Generating Example Graphs for Gremlin Queries

JACKSON NEAL, Northeastern University, USA

Query development for data-centric programs is an often iterative process requiring investigation into query behavior during execution. Executing queries directly on real data may not be a viable option due to security concerns, incomplete existing data, long query runtime, or slow data ingestion. Creating mock data is a tedious exercise that leaves open questions as to the full behavior of queries.

Graph databases have become increasingly useful to the world of big-data due to their flexible schema, easily understood object-relational mapping, and index-free adjacency. These characteristics make graph databases prime candidates for recommendation systems, fraud detection, and classic graph algorithms such as path finding. Graph databases have shown particular promise with queries that require deep traversals and would incur expensive join costs in a relational database.

This report introduces a tool that interprets graph database queries and produces example graphs that completely and minimally demonstrate query behavior. The tool is available as a web application and can be essential to understanding and demonstrating query behavior for both validation and learning. To the best of our knowledge, methods for generating example data for graph database queries are non-existent. However, graph database queries can often be expressed as dataflow programs (e.g. K-hop paths), allowing us to extend concepts from the 2009 Yahoo research paper "Generating example data for dataflow programs" [12]. We show that adapting the techniques from [12] to produce example graphs instead of example records in tables more clearly and concisely demonstrates graph query behavior. This paper also enhances the approach from [12], introducing methods to handle complex filters and cycles which can be applied to both the graph queries explored here and the dataflow programs in [12].

Additional Key Words and Phrases: graph databases, data generation, query processing

1 INTRODUCTION

The need for a tool to efficiently and effectively demonstrate graph query behavior is evident through the rise in prevalence of graph databases. Graph databases are particularly popular in applications seeking to identify and analyze clusters, partitions, shortest paths, recommendations, social networks, and knowledge graphs. Applications that utilize graph databases are often dealing with large amounts of sensitive data. These conditions lead to several hurdles when attempting to develop queries for graph databases. User data may have security requirements and be unavailable for testing purposes. Obfuscating data poses its own challenges and leads to difficulty in evaluating results. Graph databases may grow to billions of nodes, requiring non-trivial sampling for test usage. In many applications, relevant data may not yet be ingested or prototypes may be built before any data is available. A tool that facilitates the demonstration of graph database query behavior could be indispensable to developers and corporations.

Faced with these difficulties, developers often find themselves left with the option of creating synthetic data to test and demonstrate their queries. Manufacturing synthetic data is an intensive task that has changing requirements as new queries are introduced. Still, synthetic data provides the opportunity to clearly demonstrate all aspects of a query's behavior while avoiding the challenges of using real data. Automation of the creation of synthetic example graphs for graph database queries is the key objective of this project. A good solution to this problem will alleviate the responsibility of creating example graphs from developers and provide immediate feedback on the behavior of queries. The tool should be easily accessible and the graphs produced should be concise and complete in their demonstration of query behavior. The tool should be able to dynamically generate example graphs for unseen queries in near-real-time (NRT).

Author's address: Jackson Neal, neal.ja@northeastern.edu, Northeastern University, Boston, Massachusetts, USA, 02115.

1.1 State of the Art

There is extensive literature on the generation of example data for demonstrating behavior of relational database queries. However, none of the existing approaches are applied to the problem of generating example graphs for graph database queries and, to the best of our knowledge, no such solution exists. Graph database queries can often be expressed as dataflow programs and the state of the art for generating example data for relational dataflow programs comes from [12]. In this report, we introduce a method of producing example graphs for graph database queries that builds off the existing relational database techniques proposed in [12] while more effectively demonstrating query behavior.

1.2 Illustrative Example

Figure 1 illustrates an example People You May Know (PYMK) graph query with a generated example graph. The query is used to recommend friends who are connected to an initial user by a two-hop path, who are not connected to the initial user by a one-hop path, and who are not the initial user. In the following sections we will explain how the example graph is produced for the PYMK query. When considering the execution of a graph database query, it is helpful to think of the query as moving a set of traversals over the graph. At any step of the query's execution, traversals may be moved, altered, created, or deleted. During execution, we refer to the location of a traversal as the traversal *head*. By analyzing traversal behavior during execution, we can determine if query behavior is completely demonstrated by an example graph. Table 1 identifies how traversals behave in the PYMK query, showing the names of the vertices that have traversals at each step.

