilon; \epsilon; \begin{array}{l}\mathbf{InsertNode\ 2}\ 20,\\ \mathbf{InsertNode\ 3}\ 30,\\ \mathbf{InsertNode\ 4}\ 40,\\ \mathbf{InsertEdge}\ \mathit{12}\ \mathbf{1\ 2},\\ \mathbf{InsertEdge}\ \mathit{13}\ \mathbf{1\ 3},\\ \mathbf{InsertEdge}\ \mathit{34}\ \mathbf{3\ 4},\\ \mathbf{InsertEdge}\ \mathit{24}\ \mathbf{4\ 4}\end{array} ; \epsilon \rangle$$

$\Delta_1 =$ **InsertNode 2** 20 | **InsertNode 3** 30 | **InsertNode 4** 40 | **InsertEdge** *12* **1 2** | **InsertEdge** *13* **1 3** | **InsertEdge** *34* **3 4** | **InsertEdge** *24* **4 4**

3.2 Lazy Update

Any graph operation involving graph updates will be one-step reduced to appending an update intention (which we henceforth call *update label*) to the *tail* of the delta list of the graph's root. Given a delta graph $\Gamma = \langle \texttt{nk}; e; \Pi; \hat{\Pi}; \Delta; \hat{\Delta} \rangle$, and an update operation whose corresponding update label is δ, we define this append operation as:

$$\Gamma \oplus \delta = \langle \texttt{nk}; e; \Pi; \hat{\Pi}; \Delta, \delta; \hat{\Delta} \rangle$$

The notion of "corresponding update label" is straightforward: if the update expression is $\mathbf{insertNode}\ \texttt{nk}\ e\ e'$, then the corresponding update label is $\mathbf{InsertNode}\ \texttt{nk}\ e$; if the update expression is $\mathbf{deleteNode}\ \texttt{nk}\ e$, the corresponding update label is $\mathbf{DeleteNode}\ \texttt{nk}$; and so on.

This very simple computation epitomizes the true spirit of DELTAGRAPH: updates are an O(1) operation, as cheap as a sequence append. The most challenging part of lazy update evaluation is how this particular update label is propagated through the graph; we delay this discussion to Section 4.

3.3 Lookup and Fold

The node lookup procedure must only consider nodes that would reside in the corresponding logical graph. This process needs careful consideration in the presence of memoized updates. A solution must (1) include graph nodes that would have been realized but are still in the delta list; (2) exclude those that exist in the graph structure but will be deleted according to the (deletion) update labels in the delta lists; and (3) obey the chronological order of the updates.

The $\mathbf{lookupNode}\ \texttt{nk}\ \Gamma$ expression reduces to $(\mathit{nodes}(\Gamma))(\texttt{nk})$ in one step where *nodes* is defined in Figure 3. Function *nodes* computes all of the nodes in the graph as a mapping (metavariable M) from node keys to their corresponding payloads. The function depends on the *nodesDelta* function also in Figure 3. The key observation is that *nodesDelta* must take into account the memoized node insertion labels according to (1) above, the memoized node deletion labels according to (2) above, and must be defined by preserving the order in the delta list according to (3) above. Similarly, expression $\mathbf{lookupEdge}\ \texttt{ek}\ \Gamma$ one-step reduces to $(\mathit{edges}(\Gamma))(\texttt{ek})$ where *edges* is also defined in Figure 3. Function *edges* computes edges in a delta graph as a mapping (metavariable B) from edge keys to a 2-member set of nodes to indicate the two nodes connected by the edge.

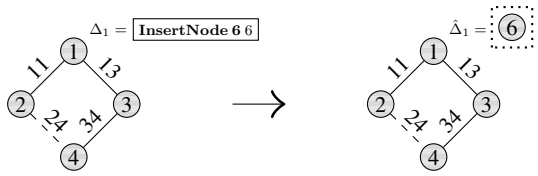
Our *nodes* and *edges* functions are mathematical in nature. Our implementation is more efficient: for instance, a lookup does not need to compute all node/expression mappings, and can just evaluate to the first expression it encounters with the matching key.

The $\mathbf{foldNodes}\ e\ e'\ \Gamma$ expression reduces to $fe_k \ldots (fe_1\ e')$ in one step where $\overline{\texttt{nk} \mapsto e}^{1..k} = \mathit{nodes}(\Gamma)$. An example fold is the expression $\mathbf{foldNodes}\ (*)\ 1\ e$ which folds the payload values of all nodes of *e* together through multiplication with an initial value of 1.

3.4 Update Realization

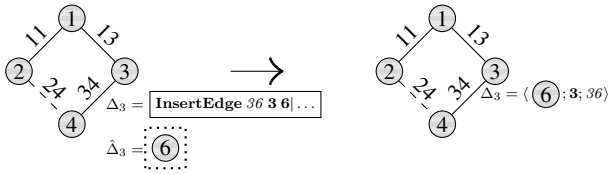
We now describe update realization, *i.e.*, the actions taken when the node/edge keys involved in an update label happen to relate to the root node of the current graph. For example, given the update label $\mathbf{InsertEdge}\ \mathit{12}\ \mathbf{1\ 2}$, when the current root node's key is either **1** or **2** then this edge insertion operation should be realized. We now informally describe the behavior of realizing every update label through visualization.

The graph below visualizes node insertion. A new node cannot be immediately attached to the inductive structure so it is realized as a singleton graph of its own in the disconnected components where it will remain until an attachment opportunity presents itself.

