

neon

git for structured data

Introduction

The use of git to collaborate around structured data is a good thing as it draws a clear path for reproducible science.

git is not meant to store a lot of data. Extensions of git like git-annex or git-lfs don't allow to track changes in a semantic way. The lack of proper structure and typing in a git based database leads to reinvent formats and schemas to describe the stored data without delivering good enough structured semantic on changes. The difference between two versions of the data is done on raw line. They use file formats that don't understand the underlying schema that correspond to the schema of the stored data. Storing data in git it's loosely structured, because git is not aware of the schema and distribution of the stored data. This leads to a semantic on differences that are a superset of the actual differences. For instance, when you change a cell in a csv or tsv table the change appears to git as a difference between two rows. Neon store the information of that particular cell, in a particular row, in particular dataset. By changing that cell, you updated a particular row that is part of a table. In that particular table, there is a particular header, that heading semantic is attached to its type, constraints, distribution description and some metadata like labels in multiple languages. You can attach indexing strategies to columns in your table like geospatial indexing or full-text search. This description allows neon

to *checkout* a database without versioning information, that is an optimized instance of the last version of the stored data.

Projects that could benefit from a versioned data store of structured data are: [db.nomics](#), [datahub](#) and wikidata.

The use of a Ressource Description Framework (RDF) and Linked-Data (LD) as described in [Collaborative Open Data Versioning: A Pragmatic Approach Using Linked Data](#) will allow organizations to work on standard ontologies like [INSPIRE RDF Vocabularies](#) to help in the process of sharing and re-using structured data.

Having a versioned data store that works like git will allow to cooperate around data the same way people collaborate around source code

The goal of neon is to replace the use of git, streamline and make practical cooperation around the creation, publication, storage and re-use of structured data that are possibly bigger than memory.

Implementation

It's inspired from [R&Wbase: Git for triples](#) and [R43ples: Revisions for Triples](#)

neon is implemented on top of [wiredtiger storage engine](#).

A prototypal implementation is available at <https://github.com/amirouche/neon>

Neon is a triple store that implements the notion of graph ie. it is a quad store. That is it expose the following kind of tuples:

```
(graph subject predicate object)
```

The store

That being said, the backend stores 7-tuples in a table called store that as the following schema:



```
(uid) → (graph, subject, predicate, object, life, transaction)
```

Where the first tuple contains wiredtiger table key columns and the second tuple contains the table value columns. The columns can be described as follow:

- uid is a random unique 64 bits unsigned integer identifying the tuple
- graph, subject, predicate, object are 64 bits unsigned integers which maps to strings in their respective graphs, subjects, predicates, objects, tables
- transaction is a 64 bits unsigned integer identifying the transaction defined in another table called transactions described later.
- life is a flag denoting whether the associated (graph, subject, predicate) tuple exists in a version of the knowledge base. Right now, life is a string column (text column in SQL parlance) but it might changed in the future to something else because right now life column can have only two values : an empty string to describe alive state and something else for dead state. It could be boolean column. The life flag is used to describe that a particular quad (graph, subject, predicate, object) doesn't exist in particular version.

Based on the 7-tuples stored in store table, it is created another table called `index:store:index` created automatically by wiredtiger to store the following tuples:

```
(graph, subject, predicate, object, life, transaction, uid) → (
```

In this case the table has not value columns, this is done like so to speed up querying.

wiredtiger allows to quickly iterate over rows of the table that match a particular prefix. That is, it will be quickly possible to know the different versions of a given tuple that looks like:

```
(graph, subject, predicate, object)
```

That will allow to complete the tuple with (life, transaction, uid) tuple. Then the “grouping” algorithm makes it's entrance and try to know what is the latest version of the quad.

The query interface is for the time being made of a single polymorphic procedure named ref that allows to retrieve a lazy stream of 4-tuples given a branch or #false when there is no results. The tuples looks like:

```
(graph, subject, predicate, object)
```

The ref procedure takes a variable number of arguments the full version of the signature looks like this in pseudo Java code:

```
branch.ref(String graph, String subject, String predicate, Seri
```

Or in lisp style:

```
(ref branch subject predicate object)
```

Where branch denotes the working branch referencing the latest version of the quads in a branch of the database.