Figure 2 provides a dataflow program equivalent to the PYMK graph query. The dataflow program treats edges and vertices as records and displays the passing of example data through the program operators using the same visual paradigm as [12]. It is important to note that the example records shown in Figure 2 could not be produced from the process proposed in [12]. We will later explore how our contributions could be applied to [12] and achieve the Figure 2 result. It is also worth noting that [12] does not support `AS`, `AGGREGATE`, or `GROUPCOUNT` operators but could be extended to do so.

These figures highlight that we are solving the same problem as [12] but in a completely new environment. Comparing the two figures, we see that using graphs instead of dataflow diagrams can provide a more concise representation for easy understanding of graph query behavior.

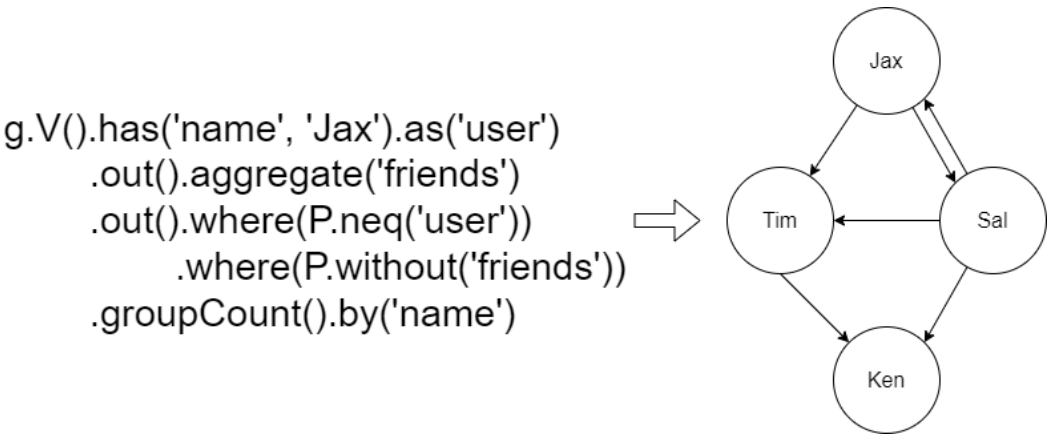

Fig. 1. PYMK Gremlin query and generated example graph

Step	Location of Traversals
`g.V()`	Jax, Tim, Sal, Ken
`.has('name', 'Jax')`	Jax
`.as('user')`	Jax
`.out()`	Tim, Sal
`.aggregate('friends')`	Tim, Sal
`.out()`	Jax, Tim, Ken, Ken
`.where(P.neq('user'))`	Tim, Ken, Ken
`.where(P.without('friends'))`	Ken, Ken
`.groupCount().by('name')`	-

Table 1. PYMK traversal behavior during execution

1.3 Contributions

In addition to the novel application of existing methods to graph databases, this report offers several improvements that are critical to the creation of example graphs and could in-turn be applied to improving example data generation in relational dataflow programs.

Importantly, our approach does not assume any existing data. This makes our tool immediately relevant to a wide range of applications that may not have access to real data for reasons explored above. Removing the use of real data allows us to focus on creating the best example graphs possible. We are also able to avoid the sampling and extra pruning that is incurred when incorporating real data into examples as seen in [12].

Our approach introduces a constraint language to facilitate lazy evaluation of values. This lazy evaluation allows single elements to satisfy multiple constraints, improving the completeness of generated examples.

Our process also replaces the pruning algorithm from [12] with a novel graph reduction and merging algorithm. Merging allows us to greatly improve upon the conciseness of examples that would be produced from [12]. Our proposed algorithm is also essential for capturing real-world graph behavior such as cycles.

2 PROBLEM DEFINITION AND NOTATION

Formally, the tool introduced by this report is capable of taking a graph database query Q as input and generating an example graph G as output. Q is a query in the TinkerPop Gremlin language composed of l traversal steps. G is composed of a set of vertices and edges that can be displayed to the user.