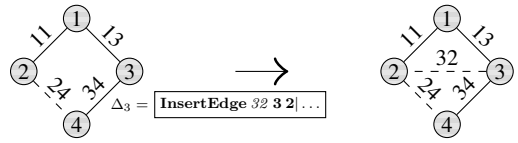


Edge insertions realize in two ways depending on how the connecting nodes relate in the inductive structure.

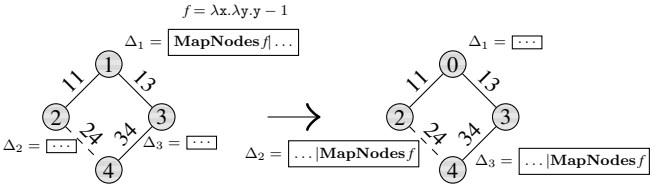
First, an inductive edge may be inserted when one of the nodes that the edge connects to is a separate graph in the current graph node's disconnected components. As seen below, the separate graph is then replaced with a graph merging update. Eventually the subgraph carried by the merging update label will be attached inductively to the other connecting node. In the illustration below, this can occur immediately as the graph can be attached to node **3**.



Second, when both nodes are already realized then a back/cross edge is inserted as shown below.



A node realizes a mapping update, depicted below, by applying the function to its node key and expression and then propagating the update down to each of its children. Each child repeats this process until each node in the graph realizes the update.



Node and edge deletion are the trickiest updates to support as they can lead to detached subgraphs, henceforth called *orphans*. Here we informally visualize the process of node deletion. There are two considerations when deleting a node: (1) all edges pointing to the node must be deleted (2) the node itself must be deleted. The graph that follows depicts both (1) and (2): node **2** removes its

Node Lookup

$M ::= \overline{\mathtt{nk} \mapsto e}$

$$
\begin{array}{lll}
nodes(\langle \mathtt{nk}; e; \Pi; \hat{\Pi}; \Delta; \hat{\Delta} \rangle) & \triangleq nodesDelta(\biguplus_{\Gamma \in \mathbf{ran}(\Pi) \cup \hat{\Delta}} nodes(\Gamma) \uplus (\mathtt{nk} \mapsto e), \Delta) & \\
nodesDelta(M, \epsilon) & \triangleq M & \\
nodesDelta(M, (\mathbf{InsertNode}\ \mathtt{nk}\ e, \Delta)) & \triangleq nodesDelta(M \uplus (\mathtt{nk} \mapsto e), \Delta) & \\
nodesDelta(M, (\mathbf{Merge}\ \Gamma\ \mathtt{nk}\ \mathtt{ek}, \Delta)) & \triangleq nodesDelta(M \uplus (\mathtt{nk}' \mapsto e'), \Delta) & \text{where } \Gamma = \langle \mathtt{nk}'; e'; \Pi'; \hat{\Pi}'; \Delta'; \hat{\Delta}' \rangle \\
nodesDelta(M, (\mathbf{DeleteNode}\ \mathtt{nk}, \Delta)) & \triangleq nodesDelta(M \setminus \mathtt{nk}, \Delta) & \\
nodesDelta(M, (\mathbf{MapNodes}\ f, \Delta)) & \triangleq nodesDelta(\{\mathtt{nk} \mapsto f\, e \mid \mathtt{nk} \mapsto e \in M\}, \Delta) & \\
nodesDelta(M, (\delta, \Delta)) & \triangleq nodesDelta(M, \Delta) &
\end{array}
$$

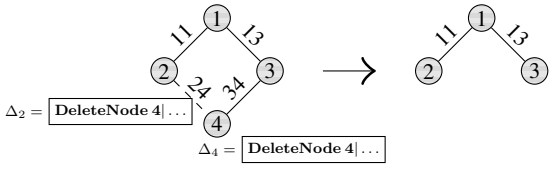
Edge Lookup

$B ::= \overline{\mathtt{ek} \mapsto \{\mathtt{nk}, \mathtt{nk}\}}$

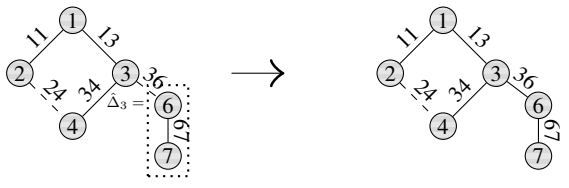
$$
\begin{array}{lll}
edges(\langle \mathtt{nk}; e; \Pi; \hat{\Pi}; \Delta; \hat{\Delta} \rangle) & \triangleq edgesDelta(\biguplus_{\Gamma \in \mathbf{ran}(\Pi) \cup \hat{\Delta}} edges(\Gamma) \uplus (\mathtt{nk} \times \Pi) \uplus (\mathtt{nk} \times \hat{\Pi}), \Delta) & \\
edgesDelta(B, \epsilon) & \triangleq B & \\
edgesDelta(B, (\mathbf{InsertNode}\ \mathtt{nk}\ e, \Delta)) & \triangleq edgesDelta(B, \Delta) & \\
edgesDelta(B, (\mathbf{Merge}\ \Gamma\ \mathtt{nk}\ \mathtt{ek}, \Delta)) & \triangleq edgesDelta(B \uplus \mathtt{ek} \mapsto \{\mathtt{nk}, \mathtt{nk}'\}, \Delta) & \text{where } \Gamma = \langle \mathtt{nk}'; e; \Pi'; \hat{\Pi}'; \Delta'; \hat{\Delta}' \rangle \\
edgesDelta(B, (\mathbf{InsertEdge}\ \mathtt{ek}\ \mathtt{nk}\ \mathtt{nk}', \Delta)) & \triangleq edgesDelta(B \uplus \mathtt{ek} \mapsto \{\mathtt{nk}, \mathtt{nk}'\}, \Delta) & \\
edgesDelta(B, (\mathbf{DeleteNode}\ \mathtt{nk}, \Delta)) & \triangleq edgesDelta(B \setminus\setminus \mathtt{nk}, \Delta) & \\
edgesDelta(B, (\mathbf{DeleteEdge}\ \mathtt{ek}, \Delta)) & \triangleq edgesDelta\ (B \setminus \mathtt{ek}, \Delta) & \\
edgesDelta(B, (\delta, \Delta)) & \triangleq edgesDelta(B, \Delta) & \\
B \setminus\setminus \mathtt{nk} & \triangleq \{\mathtt{ek} \mapsto B(\mathtt{ek}) \mid \mathtt{nk} \notin B(\mathtt{ek})\} & \\
\mathtt{nk}_0 \times \overline{\mathtt{ek} \mapsto \Gamma} & \triangleq \overline{\mathtt{ek} \mapsto \{\mathtt{nk}_0, \mathtt{nk}\}} & \text{where } \Gamma = \langle \mathtt{nk}; e; \Pi; \hat{\Pi}; \Delta; \hat{\Delta} \rangle \\
\mathtt{nk}_0 \times \overline{\mathtt{ek} \mapsto \mathtt{nk}} & \triangleq \overline{\mathtt{ek} \mapsto \{\mathtt{nk}_0, \mathtt{nk}\}} &
\end{array}
$$

