Geist: a multimodal data transformation, query, and reporting language

Meng Li, Timothy McPhillips, Bertram Ludäscher
School of Information Sciences, University of Illinois at Urbana-Champaign

cirss.github.io/geist-p

Introduction

- A new templating language for declarative data manipulation, query, and report generation.
- Aims to enable developers to use whatever language or tools they like regardless of where the data is stored.

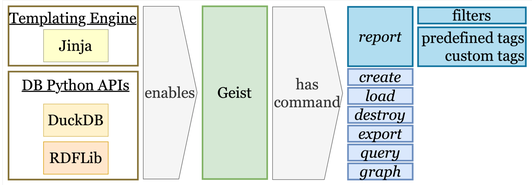

Figure 1. An overview of Geist. Building on the Jinja and database Python APIs, Geist provides "report", "create", "load", "destroy", "export", "query", and "graph" commands. Among them, the "report" command relies on filters and tags to expand a Geist template using datasets.

Why Geist?

Template-based. Specify your needs in the declarative Geist language rather than procedural Python functions.

Active tags. Combine operation tags (e.g., *create*, *destroy*, *load*, and *query*) with layout tags (e.g., *html*, *graph*, *img*, and *table)* to manage, manipulate, query, and present data.

Reuse templates as new tags. Easily define your own tag in a Geist template format through files within the *use* tag.

Modularize complex queries. Split long (SQL, SPARQL, Cypher, etc.) queries into small chunks for readability, maintainability, and reusability.

Manage databases. Besides generating readable reports, Geist can perform inserts and updates on new or existing datasets during template expansion, which makes it a general-purpose data management language.

Augment query language expressiveness. Add iteration, recursion, and conditional flow to queries while maintaining comprehensibility.

Backend-agnostic queries. Choose the languages and tools regardless of where the data is stored, e.g., SPARQL to query tables stored in DuckDB.

Multimodal data manipulation. Compose and nest queries over multiple query languages, e.g., embedding query results from a relational database into a graph query to retrieve information from a graph database.

Simple installation[1,2].

```
pip install geist-p
```

Examples

EX1: Modular SPARQL

This SPARQL query, adopted from the Structured Data Transformation History (SDTH) project[3], is used to extract program steps that use the "PPHHSIZE" variable directly and indirectly.

Method 1: a long SPARQL query without using Geist

```
SELECT distinct ?psource ?oname
WHERE {
    {
        ?pstep sdth:usesVariable+ ?o .
        ?pstep sdth:hasSourceCode ?psource .
        ?o sdth:hasName ?oname .
        FILTER (?oname = "PPHHSIZE")
    }
    UNION
    {
        ?pstep sdth:usesVariable+ ?s2 .
        ?pstep sdth:hasSourceCode ?psource .

        {
            SELECT distinct ?s2 ?oname
            WHERE
            {   ?s2  sdth:wasDerivedFrom+ ?o2 .
                ?s2  sdth:hasName ?oname .
                ?o2  sdth:hasName ?o2name .
                FILTER (?o2name = "PPHHSIZE" )
            }
        }
    }
} ORDER BY ?psource ?oname
```

Method 2: modular SPARQL query using Geist

1. Define reusable templates (chunks) in the "templates.geist" file

```
{% template chunk_usesvariable %}
    {
        ?pstep sdth:usesVariable+ ?o .
        ?pstep sdth:hasSourceCode ?psource .
        ?o sdth:hasName ?oname .
        FILTER (?oname = "PPHHSIZE")
    }
{% endtemplate %}

{% template chunk_wasderivedfrom %}
    {
        ?s2 sdth:wasDerivedFrom+ ?o2 .
        ?s2 sdth:hasName ?oname .
        ?o2 sdth:hasName ?o2name .
        FILTER (?o2name = "PPHHSIZE" )
    }
{% endtemplate %}
```