Our tool supports a subset of steps from the extensive Gremlin language:

- `V()`: Create traversals at all vertices in the graph.
- `out()`: Move traversals to neighbor vertices by following outgoing edges. Traversals on nodes with no outgoing edges are terminated.
- `has(key, value)`: Eliminate traversals that do not have the specified *key* with *value* in properties.
- `where(predicate)`: Eliminate traversals that do not satisfy the passed predicate. Predicates take the form `P.eq(var)`, `P.neq(var)`, and `P.without(P.without(var))` where *var* is defined in the context of the query or traversal.

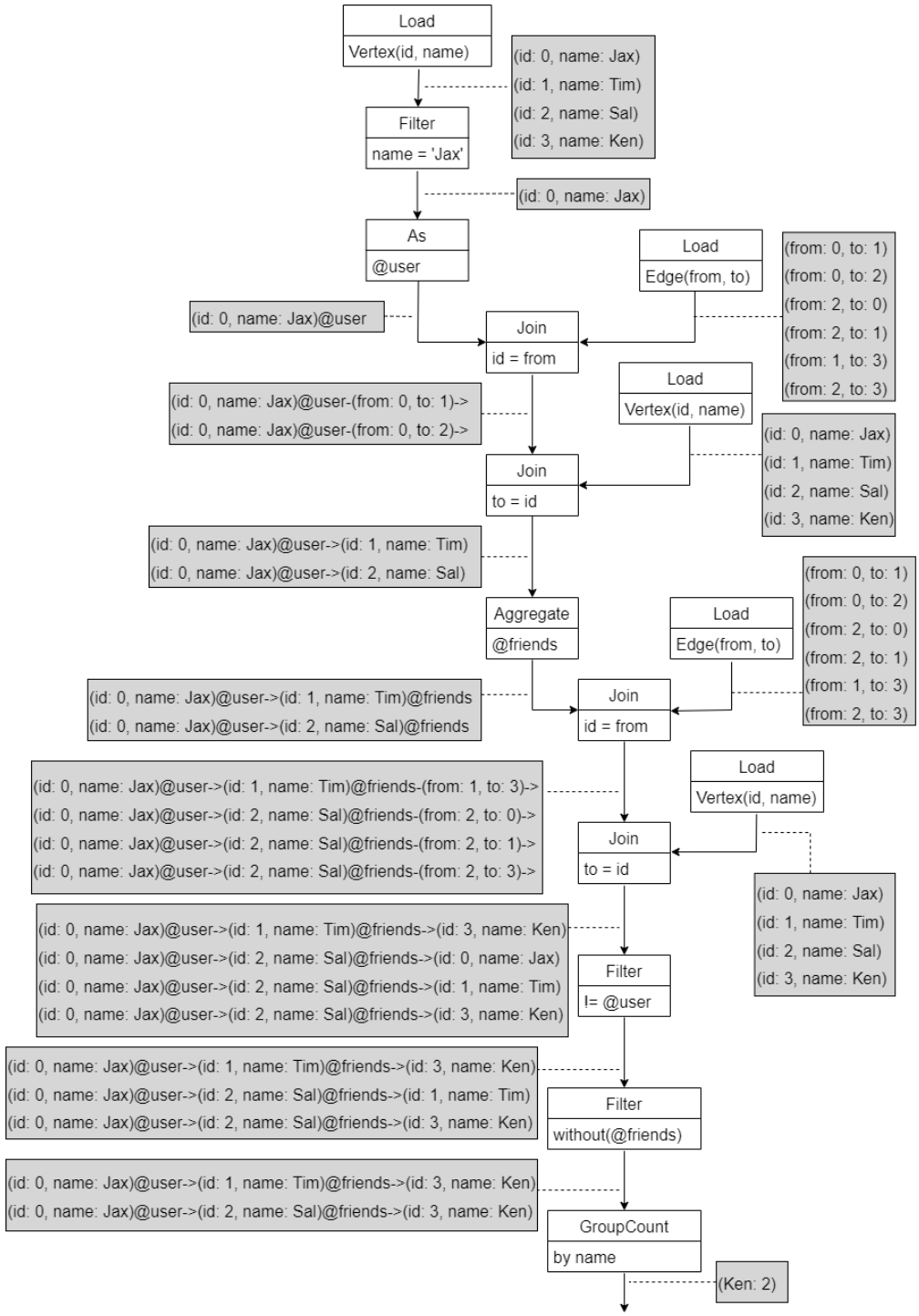

Fig. 2. PYMK as a dataflow program with example records

- `as(label)`: Create a traversal variable *label* referencing the object at the head of the traversal.
- `aggregate(label)`: Create a traversal variable *label* referencing the set of all objects at the head of traversals.
- `groupCount()`: Group traversals whose head is at the same object, resulting in a map of graph elements to counts.

2.1 Equivalence Class Model

We adopt the Equivalence Class Model from [12], clarifying some details in the context of graph queries and traversal execution. Simply stated, each Gremlin query step O has a set of equivalence classes "$\varepsilon_O = \{E_1, E_2, ..., E_m\}$" where each equivalence class represents a behavior of the step [12]. As query steps are executed, traversals are assigned to one or zero of the equivalence classes of the current step. In order to completely demonstrate step behavior, each equivalence class must have at least one member. Complete equivalence class definitions for our supported Gremlin steps can be found in Appendix A.

2.2 Objectives

Our tool prioritizes the same completeness and conciseness metrics as [12], but since our tool does not expect existing data, we do not consider the realism metric from [12]. Our project aims to produce a final product that is easily accessible with NRT graph generation for unseen queries.

Completeness of an example graph is defined as the per step average completeness for the Gremlin query. Step completeness is the fraction of step equivalence classes that are satisfied during execution.