Figure 3: Node and Edge Lookup

cross edge to node **4** and node **3** removes node **4**. On larger graphs, a back/cross edge to the deleted node could still exist in the rest of the structure. For this reason, the realization of a node deletion is followed by further propagation of the node deletion. A more complicated example involving orphans is discussed in Example 5.



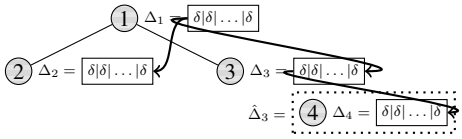
Two graphs can always be merged if they have a connecting edge. The following graph offers an example where a subgraph containing nodes **6** and **7** is attached to the main graph by changing the cross edge *36* into an inductive edge.



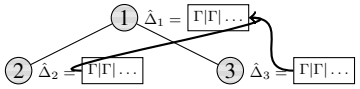
4. Propagation Support

In this section, we explore the design space in the propagation of updates in DELTAGRAPH. Recall that we take the convention of visualizing the delta graph with the root at the top. As a result, we refer to a propagation among delta graph nodes along a root-leaf path as a *vertical propagation*, and informally use *downward* and *upward* to refer to a leaf-bound and root-bound vertical propagation, respectively. In contrast, we refer to update label movement within a delta list as *horizontal propagation*. DELTAGRAPH supports all three forms of propagation for different purposes, as illustrated in Figure 4. Overall, the design is guided by several high-level principles:

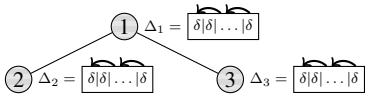
- **Update Label Management is Decentralized**: every graph node may be associated with its own delta list, carrying a set of update labels. This is consistent with the definition of the delta list, where each inductive subgraph carries its own delta list; see all three subfigures of Figure 4.
- **(Inter-Node) Vertical Order Matters**. There is an implicit chronological order among delta lists encountered in any path from the root to a leaf: the delta list inside a more leaf-bound graph node memoizes *older* updates than any inside a more root-bound graph node along the same path. This property follows from our propagation style, where delta list items are propagated downward, as seen in Figure 4a. In contrast, DELTAGRAPH supports upward propagation for items in the disconnected components, as seen in Figure 4b.
- **(Intra-Node) Horizontal Order Matters**. There is an implicit chronological order for the update labels within each delta list. A more head-bound update label within a delta list represents an *older* update than a more tail-bound update label, a prop-



(a) Downward Vertical Propagation



(b) Upward Vertical Propagation



(c) Horizontal Propagation

Figure 4: Three Forms of Propagation

erty following the propagation style where a newly propagated update label is appended to the tail of the delta list. This is illustrated in Figure 4c.

4.1 Vertical Propagation

Graph nodes propagate update labels downward when the node and edge keys involved in the label do not relate to the root node, *e.g.*, an edge insert label which doesn't mention the root node's key, or when the update affects the entire graph, *e.g.*, a graph mapping update.

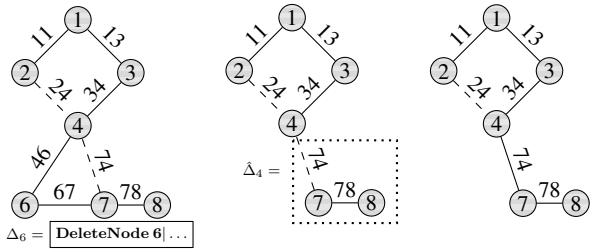
Downward propagation passes an update label to all children of the nodes in Π as well as all of the orphaned graphs in $\hat{\Delta}$. Figure 4a visualizes the propagation path that update labels take to the delta list of the children of node $\mathbf{1}$ and to the delta list of the orphaned graph of node $\mathbf{3}$.

This process explains why on a root-to-leaf path, the update labels in a leaf-bound delta list are older. All internal graph operations must adhere to the chronological order of updates. In a particular example, when a node is deleted, all of its update labels must be propagated down to the orphaned graphs in the disconnected components, the previous children of the deleted node.

The upward propagation specifically addresses orphans in the disconnected components. Intuitively, the orphaned subgraph may contain back/cross edges that reference a node in the inductive structure that it was detached from, so by propagating it through the graph we may find opportunities to *reattach* it. Such orphans are propagated upward, and reattachment opportunities are sought after along the upward propagation steps. Figure 4b visualizes the propagation path that orphans take.

