Graphflow: An Active Graph Database

Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, Semih Salihoglu
School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada
chathura.kankanamge@uwaterloo.ca, s3sahu@uwaterloo.ca, amine.mhedbhi@uwaterloo.ca, y522chen@uwaterloo.ca, semih.salihoglu@uwaterloo.ca

ABSTRACT
Many applications detect the emergence or deletion of certain subgraphs in their input graphs continuously. In order to evaluate such continuous subgraph queries, these applications resort to inefficient or highly specialized solutions because existing graph databases are passive systems that only support one-time subgraph queries. We demonstrate Graphflow, a prototype active graph database that evaluates general one-time and continuous subgraph queries. Graphflow supports the property graph data model and the Cypher++ query language, which extends Neo4j’s declarative Cypher language with subgraph-condition-action triggers. At the core of Graphflow’s query processor are two worst-case optimal join algorithms called Generic Join and our new Delta Generic Join algorithm for one-time and continuous subgraph queries, respectively.

1. INTRODUCTION
Evaluating subgraph queries, i.e., finding instances of a given subgraph in a larger graph, is a fundamental computation performed by many applications that process graphs. Existing graph databases, such as Neo4j [5] and OrientDB [7] only support one-time subgraph queries that are evaluated on a snapshot of the graph until completion. However, many applications need to evaluate continuous subgraph queries, which detect the emergence or deletion of a given subgraph continuously. We give three examples.

Example 1: Twitter’s MagicRecs recommendation application [4] continuously detects a diamond subgraph, shown in Figure 1(a) in Twitter’s who-follows-whom graph. The diamond subgraph consists of a user $a_1$ who follows two separate users $a_2$ and $a_3$, who both follow another user $a_4$. Once a diamond subgraph is detected, MagicRecs recommends user $a_1$ to follow user $a_4$, because two users that $a_1$ is following are interested in $a_4$.

Example 2: Consider a financial fraud detection application that has as input a transactions graph. In the graph, individual customers are vertices and an edge from two customers $a_1$ and $a_2$ indicates a transfer of money from $a_1$ to $a_2$. The application detects circular transactions, which may indicate money laundering activity, and reports them to a fraud detection specialist. This query would correspond to a continuous cycle query. The triangle query, shown in Figure 1(b) is an example cycle query.

Example 3: Consider a datacenter monitoring application that has permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

2. GRAPHFLOW SYSTEM
Figure 2 shows the high-level architecture of Graphflow. Graphflow is a single node in-memory system implemented in Java. There are four main components of the system: (1) an in-memory property graph store; (2) the Cypher++ query language, which extends...
Neo4j’s declarative Cypher language with subgraph-condition-action triggers; (3) a One-time Query Processor (OQP); and (4) a Continuous Query Processor (CQP). We next describe each component.

2.1 In-memory Property Graph Store

Graphflow’s data model is a property graph, i.e., a labeled graph. Graphs are directed and vertices and edges can have arbitrary key-value properties on them. The graph is stored in two components. Structure Store: Stores, for each vertex \( u \), its incoming and outgoing adjacency lists stored as arrays. The adjacency lists are stored in sorted order of the outgoing neighbor IDs, which allows efficient intersections of two adjacency lists. Property Store: Stores the key-value properties on vertices and edges. The keys can be arbitrary strings, and values can be integers, doubles, booleans, or strings.

2.2 One-time Subgraph Queries

One-time subgraph queries are expressed using original Cypher’s MATCH clause. The following is the one-time query shown visually in Figure 2 that finds all of the triangles in the graph that are formed by transfer type edges:

\[
\text{MATCH } \ast \{\text{type}=\text{transfer}\} \rightarrow \ast \{\text{type}=\text{transfer}\} \rightarrow \ast \{\text{type}=\text{transfer}\} \rightarrow \ast \{\text{type}=\text{transfer}\}
\]

The type ‘transfer’ brackets specify a filter on the edges.