That particular form of ref, will return a boolean denoting the existence of the quad passed as argument in the associated database version.

There is also a form of the ref procedure that takes only four arguments:

```
(ref branch graph subject predicate)
```

That is somewhat implemented as the following SQL query over a table of 6 integer columns:

```
SELECT graph, subject, predicate, object, MAX(version), uid  
FROM store  
GROUP BY graph, subject, predicate, object
```

It will return `#false` if there is no result.

To compute `MAX(version)` it use a topological sort over the Direct-Acyclic-Graph (DAG) history this gives a hashmap association (called simply history) of a version integer with an integer value denoting the *historical significance* of the tuple. `MAX` takes into account the `life` flag so that the whole query returns only the (graph, subject, predicate, object) tuple that exist at a particular point in history.

The history mapping transactions to history significance must be cached.

Merge commit must introduce the necessary changes to avoid ambiguities aka. reconcile the history between two branches. Because merge tuples are made after conflicting tuples, the topological sort score them better ie. higher in history significance.

It's possible to have multiple times the same predicate for a given (graph, subject). That is it's possible to store things like the following in pseudo turtle syntax:

```
wiki:Entity-Attribute-Value-Model wiki:seeAlso wiki:Triple_Store
wiki:Entity-Attribute-Value-Model wiki:seeAlso wiki:BlazeGraph
wiki:Entity-Attribute-Value-Model wiki:seeAlso wiki:Neon_Database
```

Otherwise said, a vertex can have several edges labeled with the same predicate. In a property-graph, it could possibly be interpreted as the ability to have an arbitrary number of (key, value) properties with the same key

transactions and history significance

The equivalent of what git calls commits are called transactions in neon because the default behavior is to map one-to-one with database transactions

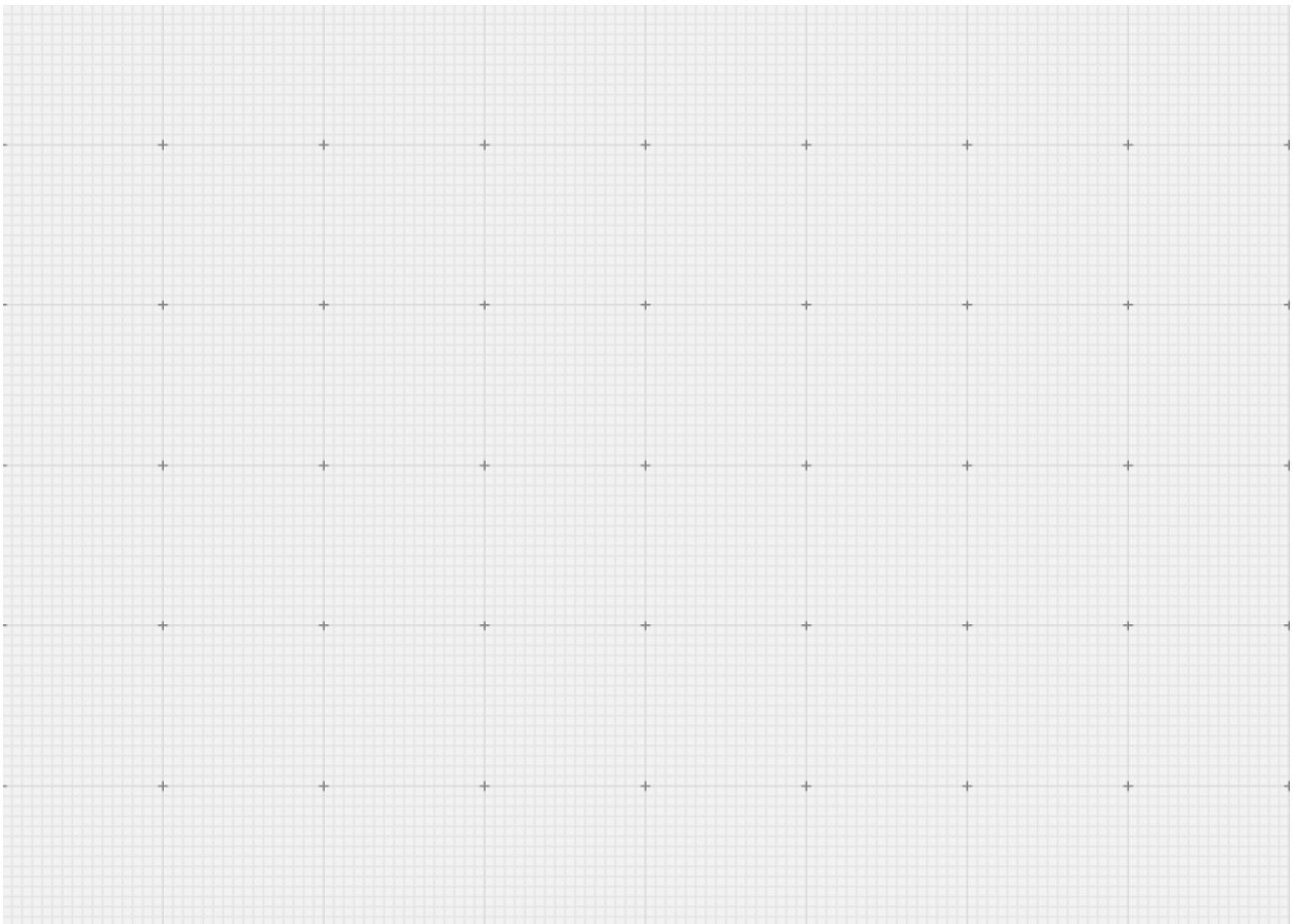
There is a table called history that stores the information about the transactions and how they related to each other. history table has the following schema:

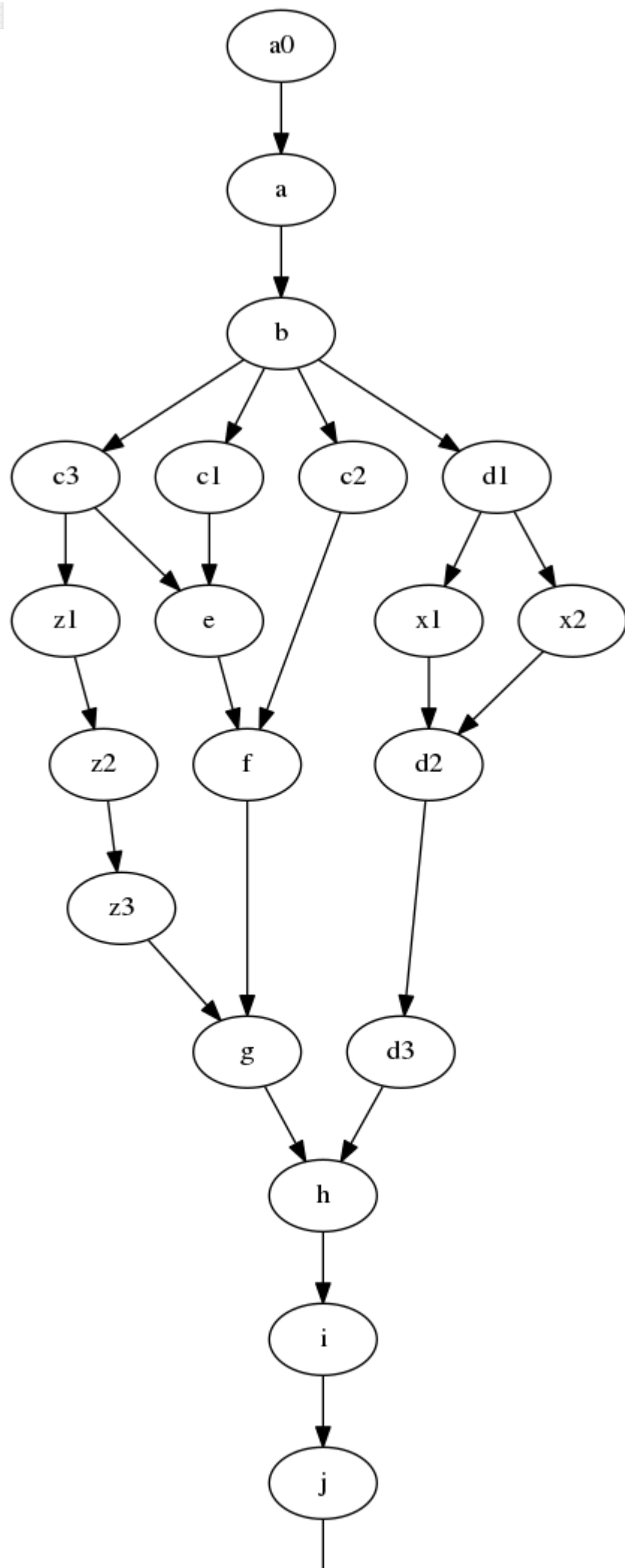
```
(txuid) → (parent1, parent2, sha-256)
```