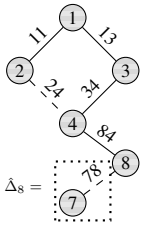
When an orphan moves up, the update labels of its new owner are chronologically newer than the update labels of the orphan according to *Vertical Order Matters*. The labels must be appended observing this chronological order as the orphan moves its way up.

Example 5 (Node Deletion and Orphan Reattachment)**.** In graph (a) below, node $\mathbf{6}$ is ready for deletion.



(a) (b) (c)

The realization of the $\mathbf{DeleteNode\ 6}$ update label leaves an orphaned subgraph which is outlined in a dashed rectangle in graph (b). The orphan, consisting of nodes $\mathbf{7}$ and $\mathbf{8}$, the inductive edge 78, and the back edge 74, is stored inside the disconnected components of node $\mathbf{4}$. Through orphan reattachment, the back edge 74 connecting nodes $\mathbf{4}$ and $\mathbf{7}$ is transformed into an inductive edge. Graph (c) shows the final graph after reattachment. If, instead of a back edge connecting $\mathbf{7}$ and $\mathbf{4}$, there was a back edge connecting $\mathbf{8}$ and $\mathbf{4}$, orphan reattachment would result in the following graph:



4.2 Pre-natal Optimization

Horizontal propagation is the rearrangement of update labels in a delta list in the hopes to optimize further operations. All labels propagate from the tail to the head of a delta list as seen in Figure 4c. In other words, whenever an update is applied, it should always be placed at the *tail* of a delta list, according to *Horizontal Order Matters*.

In essence, horizontal propagation defines the ways that update labels may be rearranged while retaining the logical *meaning* of the updates. The goal is *pre-natal optimization*: update labels may relate to one other and preprocessing may simplify the delta list or simplify future operations. On the high level, we propagate node deletion labels and node mapping labels to the head of the delta list. On every update label swap we apply any changes the two labels *would have* led to had they both been realized. We now describe every swapping scenario.

Intuitively, a node which is inserted and immediately deleted can be entirely forgotten: saving the cost of propagating and eventually realizing this node. Instead of swapping two update labels such as $\mathbf{InsertNode\ 6}\ 6$ and $\mathbf{DeleteNode\ 6}$ we simply remove them both.

A less intuitive scenario is when swapping a node deletion label and an edge insert label. Consider the two update labels $\mathbf{InsertEdge}\ 34\ \mathbf{3\ 4}$ and $\mathbf{DeleteNode\ 3}$. The edge insertion label can safely be removed. However, we must keep the node deletion label to continue propagating—the node it is deleting still resides in the graph. This leaves us with a result of just $\mathbf{DeleteNode\ 3}$. If instead we were deleting the node with key $\mathbf{5}$ then the two labels would safely swap into $\mathbf{DeleteNode\ 5}$, $\mathbf{InsertEdge}\ 34\ \mathbf{3\ 4}$. A node deletion label swaps with a graph merge label by propagating the node deletion to the merging graph, *i.e.*, $\mathbf{Merge}\ \Gamma\ \mathtt{nk}\ e, \mathbf{DeleteNode}\ \mathtt{nk}'$ swaps to $\mathbf{DeleteNode}\ \mathtt{nk}'$,

$\mathbf{Merge}\ (\Gamma \oplus \mathbf{DeleteNode}\ \mathtt{nk}')\ \mathtt{nk}\ e$. Node deletion update labels can safely swap in any other situation.

Two mapping update labels can be combined into one through function composition, *i.e.*, the update labels $\mathbf{MapNodes}\ f$ and $\mathbf{MapNodes}\ f'$ combine into $\mathbf{MapNodes}\ f' \circ f$. This rule indicates an effective approach in the presence of frequent payload updates: the earlier payload updates are *de facto* discarded. A mapping node swaps with a node insertion label by first applying the function to the node expression, *i.e.*, $\mathbf{InsertNode}\ \mathtt{nk}\ e$, $\mathbf{MapNodes}\ f$ swaps to $\mathbf{MapNodes}\ f$, $\mathbf{InsertNode}\ \mathtt{nk}\ (f\ e)$. Similarly, a mapping update label swaps with a graph merge label by propagating the mapping update to the merging graph. For example the update labels $\mathbf{Merge}\ \Gamma\ \mathtt{nk}\ \mathtt{ek}$ and $\mathbf{MapNodes}\ f$ swap to become $\mathbf{MapNodes}\ f, \mathbf{Merge}\ (\Gamma \oplus \mathbf{MapNodes}\ f)\ \mathtt{nk}\ \mathtt{ek}$. Mapping update labels can safely swap with any edge insertion label.

The repeated application of horizontal propagation leads to a normal form of a delta list following the structure $\Delta, \Delta', \Delta''$ where all labels in Δ are of the form $\mathbf{DeleteNode}\ \mathtt{nk}$; all labels in Δ' are of the form $\mathbf{MapNodes}\ f$; and Δ'' contains all other update labels. DELTAGRAPH does not require delta lists to be in normal form, but there is a practical benefit to perform normalization. Recall that the $nodes$ function in Section 3.3 inspects the delta list in its original order to determine whether a node exists in the logical graph. If this delta list is normalized, all the "should have been deleted" nodes will be at the head of the list, and "should have been inserted" nodes are in the Δ'' portion of the normal form in the definition above. Both may lead to faster lookups.