Any subgraph query \( Q \) on an input graph \( G \) can be seen in relational algebra as a multiway join on replicas of an edges table that contains all of the edges in \( G \) as (source ID, destination ID) tuples. For example, the triangle query, ignoring filters, is equivalent, in Datalog syntax, to the query:

\[
\text{triangles } (a_1, a_2, a_3) := \text{edges}(a_1, a_2), \text{edges}(a_2, a_3), \text{edges}(a_3, a_1)
\]

We refer to the vertices in \( Q \), e.g., \( a_1, a_2, \) and \( a_3 \), as variables. Graphflow evaluates the equivalent multiway join queries using Generic Join (GJ) plans, which we describe next.

2.2.1 Generic Join \([6]\)

The traditional way of evaluating multiway join queries is using binary join plans, which join the input tables one at a time. This corresponds to evaluating subgraph queries by matching \( Q \) in \( G \) one edge-at-a-time. Recently, Ngo et al. \([7]\) showed that for many queries any binary join plan can produce many more intermediate tuples than the maximum worst-case output of the query. Ngo et al. then developed the first worst-case join algorithms, one of which is GJ. Unlike binary join plans, GJ evaluates queries using a vertex-at-a-time strategy. Let \( Q \) contain \( m \) variables, \( a_1, a_2, \ldots, a_m \), and \( n \) edges, GJ consists of two steps:

- **Global Variable Ordering**: As a first step GJ orders the variables in \( Q \) arbitrarily. Any permutation of the variables can be picked. We assume for simplicity that \( a_1, \ldots, a_m \) is picked.

2.2.2 Generic Join Plans

Given a MATCH query \( Q \), Graphflow constructs its logical GJ plan. The logical plan construction consists of picking an ordering of the variables in \( Q \) using heuristics, organizing the intersection operations that is consistent with the picked variable ordering, and deciding the placement of the filter operations in the query. Figure 3a shows an example plan for the triangle query in Graphflow’s Plan Viewer (PV) interface. We describe the optimizations we have implemented in Graphflow for constructing GJ plans.

**Sorted Adjacency Lists**: As discussed in Section 2.1. Graphflow stores the adjacency lists in sorted order of neighbor IDs, and uses the *galloping* intersection technique \([6]\) for fast intersections.

**Variable Ordering Heuristics**: We start with the highest degree variable in \( Q \), breaking the ties arbitrarily. Then, we iteratively pick the variable that is adjacent to the highest number of already picked variables, breaking ties by the overall degree or arbitrarily if the degrees are equal. Although GJ’s worst-case optimality is independent of the variable ordering, the ordering can have significant performance implications in practice, which we plan to demonstrate to the conference attendees.

**Filter Placement**: Any filters on a single vertex or edge in \( Q \), e.g., type ‘transfer’ on the edge \((a_2, a_3)\), is executed during the intersection operation that involves the edge \((a_2, a_3)\). This is done by the Intersection-and-Filter operation shown in Figure 3a. Any filter that accesses two or more properties is executed as a separate Filter operation that is pushed down to the first prefix subgraph it can be applied on.

2.3 Continuous Subgraph Queries

Continuous subgraph queries are expressed through subgraph-condition-action triggers using our CONTINUOUS MATCH clause extension to Cypher. The following is an example of a trigger that detects the directed triangles in a transactions graph and calls a
CONTINUOUS MATCH \( a_1 \cdot [\text{type}=\text{transfer}] \rightarrow a_2 \cdot [\text{type}=\text{transfer}] \rightarrow a_3 \),
ON EMERGENCE ACTION UDF reportCircularTransfer IN udf.jar

The other values for the ON clause are DELETION or ALL, which detect only the deletions or both the emergence and the deletions of the subgraph. Currently, the other ACTION value is FILE, which writes the subgraphs to a local file.

Updates, such as insertions or additions of edges, are one-time queries that arrive at GJP. Upon an update, GJP first temporarily modifies the in-memory graph store, which implicitly stores three different versions of the graph upon receiving an update:

- **Current**: Version of the graph prior to the update.
- **Delta**: Only the updates.
- **New**: New version of the graph after applying the updates.

GJP then notifies the CPQ of the updates, which checks for each registered trigger using DeltaGJ plans, if the subgraph specified in the trigger emerged or was deleted, and performs the action of the trigger. Once CPQ is done detecting subgraphs, the updates become permanent in the graph. We next explain CPQ's DeltaGJ plans.

