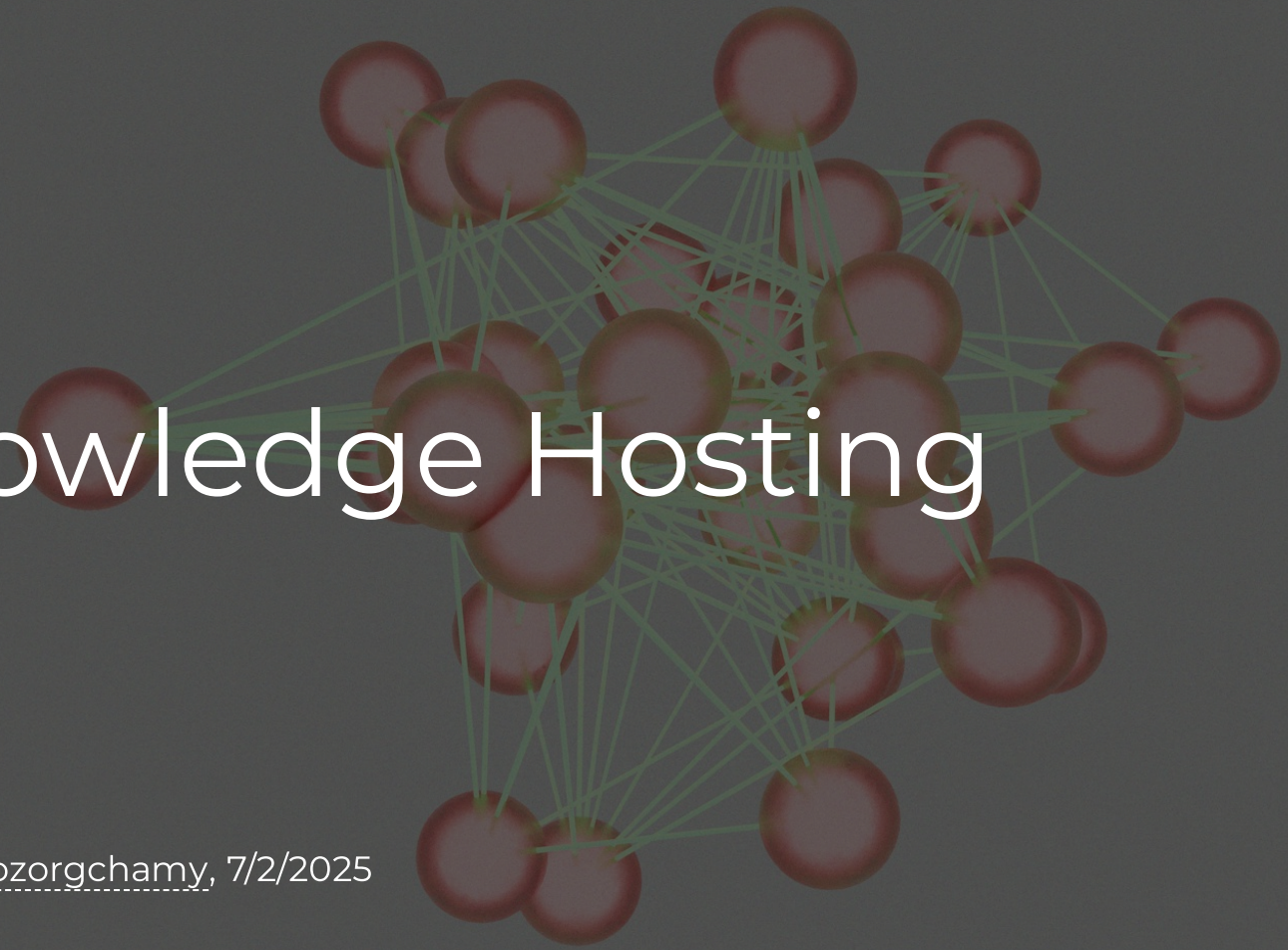


Knowledge Hosting

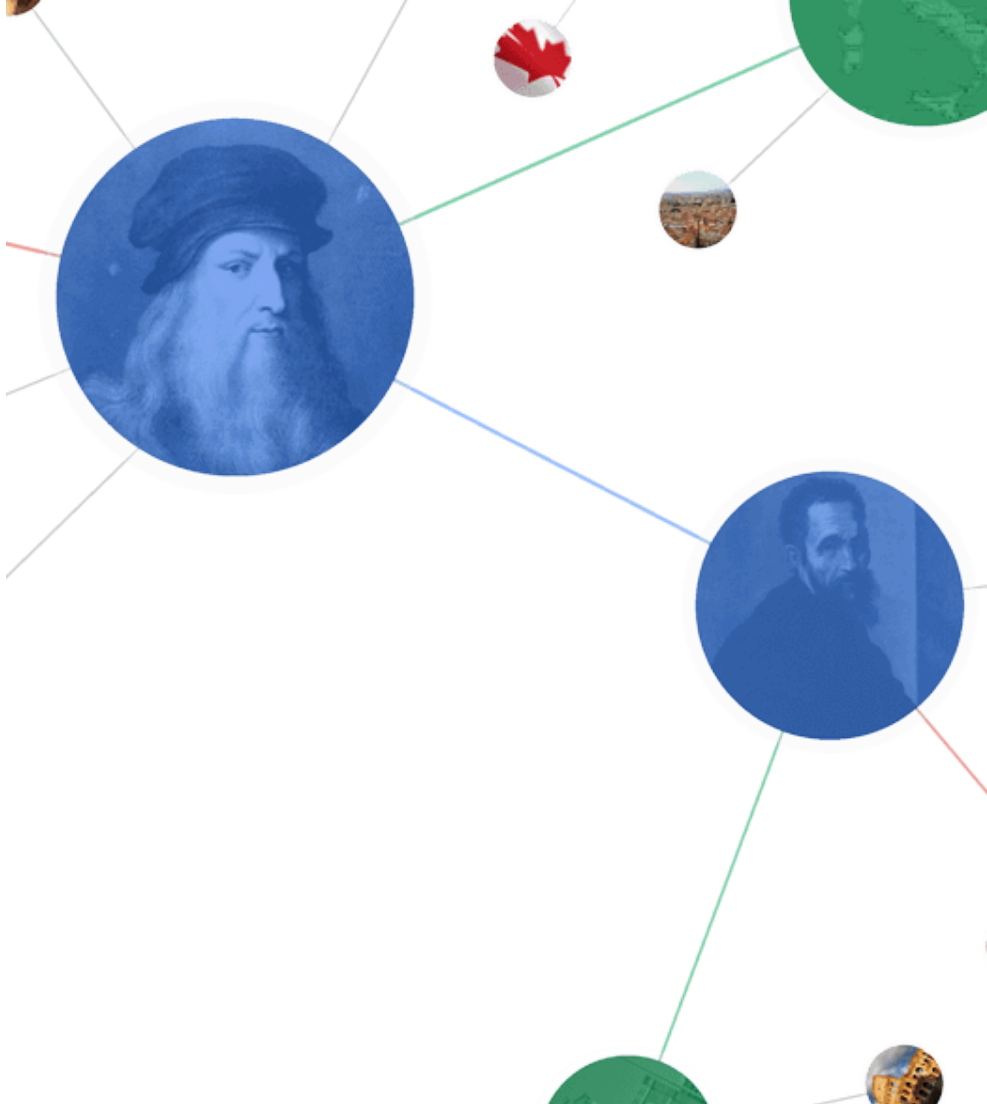
behrooz bozorgchamy, 7/2/2025

Why we need to host a KG(Knowledge graph)?

- It has many application.
- We saw what is it benefits.

What build KGs?

a set of nodes and edges between them that represent entities and their relationships



What challenges we face to host a KG?

- Size
  - KGs have vast size.
  - might have billions of facts.
- Data model
  - A KG is technically a semantic network.
  - A directed or undirected graph consisting of vertices which represent concepts, and edges, which represent semantic relations between concepts.
- Heterogeneity
  - KGs by nature have a flexible schema and the host have to support it.
- Velocity
  - KGs change rapidly
    - new knowledge is inferred through reasoning based on the given facts.
    - new sources may get integrated.

What challenges we face to host a KG?

- Points of view
  - Different users or applications may need to interact with the same KG in distinct ways.
  - The hosting solution should allow the data to be viewed or processed differently depending on the use case.
    - Applying specific rules, filters, or inference logic to suit each scenario.
- Deployment
  - The hosting solution provides various Application Programming Interfaces (APIs) and query interfaces and supports different formats to enable a variety of applications.

Knowledge Hosting Paradigms

- KG have a **graph data model** *logically*.
- It can host in other database paradigms.
- Every paradigm has it own benefits and disadvantages.
- Three Most most popular hosting paradigms for KGs
  - Relational databases
  - Document stores
  - Graph databases

Relational Databases

- **Relational Model (Codd 1970[1])**: Decouples logical data representation from physical implementation, Isolates data from hardware and application logic (program).
- **Structured Query Language (SQL)**: High-level declarative language reflecting relational algebra for querying relational databases.
- **Relational Model Storage**: Data stored as tuples in structures called relations (tables) and you can querying it with SQL.
- **Relation Structure**:
  - **Header**: Finite set of attribute names (columns).
  - **Body**: Set of tuples (rows).
- **Abstract Access Layer**: Provides applications with a way to access, store, and modify data.