5. Implementation and Experimental Settings

We have implemented DELTAGRAPH as an open-source Haskell library on top of GHC version 7.10.3. Our library supports several addition features not discussed in the paper: (1) we support directed graphs; (2) we support graph filtering in addition to mapping and folding; (3) we support edge operations (map, filter, fold) in addition to node operations.

To validate the usefulness and efficiency of DELTAGRAPH, we have implemented a simplified version of Twitter [1], which we call MINITWITTER. As a social network, Twitter's user base forms a graph. MINITWITTER preserves a similar graph structure, where users are nodes and uni-directional 'following' relationships are represented by edges; when user A follows user B, a directed edge is created from A to B. The payload of each node is a string, uniquely identifying the user. MINITWITTER supports lookup of users and their following status, adding users, following other users, removing users, and unfollowing users. Such a representation is sufficient to support tools that use Twitter data for mining (*e.g.*, the use of Cassovary for Twitter's "Who to Follow" service [14]). MINITWITTER is built as a client library on top of DELTAGRAPH. The interface between the two is well-defined, so that we can perform comparative studies.

MINITWITTER is built to process large numbers of "request traces." Each request trace is a long sequence (approximately one million) of graph operations as in Example 1, where each operation in that sequence can be either a (i) new user request; (ii) following request; (iii) removal of user request; (iv) unfollowing request; (v) user lookup; or (vi) following lookup. Informally, we also call (i)(ii)(iii)(iv) *update requests*, and (v)(vi) *lookup requests*. The request traces are generated from two sources.

- Real-world twitter data gathered using the Twitter API [2] resulting in social network data which has been shown to be scale-free [7]. This data was collected by mining user following relationships on a subset of twitter users.
- Uniformly generated graphs where each node has a uniformly random chance of being attached to all other nodes.

Selecting both sources allows us to balance between real-world common cases (scale-free graphs) and generality (randomly generated graphs). We run each experiment using both sources and present both sets of results.

In Section 4.2, we discussed the benefit of normalization: even though DELTAGRAPH does not require normalization, delta lists in normal form may speed up lookups. During our experiments, we found better performance is achieved in nearly all scenarios if delta lists are maintained in the normal form. This is the setting for our following experiments. When an update label is added to the tail/head of a delta list, normalization is performed.

In scale-free graphs, some "hub" nodes are connected with a disproportional number of edges (*e.g.*, Obama in Twitter graphs). These nodes cause inefficiencies in our implementation which benefits from cheap propagation and normalization steps. To handle large, "hub" nodes efficiently, our implementation treats them in a fashion that can be analogously thought of them as multiple smaller nodes. In other words, we handle the delta lists in "chunks," and normalization and propagation is only performed at the chunk level. Thanks to this design, differences in results between the two different graph sources are slight.

We additionally developed an eager, standard graph-theoretic implementation which we call STANDARDGRAPH. Our implementation uses binary trees as an optimization over node and edge sets—a strategy that is generally applied in mature graph libraries. We also interfaced with an existing graph library FGL [12].

All of our benchmarking results are produced on an Intel(R) Core(TM)2 Quad CPU Q8200 at 2.33GHz with 8GB of memory. All results are the average over 30 runs. The standard deviation in none of our experiments exceeds 3%.

6. Experimental Evaluation

In this section we discuss the experimental results of DELTAGRAPH. We divide our results into the impacts of delta propagation, parallelization, and opportunistic computing; the different forms of lookup; and a comparative study with other Haskell graph libraries.

6.1 Delta Propagation

DELTAGRAPH allows for nondeterministic propagation. Intuitively, on one hand, an overly lethargic propagation may increase the delta list length of root-bound nodes, which in turn negatively impacts performance due to the increased amount of update labels that lookup normalization must handle on those high traffic nodes. On the other hand, an overly zealous propagation may also be suboptimal, as nodes being queried could be propagated many steps from the root and require more recursive steps to reach in a lookup; in addition, it thwarts the possibility of pre-natal optimization. Overall, we believe the grand question of "to propagate or not to propagate" at any node of the inductive graph should be influenced by two factors:

- The likelihood that elements in the delta list will be queried in the future: the higher the likelihood, the more lethargic the propagation should stay;
- The length of the delta list: the longer the list, the more zealous the propagation should stay;

To experiment with this design decision, we run MINITWITTER with DELTAGRAPH using (1) varying frequencies at which propagation happens; (2) varying (maximum) delta list lengths that propagation aims to maintain; (3) varying data correlations between