Conciseness of an example graph is defined as the per step average conciseness for the Gremlin query. Step conciseness is determined through the ratio of the number of step equivalence classes to the number of traversals that pass through the step during execution.

3 APPROACH

The process of generating example graphs for graph database queries occurs in three major steps: query parsing, upstream graph generation, and graph reduction and merging. The latter two steps are heavily inspired by the approaches in [12] with key innovations to improve performance on graph queries.

3.1 Query Parsing

Out tool utilizes ANTLR (ANother Tool for Language Recognition) for parsing of Gremlin graph database queries [13]. The Gremlin language has an open source ANTLR grammar which we modify to include only traversal steps that our tool supports. Figure 3 demonstrates how we feed our grammar, *GremlinGx.g4*, to ANTLR, generating parser classes necessary for processing our query input as text. The generated parser includes a visitor interface that allows us to handle query steps as they are parsed. After parsing, we have a complete tree of Gremlin steps, similar to the dataflow diagrams in [12], that will allow us to generate our example graph in the subsequent upstream step.

3.2 Upstream Generative Pass

After query parsing, we perform an upstream pass from the root operator, propagating constraint records using the same procedure as [12]. In the graph context, a constraint record represents traversals that must exist during query execution. More specifically, a constraint record lays out

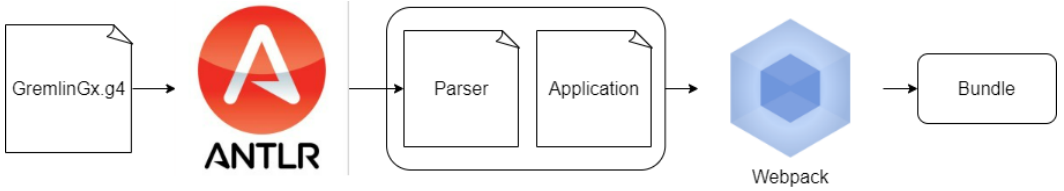

Fig. 3. GremlinGx application build process

a path through nodes and edges that must exist during execution. These constraint records are visualized in the intermediate tables of Figure 2. Operator specific upstream propagation rules can be found in Appendix A.

3.2.1 Lazy Constraint Evaluation. In order to improve completeness of the generated example graphs, we introduce the concept of lazy constraint evaluation. Avoiding generating concrete field values until the last possible opportunity allows elements to satisfy multiple constraints, which is particularly relevant when queries apply multiple filters across independent steps. This strategy improves on the approach of immediately filling in field values with sampling from existing distributions [12].