### 2.3.1 Delta Generic Join Plans

DeltaGJ is a new incremental view maintenance algorithm we developed that evaluates a set of delta queries \( dQ_1, \ldots, dQ_n \) for \( Q \) using GJ as the input tables change to maintain the result of \( Q \). We note that, although the idea of delta queries is old and based on techniques from references [2, 3], our evaluation of \( dQ_i \) using GJ has performance implications that do not exist for existing IVM algorithms based on delta queries evaluated with binary join plans.

Upon an update, let \( c \)-edges, \( \Delta \)-edges, and \( n \)-edges correspond to the edges in the Current, Delta, and New graphs, respectively. Figure 4 shows examples of these tables when the input graph in Figure 2 is updated to Figure 3. The +/– signs indicate the inserted and deleted edges, which are also used to differentiate emerged and deleted subgraphs in the output. References [2, 3] have shown that the union of the following \( n \) queries gives the newly emerged tuples (or subgraphs) in \( Q \) due to updates.

\[
dQ_1 := \Delta x_{1}, c x_{1}, c x_{2}, c x_{3}, \ldots, c x_{n}, \delta \-edges_n \\
dQ_2 := n x_{1}, n x_{2}, n x_{3}, \ldots, n x_{n}, \delta \-edges_n \\
\ldots \\
dQ_n := n x_{1}, n x_{2}, n x_{3}, \ldots, \Delta x_{n} \\
\]

DeltaGJ evaluates each of these queries using GJ with the following important restriction. Note that in each \( dQ_i \), there is exactly one copy of the \( \Delta \)-edges table. Let \( a_i, a_j \) be the variables in that \( \Delta \)-edges table. The variable ordering of GJ when evaluating \( dQ_i \) has to start with \( a_i, a_j \) or \( a_j, a_i \). This corresponds to starting the evaluation from the vertices adjacent on the newly inserted or deleted edges when we match \( Q \) vertex at a time in \( G \). As we show in our tech-report [4], when variables are ordered in this fashion, DeltaGJ is worst-case optimal under insertion-only workloads.

Graphflow's CPQ produces the \( n \) logical plans for each query specified in each registered trigger. Figure 5 shows the three logical plans for the three delta queries when evaluating the triangle query. Notice that in each plan, the variables are ordered differently. Finally, note that for intersection operations, the plans include information about from which version of the graph (Current, Delta, or New) the adjacency lists should be read from.

### 3. PERFORMANCE

We compared the performance of Graphflow against Neo4j on one-time queries and PostgreSQL delta query plans that use binary joins on continuous queries. Unlike Graphflow, both systems use edge-at-a-time strategies for evaluating subgraph queries. Table 1 shows the three real-world input graphs we used. We used a Linux machine with 256GB RAM and Intel E5-2670 processor. We note that our experiments cover a limited set of input graphs and queries. We leave an extensive performance study to future work.

For one-time queries, we used the triangle, diamond, and a simple path query outputting the two degree neighbors of each vertex. We added an integer group property, starting from 1, on the edges to make the queries faster. The maximum group value varied across experiments. We searched for only the subgraphs in which each edge had a group of 1. We ensured that Neo4j keeps the entire graph in its in-memory cache. For each query, we measured the query runtime and the intermediate number of subgraphs each system generates. Table 2 shows the results. In the table the Tr, D, and P suffixes indicate the triangle, diamond, and path queries, respectively. The numbers in parentheses indicate the number of groups we used in the input graph. As seen in the table, Graphflow generates up to 10.9x less intermediate data than Neo4j and is between 5.8x and 129x faster on our queries. We note that Graphflow generates less intermediate data than Neo4j as a consequence of its vertex-at-a-time strategy instead of Neo4j's edge-at-a-time
Table 4: Continuous query experiments.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>GF-R</th>
<th>GF-T</th>
<th>GF-I</th>
<th>PSQL-R</th>
<th>PSQL-T</th>
<th>PSQL-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soc-t(4)</td>
<td>0.008s</td>
<td>26K</td>
<td>0.12s</td>
<td>1.2s</td>
<td>186</td>
<td>37/5K</td>
</tr>
<tr>
<td>Soc-t(8)</td>
<td>0.13s</td>
<td>183K</td>
<td>0.25s</td>
<td>3.3s</td>
<td>8K</td>
<td>245.1K</td>
</tr>
<tr>
<td>LJ-t(4)</td>
<td>0.05s</td>
<td>21K</td>
<td>0.12s</td>
<td>15s</td>
<td>42</td>
<td>90.5K</td>
</tr>
<tr>
<td>LJ-t(8)</td>
<td>4.5s</td>
<td>125K</td>
<td>0.85K</td>
<td>14s</td>
<td>847</td>
<td>723.3K</td>
</tr>
</tbody>
</table>