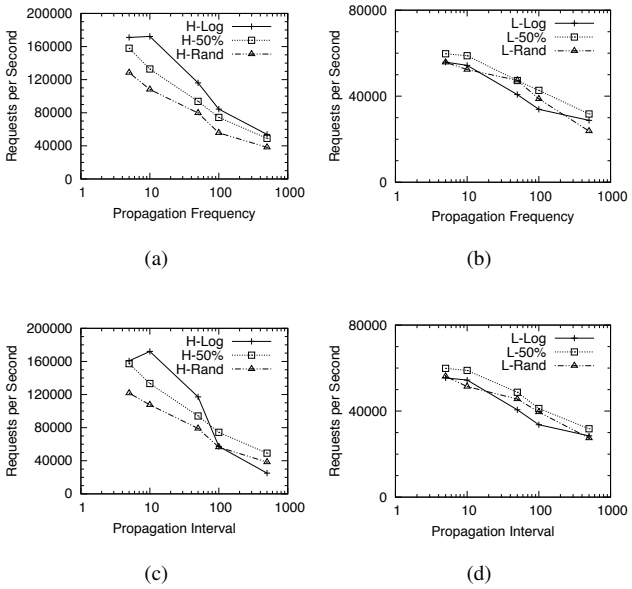
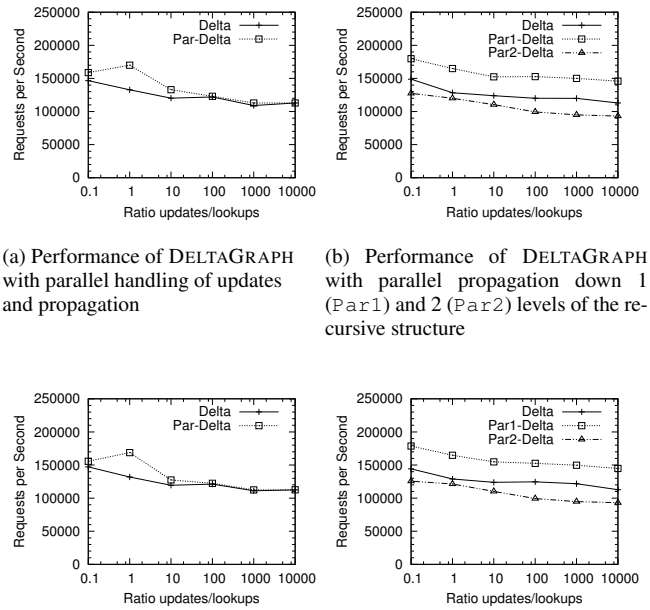


Figure 5: The effect different propagation strategies and update/lookup correlations have on DELTAGRAPH performance on scale-free graphs (a-b) and on randomly generated graphs (c-d). The X-axis is the interval between two propagations (as the number of requests in between).



(a) Performance of DELTAGRAPH with parallel handling of updates and propagation

(b) Performance of DELTAGRAPH with parallel propagation down 1 (`Par1`) and 2 (`Par2`) levels of the recursive structure

(c) The experiment in (a) on randomly generated graphs

(d) The experiment in (b) on randomly generated graphs

Figure 6: The Effect of Parallelism

updates and lookups. The frequencies are set by interspersing propagations through requests (in the request trace), varying from every propagation for 5, 10, 100, and 500 requests. The requests are chosen uniformly between new users, removal of users, following requests, unfollowing requests, user lookups, and following lookups.

The three delta list length maintenance strategies we experiment with are (1) `Log`, where propagation will trim the delta list until it only contains $\log K$ update labels, where K is the sum of the update labels in its immediate children; (2) 50%, where the threshold of trimming is set at $\frac{K}{2}$; (3) `Rand`: a baseline strategy where the propagation procedure is blind to the length of the delta list. The number of labels it trims and propagates is chosen uniformly randomly from the number of labels it has. In all three strategies, the propagation procedure "cascades" trimming: it starts from the root, propagates the trimmed labels to the root's children, and then start each children's trimming, until all the way to the leaf.

Each of these strategies is run with high (`H`) correspondence between the nodes/edges (users and following relationships) most recently inserted and operated upon, plotted in Figure 5a, and low (`L`) correspondence, plotted in Figure 5b. In the `H` setting, a node recently inserted into the graph or updated would have a much higher chance of being queried than older nodes do. Specifically, under `H`, queries follow an exponential distribution over the nodes/edges ordered by age with a mean of 10; newer inserts are exponentially more likely to be queried than older ones. Under low correspondence the exponential distribution has a mean of 1000. Each run takes on the order of tens of seconds and the requests completed per second is shown.

The difference between the high and low correspondence immediately stands out. Intuitively, this says DELTAGRAPH is more friendly for graph processing if update/lookup requests are strongly correlated with recent update/lookups. In the real world, this is a common phenomenon: not all Twitter users are equally active, and some active contributors make frequent updates/lookups than others. The design of DELTAGRAPH happens to reward these frequent users in terms of response rate.

We further see that mostly, the more frequent propagation happens, the higher the response rate of the system. Indeed, in the extreme case where a propagation is issued for every 500 requests (toward the right of the graph), the root node may have an overly long delta list to hamper performance. The only exception to this is that the `Log` strategy does better when propagated every 10 requests than it does with every 5 requests. This shows that too much propagation may also be harmful. The propagation overhead can cost more than it saves if done too often.

The `Log` strategy dominates the other strategies in the scale-free results by a larger margin and more consistently than it does for the random graphs. This is more than likely due to hub nodes, even though they are chunked, receiving many update labels. A relatively active propagation strategy suits the high traffic well.

Next, let us visit the relative effectiveness of the three length-based strategies. Overall, the strategy choice makes more difference for `H` than for `L`. Considering real-world graph requests often come with some level of data correspondence, the `H` case is more interesting. One interesting fact about `Log` is it has the sharpest decline. It outperforms the other two strategies when propagations are more frequent, but underperforms when propagation becomes more rare. In other words, when used at the "right" time, `Log` can be the best strategy. It should not be used when propagations are rare: `Log` is the strategy which propagates the most update labels; infrequent propagation of large numbers of labels hurts the strategy's performance greatly. The `Rand` strategy propagates blindly and sporadically; sometimes propagating large numbers of labels while sometimes only propagating a few. This imbalance causes its poor performance in `H`.