Take the Gremlin query `g.V().has('name', P.neq('Jax')).has('name', P.neq('Tim'))`. Here, we are querying vertices, filtering to those not named `Jax` and then filtering to those remaining that are not named `Tim`. On the left of Figure 4 we can see the upstream pass using immediate field resolution. Each filter takes downstream elements as input and produces a passing and failing element upstream. In this example, it is clear that if we then passed these elements downstream, only one element would reach the second filter and we could not completely demonstrate its behavior.

On the right of Figure 3, we can see the employment of lazy constraint evaluation. We are encoding the characteristics of fields instead of actual values. This allows us to completely demonstrate all filter behavior in the downstream pass.

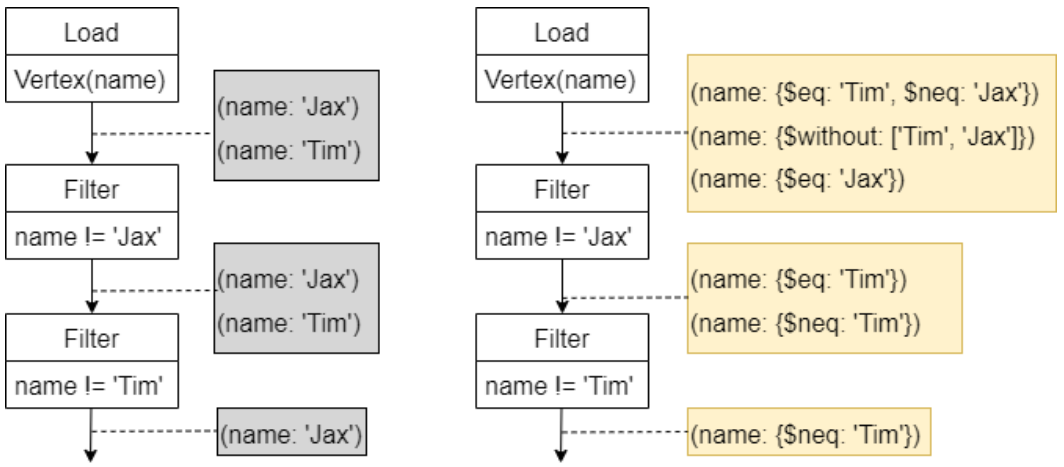

Fig. 4. Upstream lazy constraint evaluation
`g.V().has('name', P.neq('Jax')).has('name', P.neq('Tim'))`

3.3 Reduction and Merging

After we complete the upstream pass, we have generated a complete but non-minimal graph. Consider the query `g.V().out().has('name', 'Jax')`. After the upstream pass, this query would correspond to generated dataflow diagram and example graph seen in Figure 5. It is evident that these examples completely demonstrate the query behavior. However, we can also easily picture a more concise demonstration of query behavior as seen in Figure 6.

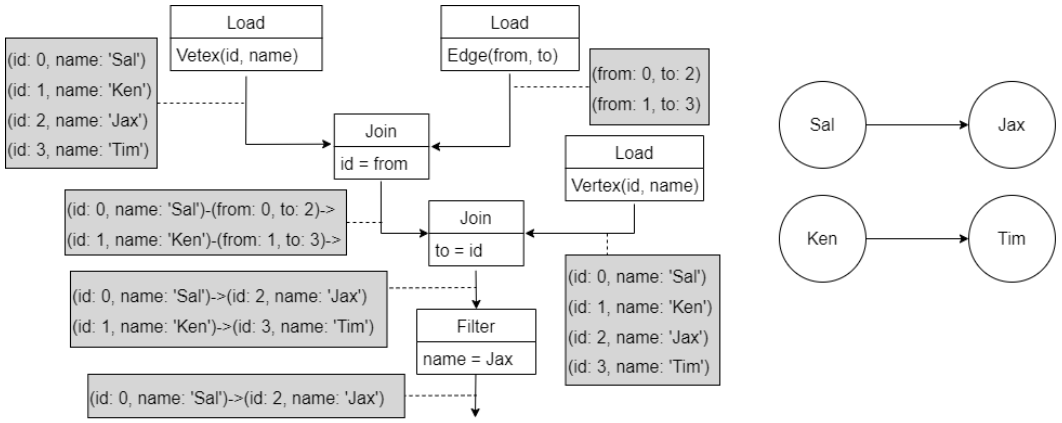

Fig. 5. Non-minimal generated dataflow records and example graph
`g.V().out().has('name', 'Jax')`

