GraphWrangler: An Interactive Graph View on Relational Data

Nafisa Anzum, Semih Salihoglu, Daniel Vogel
University of Waterloo
nanzum@uwaterloo.ca,semih.salihoglu@uwaterloo.ca,dvogel@uwaterloo.ca

ABSTRACT

Existing data stores of enterprises are full of connected data and users are increasingly finding value in performing graph querying, analytics and visualization on this data. This process involves a labor-intensive ETL pipeline, where users write scripts to extract graphs from data stored in legacy stores, often an RDBMS, and import these graphs into a graph-specific software. We demonstrate *GraphWrangler*, a system that allows users to connect to an RDBMS and within a few clicks extract graphs out of their tabular data, visualize and explore these graphs, and automatically generate scripts for their ETL pipelines. GraphWrangler adopts the predictive interaction framework and internally uses a data transformation language that is a limited subset of SQL. Our demonstration video can be found here: https://youtu.be/k92Qk6vuIsU.

1 INTRODUCTION

Graphs are one of the most natural data structures to represent connected data appearing in a wide range of application domains, such as social networks, the web, communication networks, and finance. A recent user survey [7] we conducted across users of graphs revealed, among others, three surprising facts about how graphs are used in practice:

(i) Business transaction data, e.g., representing products and orders, that are traditionally associated with the relational model are also commonly modeled as graphs.
(ii) Graph visualization is a popular task performed by users primarily for data exploration, debugging, and a presentation tool within the enterprise. This process often involves a cumbersome ETL pipeline, where the data from an RDBMS or other data sources are transformed into a graph in the format of a software with a graph visualization component.
(iii) Graph management and processing software are rarely the main system of record. In all of our in-person interviews, the graphs stored in a graph processing software were replicated and extracted from an RDBMS or another store, e.g., a key value store.

These observations indicate that graphs are often another view of already existing connected data stored in other legacy formats and software. Adopting the terminology of "wrangling" [5], used for extracting tables from other sources, we refer to the above ETL processes as *graph wrangling*.

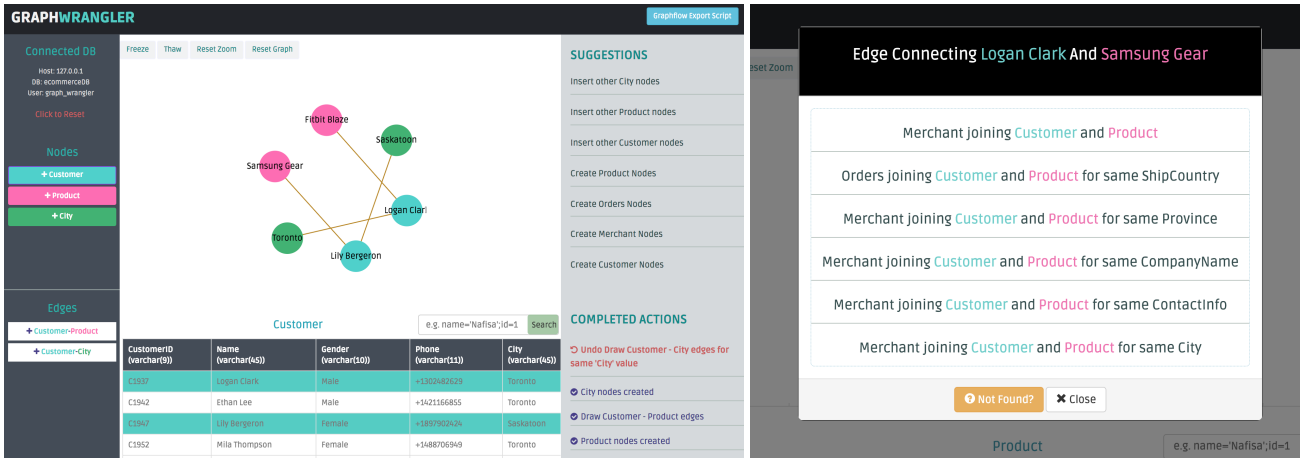
We developed an interactive system called *GraphWrangler* (GW) that easily allows users to wrangle graphs out of relational databases. Figure 1a shows the main interface of GW. Users connect to an RDBMS and start with a tabular view of their data. Then, through visual interactions, such as dragging and dropping of rows or columns, users create a set of nodes and edges. Through several clicks and in the order of several seconds, users can: (i) Extract a small graph and visually explore their data as graphs. Interestingly, we demonstrate how this functionality also allows users to answer *cuboid* and *crossboid* queries from graph OLAP literature [8] directly from their tabular data. (ii) Use an automatically generated script to streamline importing large extracted graphs into a graph-specific software.