[1] Codd

Example Relation

Customer (*Customer_ID*, Tax_ID, Name, Address, City, State, Zip, Phone, Email, Sex).

Customer ID	Tax ID	Name	Address	[More fields. . .]
1234567890	555-5512222	Ramesh	323 Southern Avenue	. . .
2223344556	555-5523232	Adam	1200 Main Street	. . .
3334445563	555-5533323	Shweta	871 Rani JhansiRoad	. . .
4232342432	555-5325523	Sarfaraz	123 Maulana Azad Sarani	. . .

An example

Relational Databases

Operations

There are various operations that can be done on a relation.

- Join
- Projection (𝛱)
- Selection (σ)
- set union, set difference, and exist ...

Storing KGs

- Large KGs may lead to large tables.
- Querying Challenge
  - Requires any type joins, potentially processing vast data.
    - A graph with 1M triples could involve 10^{12} rows in a naïve implementation.

Domain and Range Definitions (RDF Schema)

- Domain: Specifies classes to which properties apply.
- Range: Defines the type of value a property can take.
- **Challenge**: Integrity constraints can partially enforce domain/range in a closed-world setting, but full RDFS semantics must be handled by applications.

Class and Property Hierarchies

- Representation requires multiple auxiliary tables and joins.
- Application must hardwire semantics for hierarchies.
- Inheritance Issue: Property inheritance between subclasses (e.g., range inheritance) is complex and must be managed by the database designer.

Impact on Declarative Nature

- **Application logic must hardwire semantics of modeling languages (e.g., RDFS), reducing the declarative nature and reusability of knowledge graphs.**

Statement table

- A simple way.

How?:

- RDF triples representing the KG can directly be stored in this table without any change

Problems

- The data are not normalized.
- A growing number of triples result in inefficient self-joins

Subject	Predicate	Object
Hotel	subClassOf	Thing
Hotel	subClassOf	LocalBusiness
166417787	type	Hotel
166417787	description	“Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig.”
166417787	address	_:123
_:123	addressCountry	_:456
_:456	type	Country
_:456	name	“DE”
. . .	. . .	. . .

An example

Class-centric table

- Creates one table per class type to store all property values for that class.
- Properties may appear in multiple tables if shared across classes.

Table Structure:

- Each class in the KG is represented as a table.
- Properties of the class are represented as columns.
- Properties without value assertions in the KG may result in columns with NULL values, leading to sparse tables (many NULLs).
- For Queries use join with other classes(table).

a. *Class* table

ID	subClassOf
LocalBusiness	Thing
Hotel	LocalBusiness

b. *Hotel* table

ID	Type	Description	Address	Name	Geo
166417787	Hotel	“Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig.”	_:123	“Hotel-Restaurant La Fontana”	_:789

c. *Country* table

ID	Name
_:456	“DE”

d. *PostalAddress* table

ID	addressCountry
_:123	_:456

An example

Drawbacks

1. Adding new properties and classes is cumbersome as the schema must be recompiled.
2. The level of normalization is not enough to handle multivalued properties as they lead to repeat of tuples for each value of a property.
    - A hotel has multiple descriptions, we would need to create another tuple for the same hotel with a different description but repetitive values for all other columns.

Property-centric

- This approach use one table per property
- Each table contains two columns
  - Subject
  - Object
- This approach easily allows multivalued properties.
- But, duplication of subjects is still necessary.

Drawbacks

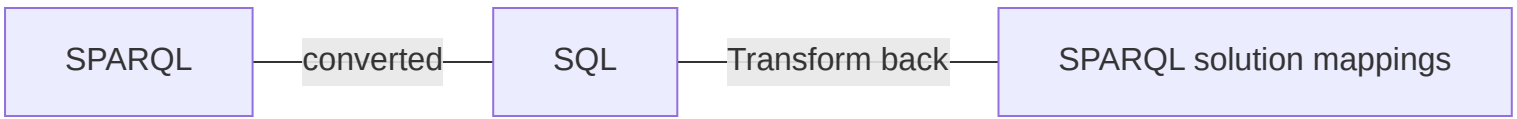
- Adding a new property requires the creation of a new table and, therefore, a recompilation of the schema.
- A common operation like retrieving properties defined on a single instance requires joins over a vast number of tables.