4. DEMONSTRATION SCENARIOS

Our demonstration illustrates three aspects of Graphflow: (1) an overview of the system; (2) the ease of development of a Graphflow application; and (3) Graphflow’s GJ and DeltaGJ plans.

4.1 Graphflow Overview

Our first scenario aims to give an overview of an active graph database and the continuous graph applications that can be built on it. Prior to the conference, we will prepare a graph database that contains as nodes, the authors of SIGMOD 2017 papers, and as edges, their Twitter followers. Nodes will have optional email addresses as a property. We will build two applications:

- MagicRecs: Continuously detects diamonds and sends follower recommendations as emails to the attendees who visit our demo.
- SIGMOD Cliques: Continuously finds a clique of 4 users and shows the most recent SIGMOD clique on our monitor.

We will grow our preloaded database continuously by adding each attendee that visits our demo. We will insert the attendee’s Twitter username and email address to a browser UI, which will add the attendee and her followers to Graphflow using Twitter’s API. Upon this insertion, our application will send her an email based on the diamonds she participates in. If she is part of a 4-clique, our monitor will refresh to show her clique as the most recent SIGMOD clique. We will determine and ask her to follow certain users that lead to new diamonds that she is part of. She will get new recommendations soon after she follows those users.

Throughout our interaction, we will give the attendee an overview of our graph store, Cypher++, language, and query processors.

4.2 Graphflow Application Development

Our next scenario demonstrates the ease of application development with Graphflow. The attendee will develop from scratch the financial fraud detection application from Section 1. A typical development cycle will be as follows:

1. Using Graphflow’s console, the attendee will create a transactions graph that contains customer nodes and transaction edges.
2. The attendee will write a UDF in Java that processes cycles. In the UDF, the attendee can choose among possible actions to take, including printing the cycle on the screen, sending emails or text messages to someone, or issuing further queries to Graphflow to delete a transaction in the cycle.
3. The attendee will finally write the Cypher++ trigger to detect the emergence of a cycle of a small size (e.g., 3 or 4) and, as action call the UDF she implemented in the previous step.

Once the application is built, we will test the application as follows. Prior to the conference, we will build a UI that contains a synthetic transactions graph. Initially all of the edges will be colored gray indicating the edge has not been inserted to the database. When the attendee clicks on an edge, the edge will be added to the database and turn green. Once the attendee forms a cycle, she will observe that the action she implemented is taken.

4.3 GJ and DeltaGJ Plans

The goal of this scenario is to demonstrate Graphflow’s GJ and DeltaGJ query plans. The attendee will write a one-time or continuous subgraph query and see Graphflow’s default GJ or DeltaGJ plan for the query visually on PV (as in Figure 3). We will explain the different operators, graph versions, and variable orderings in the plan. To demonstrate the effects of variable orderings, the attendee will type a new ordering on PV that will override Graphflow’s default plan and generate a new plan. The attendee will execute different plans on one of the real-world graphs from Table 1 by clicking a button on PV. For continuous queries, edges of the graph will be streamed at the background using a script we have prepared. The attendee will observe the performances of different plans through the statistics on PV about run-time, intermediate prefixes generated, and throughput. By executing a fixed plan on different scale inputs, the attendee will also test Graphflow’s scalability.

5. REFERENCES