Users' actions are internally expressed as rules that transform tabular data into a node, edge, or a property on a node or an edge. These rules are expressed in the system's *data transformation language* which is a subset of SQL that consists of select, join, and aggregation queries. Actions that create nodes and properties map to a single transformation rule. For creating edges, GW adopts the *Predictive Interaction* framework [2]: users draw an edge between two nodes on the UI and the system makes predictions about the rule connecting the nodes. For example, two nodes representing customers Alice and Bob could be connected because they bought the same item or they live in the same city. Internally, these rules correspond to different ways to join the tuples that Alice and Bob were created from. These rules are presented to the user in a human-readable format and the user selects one of the rules.

GW currently supports wrangling graphs out of MySQL and generates scripts to import graphs to Graphflow [6], a prototype graph database we are developing at University of Waterloo. This paper gives an overview of GW, its transformation language, and how it predicts and ranks rules.

2 GRAPHWRANGLER SYSTEM

Wrangled graphs in GW consist of a set of nodes, edges, and key-value properties on nodes and edges. Throughout, we assume a database consisting of relations $R_1, ..., R_n$, and $cols(R_i)$ are the columns in R_i's schema. Our running examples use three relations: $\texttt{Product}(\texttt{P}_{\texttt{ID}}, \texttt{P}_{\texttt{Name}})$, $\texttt{Customer}(\texttt{C}_{\texttt{ID}}, \texttt{C}_{\texttt{Name}}, \texttt{City})$, and $\texttt{Order}(\texttt{O}_{\texttt{ID}}, \texttt{C}_{\texttt{ID}}, \texttt{P}_{\texttt{ID}}, \texttt{Amount})$.



(a) Main interface. **(b) Edge type predictions.**

Figure 1: GraphWrangler interfaces.

2.1 Wrangling Nodes and Node Lineage

Users can create a node by highlighting one or more cells of a single row (possibly the entire row) in a single table and dragging and dropping the cells on GW's node-link panel. This action draws a single node v on the panel, which is internally associated with three pieces of information:

(i) *Node Type*: Each node v is of a particular *node type* NT_k, which is a pair $(R_i, cols(NT_k) \subseteq cols(R_i))$ indicating the R_i and R_i's columns that v was wrangled from. We use $v.rel$ and $v.cols$ to refer to R_i and $cols(NT_k)$, respectively.

(ii) *Values*: This is a tuple $v.vals = (val_1, ..., val_t)$, where $t = |v.cols|$, indicating the values of $v.cols$ identifying v.

(iii) *Lineage*: v's lineage $v.lin$ is the set of tuples from R_i with the same values as v for $v.cols$, i.e., $v.lin = \sigma_{v.cols=v.vals} R_i$.

Example 2.1: Suppose the user drag and drops an entire `Customer` *row* t_5=(`C5`, `Alice`, `Waterloo`) *to the panel. GW interprets this as the entire row of the* `Customer` *table representing a node* v*, so* v*'s type is* NT_1=(`Customer`, {C_{ID}, C_{Name}, `City`}), $v.vals$=(`C5`, `Alice`, `Waterloo`), *and* $v.lin$=$\{t_5\}$.