Fig. 6. Minimal example graph
`g.V().out().has('name', 'Jax')`

The pruning algorithm from [12] uses a greedy approach similar to a solution for the classic weighted set-cover problem. Their algorithm defines a lineage as a record and all records that contributed to its formation (e.g. a `JOIN` output record is in the same lineage as the records that were joined as input). The algorithm then chooses lineages from the upstream generated records and adds them to a reduced final example set.

The idea of lineages has two issues in the graph domain. First, due to the high amount of joins - think edge traversals - in graph queries, there are often very few independent lineages and we cannot sacrifice any of them without reducing completeness. This is especially true in our generated example graphs, since we are not sampling from existing data and we avoid generating excessive lineages in our upstream pass. This is evident in Figure 5 where no record and its lineage could be removed and maintain the same completeness. Second, as records are generated in a upstream fashion, no record is ever reused. This is required to guarantee completeness during the upstream pass but means that a lineage will never contain the same record twice. This results in generated example graphs that have no cycles, which is unrealistic for real-world examples. In order to gain cyclic behavior and better improve conciseness we need to consider an algorithm that incorporates merging of individual records.

In order to facilitate graph reduction without removing entire lineages and allow the existence of cycles, we introduce a merging algorithm to combine graph entities. In the upstream generated graph, there are $O(C(n, 2))$ possible vertex merges available, where n is a function of the number of query steps, l, since each query step adds some number of vertices during the upstream pass. We can immediately eliminate many of these due to conflicting constraint records; `(name: 'Jax')` cannot merge with `(name: 'Sal')`. For all valid merges, we execute the merge and then score the resulting candidate graph. If the candidate graph does not improve completeness or conciseness, the candidate is discarded. Otherwise the candidate is stored and we repeat this process for all remaining candidates. The candidate with the maximum completeness score, using conciseness as a tie-breaker, is the final reduced graph. Algorithm 1 lays out this procedure in detail. In practice, this algorithm runs in NRT for queries with few steps, generating example graphs as the user types. For queries with many steps, this algorithm may incur performance issues and can be altered to greedily explore the frontier similar to the greedy pruning of [12], at the consequence of lost correctness.

```
upstreamGraph ← upstreamPass(stepTree);
maxCompleteness ← 0;
maxConciseness ← 0;
frontier ← [upstreamGraph];
candidates ← [];
while frontier.length ≠ 0 do
    nextFrontier ← [];
    foreach candidate ∈ frontier do
        completeness ← downstreamCompleteness(candidate);
        conciseness ← downstreamConciseness(candidate);
        if completeness < maxCompleteness ∨ (completeness ==
          maxCompleteness ∧ conciseness < maxConciseness) then
            continue;
        else
            maxCompleteness ← completeness;
            maxConciseness ← conciseness;
        end
        candidates.push(candidate);
        foreach pair ∈ combination(candidate.vertices, 2) do
            if canMerge(pair) then
                reduced ← candidate.copy().merge(pair);
                nextFrontier.push(reduced);
            end
        end
    end
    frontier ← nextFrontier;
end
```

Algorithm 1: Graph reduction through merging

3.4 Technical Design

The objectives of easy access and NRT example generation have driven the technical design decisions for this tool. The tool exists as web application with no reliance on external services. The source code consists of HTML, CSS, JavaScript, and TypeScript. TypeScript was chosen for its optional typing system that facilitates the development of complex application logic while also giving the flexibility necessary when writing a parser for generic types. ANTLR provides the ability to quickly generate JavaScript parsers to handle our Gremlin language subset entirely in the frontend. Our application uses Cytoscape.js for sleek, dynamic graph visualization [7]. We utilize Webpack to generate a minimal application bundle for deployment [1]. The project repository is available on GitHub [2].