6.2 Parallelization

Functional graphs are generally friendly for parallelization. For instance, multiple lookups can be conducted in parallel, which of-

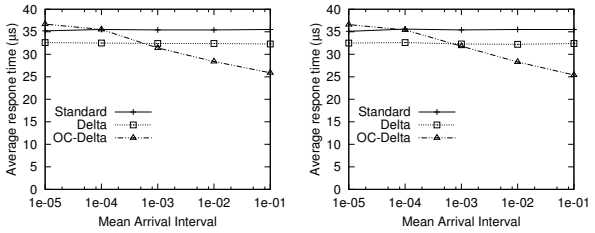


Figure 7: DELTAGRAPH and Opportunistic Computing on scale-free graphs (a) and randomly generated graphs (b)

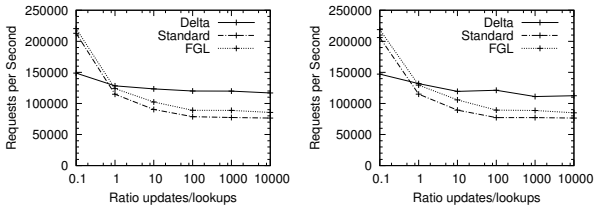


Figure 8: DELTAGRAPH, STANDARDGRAPH, and FGL on scale-free graphs (a) and randomly generated graphs (b)

ten leads to enormous speedups. More relevant to this paper, is (1) whether lookups and propagations can be conducted in parallel, and (2) whether a propagation procedure can be divided into parallel/-concurrent units by itself.

While a lookup is being processed, propagation can occur. The referential transparency of Haskell (and pure functional languages in general) allows this to happen without the need of any concurrency constructs. The lookup happens on the 'old' version of the graph while the propagation procedure produces a 'new' copy. The results in Figure 6a show `Par-Delta` (the implementation with lookup and propagation happens in parallel) dominating `Delta` in the first three ratios peeking at a speed up of 30% when the number of updates and lookups are equal. The loss of speed up in the larger ratios is due to there not being many opportunities to gain performance. There are few lookups and the parallelization is not able to increase speed by any noticeable amount.

To study (2) above, we run DELTAGRAPH except that when an update label is propagated to a node's children, each child spawns a new thread for further propagation. `Par1-Delta` describes the scenarios when only the propagation to the root nodes' children is parallelized: there is only 1 level of parallel propagation. `Par2-Delta` is when two levels of propagation: each root's grandchildren also conducts propagation in its own thread.

For this experiment, the correlation between updates and lookups was high. The system we used had 4 cores. The results in Figure 6b show parallelization of propagation may help, but only to an extent: `Par1-Delta` significantly outperforms `Delta`, but `Par2-Delta` underperforms. This is a cautionary tale familiar to parallel computing: excessive divide-and-conquer may leave too little work for each thread, and too much cost on thread management.

6.3 Opportunistic Computing

In this experiment we introduce the concept of *request density* into the system. During periods of decreased request density, DELTAGRAPH is able to propagate nodes no longer being updated or queried in the hopes of speeding up delta list normalization and lookup times during the next increased density work load.

The system is implemented using two threads. There is a main thread which handles all requests and a propagation thread which will attempt to propagate when there are no pending requests. There is no preemption in our system. The propagation thread is used as speculative computing, where its computation is only valid if, during its run-time, the main thread received no new requests. We implement the request trace with a particular request density through timer-guided issuance of requests. To mimic real-world scenarios, we construct the experiment by generating an exponential distribution of requests and tracked changes as the mean request arrival time is varied. This is distinct from earlier experiments where requests were being made unencumbered and without delay. Request arrival modeling with exponential distribution is common in network research.

Figure 7 shows the results, with the new experiments labeled `OC-Delta`. The average response time in microseconds is plotted. `OC-Delta` outperforms regular DELTAGRAPH and STANDARDGRAPH after a mean arrival time of $1\mathrm{E}-3$. Before then, the request distribution prevents propagation from happening regularly. When the distribution allows for more time in between requests the propagation has much more of an impact and the average response rate improves.

6.4 Single Key vs. Clique Lookup

The relative under-performance of DELTAGRAPH in low-update scenarios of Figure 8 may appear disappointing. It may be quick to dismiss DELTAGRAPH as fundamentally unsuitable for building low-update graph processing systems.

This turns out depending on *the nature of the lookup*. For the earlier experiments, we assumed each lookup operation to be a single-key lookup. We implemented STANDARDGRAPH with binary search trees for its node/edge sets. FGL's run-time relies on a similar structure. In other words, the results from comparative studies with single-key lookups may be skewered due to features orthogonal to the focus of this paper.

To demonstrate the opposite scenario where DELTAGRAPH may outperform STANDARDGRAPH and FGL in low-update scenarios, we construct experiments where lookups are defined as *clique analysis*. On a graph with 10 thousand nodes and 100 thousand edges, our experiments attempt to find 10 cliques of size 3 or more. DELTAGRAPH is able to outperform both STANDARDGRAPH and FGL by 3 magnitudes, in both scenarios of a scale-free graph construction and a random graph construction. The key insight here is DELTAGRAPH as an inductive graph encodes locality in its graph. In one traversal, a significant number of cliques can be discovered by following the inductive edges. Re-traversal only occurs when the number of discovered cliques does not reach the threshold, and back/cross edges need to be followed. In contrast, neither STANDARDGRAPH nor FGL has graph locality encoded inside, so every step of connectivity check in the clique analysis would entail a traversal of the graph.

6.5 Comparative Studies

In this set of experiments, we run a variety of use scenarios on MINITWITTER, powered by either DELTAGRAPH, or STANDARDGRAPH, or FGL. We vary the use scenarios through generating the sequences of operations with different ratios between updates and (single-key) lookups. For example, with a ratio of 10, there are 10 updates for every 1 lookup. Lookups follow the same high corre-

spondence pattern as described in the previous section. The `Log` propagation strategy is used for DELTAGRAPH. The results are in Figure 8, showing MINITWITTER backed by DELTAGRAPH can handle between 110 and 150 thousand requests per second depending on the ratio of updates to lookups, giving it up to a 60% speed up over STANDARDGRAPH. The only exception is when (single-key) lookups start to outnumber updates. As we explained earlier, both STANDARDGRAPH and FGL have key-based binary search built-in, whereas DELTAGRAPH does not. This may explain their good performance in lookup-heavy use scenarios. DELTAGRAPH outperforms when processing highly dynamic graphs. This provides evidence on the effectiveness of DELTAGRAPH in update-intensive graph applications.