Example 2.2: If the user drag and drops only the `Waterloo` *cell in* t_5*, this is interpreted as each unique* `City` *value being a node. So* v*'s type is* NT_2=(`Customer`,{`City`}), $v.vals$=(`Waterloo`), *and* $v.lin = \sigma_{City=Waterloo}$`Customer`*, which includes* t_5 *as well as the other tuples whose* `City` *values equal Waterloo.*

Node types are given a human readable name by the users, e.g., "Customer" or "City" as shown in Figure 1a. If the number of nodes of a particular type are small, users can click a button and extract all nodes of this type and display them in the panel. For example, creating all NT_2 City nodes could add `Kitchener`, `Toronto`, and `Guelph` nodes to the panel.

2.2 Wrangling Edges

Users can create edges that connect nodes by defining an *edge type* which expresses a way to join the tuples in the lineages of the nodes. Users define edge types through a predictive interaction with GW, which is explained momentarily. Formally, an edge type ET_k is an equi-join query Q and connects two nodes v_1 and v_2 if the output of Q is non-empty. Q is formally expressed as follows (jc stands for **j**oin **c**olumn, and in jc_{x_L} and jc_{x_R}, L and R stand for **L**eft and **R**ight):

$$v_1.lin \underset{jc_{v_1}=jc_{1_L}}{\bowtie} R_{i1} \underset{jc_{1_R}=jc_{2_L}}{\bowtie} R_{i2} \cdots R_{it} \underset{jc_{t_R}=jc_{v_2}}{\bowtie} v_2.lin \quad (1)$$

Above: (i) $jc_{v_1} \in cols(v_1.rel)$, $jc_{v_2} \in cols(v_2.rel)$, and $jc_{x_L}, jc_{x_R} \in cols(R_{ij})$; (ii) t can be 0, meaning a direct join between the tuples in $v_1.lin$ and $v_2.lin$ on columns jc_{v_1} and jc_{v_2}; and (iii) R_{ij} are not necessarily distinct relations. Currently edges are undirected but a direction to edges can easily be given e.g., the left node v_1 in Equation 1 could be the source.

Example 2.3: Suppose a user has created a node v_1 *representing customer Alice, with lineage* t_5=(`C5`, `Alice`, `Waterloo`), *and a* v_2 *representing customer Bob, with lineage* t_7=(`C7`, `Bob`, `Waterloo`). *If the user wants to add an edge between* v_1 *and* v_2 *because Alice and Bob are from the same city, the formal edge type would be* $ET_{SameCity} = v_1.lin \underset{City=City}{\bowtie} v2.lin$*, where both* jc_{v_1} *and* jc_{v_2} *are* `City` *and there is no other relation involved in the join (so* t *is 0 in Equation 1).*

Example 2.4: Consider now adding an edge between Alice and Bob because they have ordered the same product. This can be represented by an edge type $ET_{Copurchase}$ *(below* `Cu` *and* `Pr` *are respectively the* `Customer` *and* `Product` *tables):*

$$v_1.lin \underset{Cu.C_{ID}=Pr.C_{ID}}{\bowtie} Pr \underset{P_{ID}=P_{ID}}{\bowtie} Pr \underset{Pr.C_{ID}=Cu.C_{ID}}{\bowtie} v2.lin$$

Predictive Interaction: Users define edge types interactively by drawing an edge between two existing nodes v_1 and v_2, and GW starts searching for different queries that can join $v_1.lin$ and $v_2.lin$. The valid edge type definitions GW finds are ranked and presented to the user in a human readable

way, as shown in Figure 1b. The user then selects one of these predictions to indicate the correct definition.

Searching of Edge Types: GW inspects the schemas of the tables in the database and generates a set of SQL queries that fit the join template in Equation 1. This involves enumerating different column combinations that can join (e.g., have the same data type) in $v1.rel$, $v2.rel$ and other tables. For databases with a large number of tables or columns, this search can be very expensive. To present predictions at interactive speeds, GW takes two steps: (1) GW issues the queries in increasing order of t and presents them as they are found. (2) When the lineages are small in size, GW keeps the lineages in its frontend and performs the search for $t = 0$, i.e., direct joins between $v1.lin$ and $v2.lin$, without querying the RDBMS. This is a common case, e.g., in our first running example both Alice and Bob have single-tuple lineages. GW also converts joins to selections when performing the search for $t = 1$. Suppose our database has a `Friend(`C_{ID_1}`,` C_{ID_2}`)` table indicating the friendships between customers. Consider checking whether or not Alice and Bob can be connected because they are friends. This edge type involves 1 intermediate relation but can be checked with the query: Select * FROM Friends WHERE C_{ID_1}=C5 AND C_{ID_2}=C7, avoiding any joins. These steps allow GW to operate at interactive speeds on large tables, with millions of rows.