4 EMPIRICAL EVALUATION

Experimental results show our tool generating minimal graphs that completely demonstrate query behavior. Table 2 shows the results of 2-hop paths, Triangle Search, and PYMK queries with data generation through our tool. We do not show results from [12], but we know, due to lack of cycles and merging techniques, the results would be less complete than what we achieve. We only show a select set of interesting queries here, but our tool can handle any combination of our supported steps and produce graphs with minimal latency.

Query	GremlinGx	
	Completeness	Conciseness
`2-hop paths`	1.0	0.7381
`Triangle Search`	1.0	0.7222
`PYMK`	1.0	0.7417

Table 2. Select Gremlin query performance results

5 RELATED WORK

Generating test data for relational queries first appears in [11], introducing the idea of synthetic databases to sufficiently demonstrate query operation effects while also doing so minimally. Since then, many different facets of the problem of generating test databases have been explored. [5] proposes methods for generating example data with constraints on the content of intermediate results. [6] introduces methods for data generation with the goal of exposing bugs within Map-Reduce programs. As we have explored, [12] proposes methods of realistically, minimally, and completely demonstrating the behavior of high-level dataflow programs written in Pig [8]. [10] expands the work of [5] and the scope of the problem to consider a set of queries and a schema to produce databases with desired traits. The "workload-aware" approach in [10] facilitates stress testing and application benchmarking, beyond minimal, complete demonstrations. [2] introduces methods of data generation through declarative constraints, applicable in scenarios of DBMS testing, data masking, and benchmarking. [14] describes techniques for table and value generation of parameterized SQL queries through a satisfiability solver. Although these papers only explore relational DBMS, the issues faced and the proposed techniques are still applicable if translated to graph database queries and example graphs. To our knowledge, there is no literature regarding generating example graphs in a query-drive manner. Instead, current approaches focus on randomized algorithms to produce graphs with certain characteristics given domain-specific requirements [1] [9].

[1] https://jacksonneal.github.io/gremlin-gx/dist/index.html
[2] https://github.com/jacksonneal/gremlin-gx

6 CONCLUSIONS AND FUTURE WORK

The tool covered in this report successfully applies the approaches of [12] to graph database queries in order to generate example graphs that more clearly demonstrate query behavior than tables. The methods proposed do not require existing data, and the tool produced is relevant for established or prototype graph database applications. Our process employs lazy constraint evaluation and a novel merging algorithm to improve on completeness and conciseness of results from [12]. The application accepts a subset of the official Gremlin language and is easily extendable to include more query steps in the future. Our final product is available as a lightweight web application for easy access.

Future work may expand to more steps in the Gremlin language or expand to other graph query languages such as SPARQL. Such expansion into advanced query steps would be facilitated through exploration of language theory and compilation techniques, including current methods of query simplification in relational DBMS [4]. Another direction of future work may improve the theoretical basis of the merge algorithm introduced here and explore more comprehensive constraint satisfaction approaches as seen in relational DBMS data generational approaches [10] [14]. We may also explore avenues of machine learning to generate images of example graphs directly from the queries, using ground-breaking NLP techniques such as "Paint by word" [3].

ACKNOWLEDGMENTS

Thanks to Arthur Bigeard for introducing me to ANTLR and Cytoscape.js.

REFERENCES

[1] William Aiello, Fan Chung, and Linyuan Lu. 2001. A random graph model for power law graphs. *Experimental mathematics* 10, 1 (2001), 53–66.

[2] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 685–696.

[3] David Bau, Alex Andonian, Audrey Cui, YeonHwan Park, Ali Jahanian, Aude Oliva, and Antonio Torralba. 2021. Paint by word. *arXiv preprint arXiv:2103.10951* (2021).

[4] Michael Benedikt. 2018. How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)?. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 1–15.

[5] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.

[6] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. 2011. New ideas track: testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 504–507.

[7] Max Franz, Christian T Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D Bader. 2016. Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (2016), 309–311.

[8] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1414–1425.

[9] S Jamin and J Winick. 2002. Inet-3.0: Internet topology generator. *University of Michigan, Ann Arbor* (2002).

[10] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating databases for query workloads. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 848–859.

[11] Heikki Mannila and Kari Jouko Raiha. 1985. Test data for relational queries. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 217–223.

