Quine: A Temporal Graph System for Provenance Storage and Analysis

Ryan Wright(✉)

Galois, Inc., Portland, OR 97214, USA
rrwright@gmail.com

Abstract. This demonstration introduces "Quine", a prototype graph database and processing system designed for provenance analysis with capabilities that include: fine-grained graph versioning to support querying historical data after it has changed, standing queries to execute callbacks as data matching arbitrary queries is streamed in, and queries through time to express arbitrary causal ordering on past data. The system uses a novel combination of schema-less data storage and strongly-typed query language to enable well-typed analyses of types unexpected when the database was initialized. The system is designed to handle very large data with support for partitioning the graph to run across any number of hosts/shards across a network.

Keywords: Provenance · Graph database · Graph query language
Distributed systems

1 Introduction

Provenance data in its richest form can be represented as a connected property graph. Property graphs are ideal for representing the highly connected structure of provenance data as given by various established and experimental tracing tools. However, existing tools for working with graph data are not designed to favor provenance analysis. This demonstration introduces "Quine", a distributed graph database and processing system meant to address the following infrastructure and usability challenges faced when analyzing provenance data.

1.1 Highly Connected Temporal Data

Representing the state of a computer system at one point in time leads to highly connected representations. In a running computer system, provenance data likely includes information about the process tree hierarchy, the filesystem hierarchy, network connections or perhaps network topology, the flow of data from one source to another, or the graph of control flow among executing programs. Data elements among each of these topics can be highly related to many elements from other topics.

Graph databases are a natural choice for representing the highly connected nature of this kind of data. However, existing graph databases require modeling that data as a single graph, so that current and historical states are mixed together uniformly and distinguishable only by timestamps or other property-level data. This makes the database very large in terms of node/edge count.

Quine is implemented as a property graph which maintains the current state of each data item (nodes, edges, properties) as the derived result of its entire history. This history is stored in a manner that allows efficiently querying back in time for previous states, monitoring changes in state, or querying with other temporal constraints such as causal ordering.

1.2 Queries on Complex Structures over Time

Existing graph query languages allow expression of terms at the level of nodes, edges, and properties. Writing queries soon becomes akin to writing complex programs where the query author must maintain a mental model of the graph schema or face incomplete (or incorrect) query results. The complexity is compounded when the graph data is meant to represent events and changes over time—as is common among provenance data. When the history of data provenance is spread across a single graph, the query-writer must weave temporal constraints into each hop and each value test as a query is evaluated across the graph.

Quine is designed to represent time and change in graph data in a more manageable way. The runtime for Quine manages the history of every value as they change. The query language for Quine is meant to allow expressing queries at a level of abstraction higher than primitive node/property/edge queries. Graph patterns are expressed as classes and objects in a high-level programming language which offers type-safety at the query level, so that a query author has immediate feedback when the query they are assembling does not match the graph structure they are expecting. As a result, very complex queries can be expressed as the composition of smaller parts and can be evaluated on current and historical data, at one point in time or many.

1.3 Evolving Schema

Database schemas enforce types on singular values at data write-time, and aid query writing at read-time, but this help comes at the cost of flexibility. NoSQL and other “schema-less” databases provide speed and flexibility in the storage runtime, but little help in understanding the shape of the data when writing queries.

Quine tries to balance this tradeoff by putting the schema and corresponding constraints in the query language. Query instances first choose which schema to use, then issue type-checked queries corresponding to that schema. This approach is similar in spirit to the approach taken with GraphQL [1], however Quine does not require the client and server to share schemas—or keep them in sync.

As analysis evolves, the schema can change as well. One kind of evolution can occur by creating more meaningful/complex query terms by composition of existing terms. Another evolution is the creation/discovery of a useful schema entirely disjoint from the schema used to write the data initially. With either evolution, queries using that schema will be type-checked before a query to the backend is issued.