Finally, GW currently limits the search to joins involving at most two tables. Users manually specify more complex edge definitions through a separate interface (by clicking the "Not Found?" button in Figure 1b).

Ranking of Edge Type Definitions: Edge type definitions are ranked using three heuristics in this order:

1. Definitions with smaller t values rank higher.
2. Definitions with foreign key relations rank higher. GW reads this information from the database upon start.
3. Definitions, whose join queries have more outputs rank higher, interpreted as stronger links between nodes.

Our current research focuses on more advanced techniques to efficiently search and rank edge definitions, e.g., through sampling and approximate querying techniques.

2.3 Wrangling Properties

GW supports two *node property types* and one *edge property type*. Users define properties through an interface by selecting a column from a drop-down list. For node aggregation and edge aggregation properties an aggregation function is also selected (explained momentarily).

Node Column Properties: Any column value in $v.cols$ can be added as a property, e.g., in our first running example, we can give Alice the properties City=Waterloo or $C_{\texttt{Name}}$=Alice.

Node Aggregation Properties: These properties are the results of aggregation queries on $v.lin$. These are internally expressed as a pair (`key`, γ_A, fn), where `key` is the string of the property, γ_A is an optional numeric-valued column to aggregate, and fn is an aggregation function. The value of the property corresponds to the result of the relational algebra expression $\gamma_{\text{v.cols},fn(\gamma_A)}v.lin$ (so we group by all columns). GW supports count, sum, max, and min functions. The simplest example counts the size of each node's lineage (so translates to a count star query). Consider our example where we wrangled `City` nodes from the `Customer` table. Suppose the table had a `Salary` column. We could add a node property for the total salaries of customers in each city with a node aggregation property (`Total Salary`, `Salary`, *sum*).

Edge Aggregation Properties: These are expressed the same way as node aggregation properties but the values are aggregations on the results of the joins defining the edge types.

2.4 Other Interactions

Users can interact with GW in three other ways. First, users can type keywords or predicates, e.g., "name = Alice", into the search boxes above the tables they are wrangling from (shown in Figure 1a). This allows them to detect specific rows in their tables. Second, users can click on a node v_1 and an "Expand" button, which will wrangle all 1st degree neighbors of v_1 based on the existing node and edge types; further clicks expand the graph to higher degree neighbors. This enables users to quickly put a graph view on their relational data, and visualize and explore their tabular data as a graph. Third, users can automatically obtain a script to import all nodes and edges from the RDBMS to Graphflow [6], for more advanced graph querying. This streamlines the ETL pipeline of transforming data from relational tables into graphs that can be imported to a graph-specific software [7].

2.5 Implementation

GW runs as a web application built on the AngularJS framework. A NodeJS backend supports the interactions with MySQL. The web application communicates with the backend through REST APIs. We use D3.js [1] for visualizing graphs, and interact.js [4] to capture drag and drop actions.

3 DEMONSTRATION SCENARIOS

3.1 Fraud Detection Through Exploration

Our first demonstration scenario demonstrates: (1) the overall experience of wrangling graphs with GW; and (2) the value users can get from putting a graph view on their tabular data in the matter of a few clicks. The scenario involves a fraud report about a customer Alice buying a smartwatch Tactix Bravo on an e-commerce website. To investigate the

report, a fraud analyst launches GW to connect to a transactional database with the `Product`, `Customer`, `Order` tables from our running example and a new `Merchant`(M_{ID}, M_{Name}) table (the merchants selling products on the e-commerce site). The `Product` table has a new M_{ID} column to indicate the merchant producing the product. We generate synthetic data with 100K products, 100K merchants, 1M customers, and 10M orders and plant a bipartite clique into the data: a set of customers C and products P, where each customer in C has ordered each product in P. This is a popular fraudulent pattern to increase a merchant's rankings on the site [7]: a merchant m pays a set of customers C to initiate fake transactions to buy a set of products P from m.