Where:

- txuid is a random unsigned 64-bits integer that identifies a transaction
- parent1 and parent2 are unsigned 64bits integers referencing the parent transactions. In the case of merge commit parent2 is not zero.
- sha-256 is a string representing the hash of the the transaction. It's built out of the serialization of the added and deleted quads ie. the difference and the parent transactions identifiers

history schema allows to build a DAG graph where vertices have at most two incomings edges and a certain number of outgoings edges. The DAG can look like the following:





The history significance algorithm will compute an total order (?) using a topological sort that switch of branch before consuming a vertex that has several outgoing edges. In the above example, the topological sort gives the following result:

```
a0 a b d1 x2 x1 d2 d3 c3 c1 e c2 f z1 z2 z3 g h i j k
```

The index of a vertex in that list is its history significance. This allows to tell which transaction to take into account when computing the value of given quad.

branches & graphs

Right now branches and graph are stored respectively in references and graphs tables. Mind the fact that once a graph is created it can not be deleted

Roadmap

Command Line Interface

The target command line interface is the following:

```
neon init
neon checkout VERSION
neon tag list
neon tag add TAG
neon remote list
neon remote add REMOTE
neon remote remove REMOTE
neon graph list
neon graph add GRAPH
neon push
neon pull REMOTE GRAPH
neon import
neon export
```


The other interactions will happen only through code.

nql query engine

A nql query language inspired from sparql but adapted to scheme syntax

The query engine will work like that:

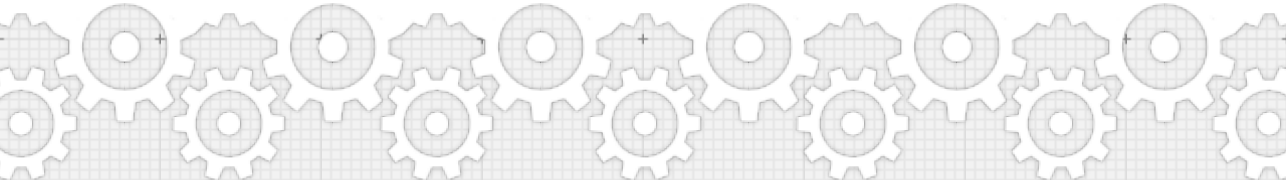
- Assign cardinality to each quad pattern
- Start with the most selective pattern
- For each remaining join propagate variable bindings and recurse

Nice to have

There is various nice-to-have features like compliance with RDF and LD standards but the most valuable feature that is not part of the proof-of-concept is the full replacement of git by neon. That is use neon to store a file-system hierarchy of source code with the extended goal to handle s-expr based language through a tree-diffing algorithm.

Conclusion

The goal of neon is to replace the use of git, streamline and make practical cooperation around the creation, publication, storage and re-use of structured data that are possibly bigger than memory.



[Attribution-NonCommercial-ShareAlike 4.0 International](#)