7. Related Work

Lazy evaluation is well-understood in functional languages. DELTAGRAPH highlights the duality of laziness: the computation centric laziness and data-centric laziness. Lazily evaluated languages and incremental computing systems [28, 3, 5, 16, 4] achieve laziness by memoizing, delaying, or reusing evaluations along a graph representing a *computation* whereas DELTAGRAPH achieves laziness by delaying expressions along a graph representing the *data* of a computation. This leads to a dual exploration of the design space: whereas the core design issue in computation-centric laziness is to propagate changes along the dependency graphs of *computations*, DELTAGRAPH propagates changes over the structure of the *data* itself.

Existing graph support in functional languages mainly addresses two goals: representation and query. Two early graph systems are FGL [12], and King and Launchbury [21]. We described FGL in the experimentation section. King and Launchbury is optimized for graph data lookup, with an efficient representation suitable for depth-first search. It does not consider dynamic graphs with update operations. Fegaras and Sheard [13] explored catamorphisms over graphs with embedded functions. Hamana [15] designed an algebra to support cyclic sharing structures. Oliveira and Cook [25] embedded an inductive graph language through higher-order abstract syntax (HOAS). Their system demonstrates some crucial advantages of inductive graphs, such as they serve as a natural basis for providing correctness guarantees on representing sharing and cycles. Hidaka *et al.* [18] studied bidirectional transformation of graphs in a query language setting. Inaba *et al.* [19] further provided support for transformation correctness via a monadic higher-order logic. More recently, an effort for querying graphs [17] focused on ordered graphs. λ_{FG}^{T} [6] supports general graph transformation over a variety of graphs (such as probabilistic graphs) via monads. None of these efforts explore laziness in graph support. The ideas behind DELTAGRAPH may enrich prior work with the ability of adapting to more update-intensive, performance-critical graph applications.

Dynamic data structures [27] and Just-In-Time data structures [9] [20] are examples where data representation may change at run-time as an optimization effort. They share the high-level view that a data structure can be viewed as a black box. The different representations dynamically constructed by those systems are driven by the nature of lookup, not updates. Our focus on how updates are propagated through the data structure is orthogonal and complementary.

An array in functional languages can be represented as a *version tree* [26] on the low level, whose updates lead to new versions. DELTAGRAPH is different from version trees in that we do not create multiple versions of the same graph, but instead propagate changes through the graph.

In database systems [10, 23], caching queries/updates is routinely supported. In the broadest sense, this can be viewed as a primitive form of data-centric laziness. The scope covered by DELTAGRAPH would analogously entail a rigorous design and verification of databases involving the query language, the update API, the database table design, the distributed cache design, and the data propagation and consistency protocols.

8. Conclusion

DELTAGRAPH is a novel graph programming system suitable for highly dynamic graphs. The solution improves the reactiveness of graph updates, and is friendly for parallelism and opportunistic computing.

Acknowledgments

We thank the anonymous reviewers for their useful comments. This project is partially sponsored by US NSF CCF-1526205 and CCF-1054515.

References

[1] Twitter. `http://twitter.com`.

[2] Twitter developers. `http://dev.twitter.com`.

[3] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, 1996.

[4] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, November 2009.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6), November 2006.

[6] Kazuyuki Asada, Soichiro Hidaka, Hiroyuki Kato, Zhenjiang Hu, and Keisuke Nakano. A parameterized graph transformation calculus for finite graphs with monadic branches. In *PPDP*, pages 73–84, 2013.

[7] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[8] Marco Conti and Mohan Kumar. Opportunities in opportunistic computing. *Computer*, 43(1), 2010.

[9] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B Sartor, and Wolfgang De Meuter. Just-in-time data structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 61–75. ACM, 2015.

[10] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems (TODS)*, 9(4):503–525, 1984.

[11] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[12] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, September 2001.

[13] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). POPL '96, 1996.

[14] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international confer-*

ence on World Wide Web, pages 505–514. International World Wide Web Conferences Steering Committee, 2013.

[15] Makoto Hamana. Initial algebra semantics for cyclic sharing structures. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, TLCA '09, pages 127–141, 2009.

[16] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[17] Soichiro Hidaka, Kazuyuki Asada, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Structural recursion for querying ordered graphs. ICFP '13, pages 305–318, 2013.

[18] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP*, pages 205–216, 2010.

[19] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation verification using monadic second-order logic. In *PPDP*, pages 17–28, 2011.

[20] Oliver Kennedy and Lukasz Ziarek. Just-in-time data structures. *CIDR*, 2015.

[21] David J. King and John Launchbury. Lazy Depth-First Search and Linear Graph Algorithms in Haskell. In *Glasgow Functional Programming Workshop*, 1994.

[22] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[23] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM, 2002.

[24] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. SIGMOD '12, pages 145–156, 2012.

[25] Bruno C.d.S. Oliveira and William R. Cook. Functional programming with structured graphs. ICFP '12, pages 77–88, 2012.

[26] Melissa E O'Neill and F Warren Burton. A new method for functional arrays. *JFP*, 7(5):487–513, 1997.

[27] Mark H. Overmars. *Design of Dynamic Data Structures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.

[28] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, 1989.