We use GW to connect to the RDBMS, find Alice's row using the search box on the `Customer` table and drag the row to create a `customer` node for Alice. After similar steps we create the product node for Tactix Bravo. Next, we draw an edge between the nodes and go through a predictive interaction step and confirm GW's suggestion that the nodes are connected because of an order Alice has made to buy Tactix Bravo. We start exploring the neighborhood of Alice: the first degree neighbors show all other products Alice has bought, bringing products P into the visualization. The second degree neighborhood brings C into visualization, making the bipartite clique visible. Suspicious of this pattern, we find the merchant who sells one of the products in P, creating the merchant m as a node. Exploring the neighbors of m demonstrates m sells all of the products in P.

3.2 Graph OLAP Queries

Our second scenario demonstrates: (1) adding properties on the nodes and edges; and (2) interactively answering cuboid and crossboid queries. The *Graph Cube* system [8] takes as input a graph G and constructs aggregate views of G. This happens by creating "supernodes" that group nodes according to a set of properties, and aggregating the edges between the nodes into "superedges". Cuboid queries construct one type of supernodes and generate a unipartite aggregate graph; crossboid queries construct two types of supernodes and generate a bipartite graph. Consider a call graph G consisting of: (i) `phone` nodes with two properties: `gender` and `province`, indicating the owner's gender and province, respectively; and (ii) `call` edges between `phones` with a `duration` property. A cuboid query can generate `province` supernodes with edges between them that contain a `total-duration` property, indicating the total duration of calls between each province. A crossboid query can generate `gender` nodes on the left side and `province` nodes on the right with the same `total-duration` edges between them.

Our scenario demonstrates how to answer these queries directly from tabular data consisting of `Phone` and `Calls` tables whose columns store the same properties as above. For the cuboid query, we create `province` nodes by dragging the `Province` column of the `Phone` table. We then draw an edge between, say Ontario and Quebec nodes, and introduce edges indicating the calls made between phone numbers from Ontario and Quebec. Then we create a `total-duration` property on the created edges, and pick the `duration` column to aggregate, which answers the first cuboid query above. We repeat the same process by creating both `gender` and `province` nodes to answer the crossboid query above.

3.3 Six Degrees of Kevin Bacon Test

Our third scenario demonstrates: (1) how GW streamlines the ETL pipeline of extracting graphs from RDBMSs and importing them to a graph-specific software for advanced graph querying; and (2) our data transformation language. We wrangle a "Co-stars" graph of movie actors from the IMDb dataset of movies [3]. We create an actor node a and a movie node m, and expand the neighborhood of m to create the rest of the actors in m. Then by connecting two of the actors, we introduce a `co-star` edge and add a property on this edge for the number of movies the two actors have co-starred in. We then click on the "Export to Graphflow" button to obtain the script that GW generates, which contains the SQL statements describing the node, edge, and edge property types we defined. Then we run the script, generate a set of files in Graphflow's format, and import the graph to Graphflow. In Graphflow, we run several advanced graph queries on our graph: shortest path queries between different actors and Kevin Bacon, and subgraph queries to find cliques of actors who have all co-acted with each other.

REFERENCES

[1] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven Documents. *TVCG*, 17(12), 2011.

[2] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive Interaction for Data Transformation. In *CIDR*, 2015.

[3] Imdb datasets. https://www.imdb.com/interfaces/.

[4] interact.js. http://interactjs.io/.

[5] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*, 2011.

[6] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

[7] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. https://cs.uwaterloo.ca/~ssalihog/papers/graph-survey-extended.pdf, 2019.

[8] P. Zhao, X. Li, D. Xin, and J. Han. Graph Cube: On Warehousing and OLAP Multidimensional Networks. In *SIGMOD*, 2011.