2. Tell Geist where these chunks are by adding the *use* tag to the Geist template

```
{% use "templates.geist" %}
```

3. Write a shorter SPARQL query

```
SELECT distinct ?psource ?oname
WHERE {
    {% chunk_usesvariable %}
    UNION
    {
        ?pstep sdth:usesVariable+ ?s2 .
        ?pstep sdth:hasSourceCode ?psource .

        {
            SELECT distinct ?s2 ?oname
            WHERE {% chunk_wasderivedfrom %}
        }
    }
} ORDER BY ?psource ?oname
```

EX2: Query over DuckDB tables and RDF triples

A citation network, which describes the citations within a collection of papers, is stored as an RDF dataset. For example, the triple ":P1 :cites :P2 ." denotes paper P1 cites paper P2. Also, there is a table named *paper_author* stored in DuckDB. Assume both datasets (databases) named *demo*.

paper	author
P1	A1
P1	A2
P1	A3
P2	A1
P2	A4

Table 1. First 5 rows of the "paper_author" table in the "demo" database. For example, the first row means paper P1 is written by author A1.

For each paper written by author A1, how can we find all papers that cite it, both directly and indirectly?

An SQL query of the *paper_author* table stored in DuckDB finds all papers written by author A1.

```
SELECT paper
FROM paper_author
WHERE author = 'A1';
```

A SPARQL query of the RDF dataset finds all papers that directly or indirectly cite a particular paper (here, paper P1).

```
SELECT ?citing
WHERE {
  ?citing :cites+ :P1 .
}
```

With Geist, these queries can be combined in a single declarative Geist template.

```
geist report << __END_TEMPLATE__

{% query "demo", datastore="duckdb",
                 isfilepath=False as papers %}
  SELECT paper
  FROM paper_author
  WHERE author='A1';
{% endquery %}

Author A1 has published the following papers:

{% for _, row in papers.iterrows() %}

 {% query "demo", datastore="rdflib",
                  isfilepath=False as citations %}
     PREFIX : <http://demo.com/>
     SELECT ?citing
     WHERE {
         ?citing :cites+ :{{ row["paper"] }} .
    }
 {% endquery %}

 {{row["paper"]}} was cited directly / indirectly by:
 {% set citing_papers = citations["citing"].values %}
 {% for citing_paper in citing_papers %}
     {{ citing_paper }}
 {% endfor %}

{% endfor %}

__END_TEMPLATE__
```

Application

The TRAnsparency CErtified (TRACE) project[4] aims at addressing the problem of computational reproducibility by certifying the original execution of a computational workflow rather than requiring confirmation of reproducibility via re-execution. The details and certification of a particular computation are represented in a Transparent Research Object (TRO) described by an RDF vocabulary (TROV) that can be queried with SPARQL. We use Geist to perform queries on TROs to yield human readable reports describing the digital artifacts (data, code, notebooks, container images) employed by the described workflow.

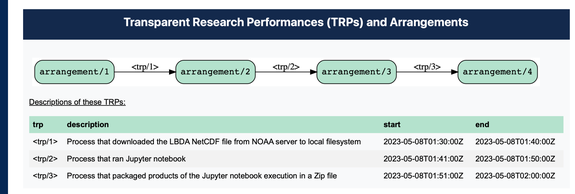

Figure 3. Part of the TRO report generated by Geist.

How Geist works

- **Uses database Python APIs.** Employs Python packages such as RDFLib and DuckDB to access and query datasets.
- **Builds on the Jinja template engine.** The Jinja template supports predefined tags. It also supports recursive nesting of tags with any number of layers.
- **Adds custom templating capabilities**. For example, before rendering a Jinja template, Geist parses the text, finds all files declared via *use* tags, generate classes for each block of *template* code within these files, and adds these classes to the Jinja environment as extensions to enable custom tags.

References and resources

[1] Geist PyPI: https://pypi.org/project/geist-p
[2] Geist GitHub: https://github.com/CIRSS/geist-p
[3] SDTH: cosmos-conference.org/2024/slides/13_Alter_SDTH_COSMOS.pdf
[4] TRACE: transparency-certified.github.io

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2209628.

ILLINOIS