1.4 Scalability for Large Datasets

Provenance datasets such as those produced by DARPA's Transparent Computing program are quite large, even though they are from one single system. Interesting provenance questions spanning a large number of systems will be encumbered by the size and processing constraints of such large data. These large datasets will need to be processed by distributed systems designed for distributing a graph across many machines. Existing commercial and open source solutions struggle in this area. Neo4j scales across machines with read-only replicas that cannot support a high writing load. JanusGraph (née TitanDB) relies on the distributed capabilities of its backing store and ends up constrained by the administrative overhead and many round-trips between the graph layer and the backing store [2, 3].

Quine was designed as a genuinely distributed graph. This system is partitioned into graph shards even when run on a single system. One holistic view of the graph is transparently queryable even when those shards are served from many different hosts across a network. This allows the computational burden of reading *and writing* a large graph to be distributed across many machines, opening the possibility of supporting the large provenance datasets we hope to see in the near future.

2 Demonstration Topics

Streaming Ingest of Provenance Data. Provenance data from the DARPA Transparent Computing program will be loaded in a streaming fashion and used as the basis of the other demonstrations (see below). This data includes benign and malicious activity from multiple host machines and operating systems.

Statically-Typed Query Language. A language for ingesting data will be demonstrated. The query language is an eDSL embedded in Scala as the host language. Defining a language is done by defining classes in Scala.

Visualizing Data. A visualization of the ingested data will be demonstrated to give an intuition for how provenance data is represented in the system.

Query the Current State. The query language for Quine is realized as an embedded DSL in the strongly-typed host language, Scala. Expressions of complex patterns in the graph are aided by compile-time errors (i.e. before the query is issued to the database) to give early user feedback when large and complex queries violate the chosen schema. These kinds of queries aid provenance analysis by answering: Is the system *currently* in a state such that ___?

Query Historical Data. Quine maintains all historical data for all data in the graph, making it possible to query past states of the graph as they existed at any arbitrary point in history. Complex queries spanning large sections of the graph can be issued for many different times simultaneously. These kinds of queries aid provenance analysis by answering: Was the system *previously* in a state such that ___?

Query Through Time[1]**.** In addition to searching a single point in the past, Quine aims to provide the capability to efficiently query all points in the past to find whether a certain query predicate holds. These kinds of queries aid provenance analysis by answering: Was the system *ever* in a state such that ___?

Standing Queries. Quine allows queries to be expressed as "standing queries" which execute a provided callback for each result found. A standing query will match existing data and all cases where new additions or transformations of data result in a match. These kinds of queries aid provenance analysis by answering: Did the system *just* transition to a state such that ___?—if so, execute the desired action in real-time.

Graph Sharding (see footnote 1)**.** Quine is built from the ground up as a distributed system. One of the primary design goals for Quine is to support running a distributed database across an arbitrary number of host machines on a network.

3 Conclusion

Research Contributions. Quine represents a management system contribution to the domain of provenance research: a graph database with first-class representations of time and changes over time. In addition, this system represents significant research progress in the areas of graph query languages and distributed graph databases.

Acknowledgments. Portions of this research have been supported by DARPA under contract number FA8650-15-C-7557. Opinions or conclusions expressed in this material are those of the author and do not necessarily reflect the views of DARPA.

References

1. Facebook, Inc.: GraphQL. Working Draft, October 2016. http://facebook.github.io/graphql. Accessed 3 June 2018
2. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, pp. 195–204. ACM (2013)
3. Pacaci, A., Zhou, A., Lin, J., Özsu, M.T.: Do we need specialized graph databases?: benchmarking real-time social networking applications. In: Proceedings of the Fifth International Workshop on Graph Data-Management Experiences and Systems, p. 12. ACM (2017)

[1] Demonstration of this feature depends on some functionality which is not complete at the time of this writing. This feature will either be demonstrated or explained as "future directions" depending on progress made before the conference.