[12] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 245–256.

[13] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[14] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446.

A GRAPH GENERATION METHODS

Here we provide specifics on the equivalence classes and upstream propagation rules for Gremlin graph query steps that allow us to generate example graphs in a manner similar to the algorithm introduced in [12].

A.1 Equivalence Class Definitions

- `V()`: All traversals with vertices at the head are placed in equivalence class E_1.
- `out()`: All traversals with vertices at the head with outgoing edges are placed in equivalence class E_1.
- `has(key, value)`: All traversals with entities at the head with the specified *key* with *value* in properties are placed in E_1; all others are assigned to E_2.
- `where(predicate)`: All traversals with entities at the head that satisfy the given predicate are placed in E_1; all others are placed in E_2.
- `as(label)`: All traversals with entities at the head with the given *label* are assigned to E_1.
- `aggregate(label)`: All traversals with entities at the head with the given *label* are assigned to E_1.
- `groupCount()`: Traversals that are grouped with other traversals are placed in E_1. This ensures at least one successful grouping.

A.2 Upstream Propagation

- `V()`: If there are no traversals, one is created. The head of all traversals are marked as vertices.
- `out()`: If there are no traversals, one is created. The head of all traversals are marked as vertices. An edge is created from a new vertex to the previous head. The new vertex becomes the head of the traversal.
- `has(key, value)`: Attempt to modify all traversal head entities to have the given *key* with *value* as properties. After modification, if no passing traversals exist, manufacture one. If no failing traversals exist, manufacture one.
- `where(predicate)`: Attempt to modify all traversal head entities to pass the given predicate. After modification, if no passing traversals exist, manufacture one. If no failing traversals exist, manufacture one.
- `as(label)`: If there are no traversals, one is created. Attempt to label all traversal head entities with the given *label*.
- `aggregate(label)`: If there are no traversals, on is created. Attempt to label all traversal head entities with the given *label*.
- `groupCount()`: If there are no traversals, one is created. Duplicate every traversal, so that there will be two paths to the current point, resulting in grouping during execution.

B META-REPORT

B.1 Neutral reflection

B.1.1 i. A concrete minimal example is provided in section 1.2. Figure 1 and Figure 2 best illustrate our problem.

B.1.2 ii. The first paragraph and first half of the second paragraph of the Introduction provide the requirements for a good solution to our problem.

B.1.3 iii. The second half of the second paragraph of the Introduction, along with section 2.2, explain objectives of this paper.

B.1.4 iv. Section 2.2 introduces the key metrics for success in this project.

B.1.5 v. The SOTA is explored in section 1.1, with more in-depth related work detailed in section 5. Lines 385-389 explain why existing approaches do not solve our problem directly.

B.1.6 vi. Techniques are broken down into concise examples such as lazy constraint evaluation in Figure 4 and reduction through merging in Algorithm 1 and Figures 5 and 6.

B.1.7 vii. Section 4 shows that these contributions are relevant for real-world queries.

B.1.8 viii. Section 6, lines 394-401 explicitly identify the key contributions of this project.

B.2 Advocate reflection

B.2.1 Strength 1. The project has clear and easily understood metrics for evaluating the generated graphs.

B.2.2 Strength 2. The project offers precise improvements on the existing methods from [12], clearly demonstrated through examples, which are critical to performance in graph scenarios and applicable to relational dataflow programs.

B.2.3 Strength 3. The implementation choices and deployment strategies for the tool are critical to its success.

B.3 Critic reflection

B.3.1 Weakness 1. The report relies heavily on the notation and metrics from [12]. This may inhibit understanding without reading [12] and be highlighted as a lack of novelty. As the project heavily relies on existing approaches from [12], it should be considered as a prerequisite.

B.3.2 Weakness 2. Evaluation of the proposed reduce and merge algorithm complexity is minimal. The upper bound on time complexity is mentioned but not explored in-depth through experimentation. However, this project is novel in its introduction of the merge strategy. Further investigation can develop an optimal merge algorithm.

B.3.3 Weakness 3. The experimentation queries may not be broad enough to prove this tool's application. This is understandable since the tool is in its early stages. This project is laying a foundation that may be extended in the future.