Type		**Geo**	
Subject	**Object**	**Subject**	**Object**
166417787	Hotel	166417787	_:789
Name		**Address**	
Subject	**Object**	**Subject**	**Object**
166417787	“Hotel-Restaurant La Fontana”	166417787	_:123
Description			
Subject	**Object**		
166417787	“Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig”		

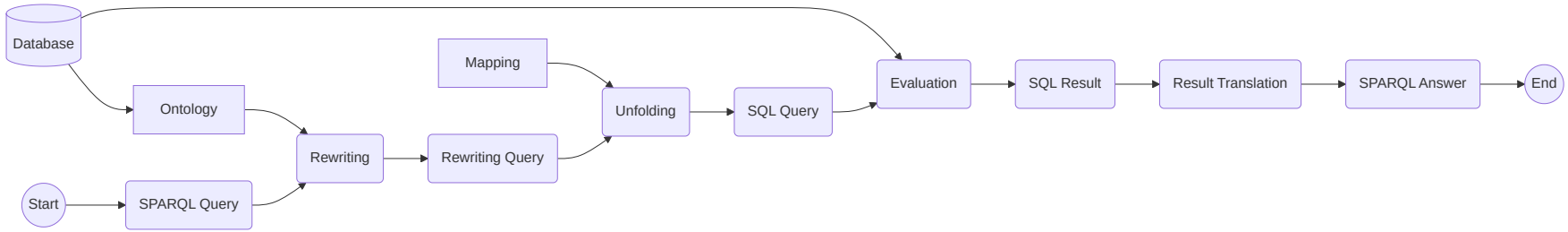
An example

Virtual RDF graphs

- This approach have been a popular way to convert relational databases to KG in recent years.
- Do not store a concrete knowledge graph in a relational database.
- Provide an ontology-based access layer over existing relational databases.



How works?



- Compiling the ontology into the mapping in an offline phase.
- Exploiting the constraints over the data to strongly simplify the queries after the unfolding phase.
- Planning query execution using a cost-based model.
- A direct implementation of this workflow may be highly inefficient, some optimizations are required.

Advantages and Drawbacks

Advantages

- They do not require any preprocessing on a RDBMS .
- They allow ontology-based access via mappings and query rewriting.
  - SPARQL to SQL
- They provide a relatively cheap way to build a KG from relational databases.
- They provide a smooth integration in industrial standard software environments.

Drawbacks

- Many things can go wrong with query rewriting and unfolding.
  - mappings need extra attention
- Querying the schema is challenging.
  - Due to the underlying relational model.
- Typically, only limited querying and reasoning capabilities are provided.

An Implementation

- **Ontop**

- Virtual RDF graph framework (Xiao et al. 2020).
- Distributed under the Apache 2.0 license.
- Supports customized mapping language and R2RML (RDB to RDF Mapping Language).
- Handles a subset of SPARQL 1.1 with optimizations for Join, Union, and LeftJoin operations.
- Implements reasoning through query rewriting, supporting the OWL 2 QL profile.

Summarize

- **Data Model**: Relational model struggles with flexible schemas of knowledge graphs, where data and schema boundaries blur, unlike rigid, decoupled schemas in relational databases.
- **Heterogeneity**: Representing graph models requires numerous queries and costly joins, making it inefficient.
- **Velocity**: Inference generating new TBox knowledge may required schema recompilation, an expensive operation.
- **Relationship Representation**: Modeling relationships like inheritance is complex.
- **Entailment Rules**: Implementing RDFS entailment rules is not straightforward and often requires application logic or external rules.

Document Model

- Stores data as nested key-value pairs, organized into documents, similar to tuples in relational databases.
- Gained popularity with big and streaming data due to flexible schema and easy scalability.
- Collections
  - Documents are grouped into collections, analogous to tables in relational databases.
  - Unlike tables, collections do not enforce a rigid schema.
  - Each document in a collection can have a unique structure and metadata.
- Ways to host KG
  - Nested objects
  - The adoption of document references

Nested objects

- The native way of storing documents
- Everything about an instance is stored in one document.
- This makes it easier to access individual documents.
- **But**, the updates are problematic as a nested object appearing in many documents needs to be updated

Example

```
{
  "id": "166417787",
  "@type": "Hotel",
  "description": "Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig.",
  "geo": {
    "latitude": "49.26178",
    "longitude": "7.1033"
  },
  "name": "Hotel-Restaurant La Fontana",
  "address": {
    "addressCountry": {
      "@type": "Country",
      "name": "DE"
    }
  }
}
```

- Each nested object without an id field represents a blank node.

Document references

- Splits documents into conceptually cohesive collections.
- Stores references to objects instead of nesting them, avoiding update issues associated with nested objects.
- Requires application-level implementation of Join operations, reducing distinction from relational databases.

```
{
    "id": "166417787",
    "@type": "Hotel",
    "description": "Das gemütliche familiär geführte Haus liegt ruhig, in Waldnähe, dennoch verkehrsgünstig.",
    "geo": "...",
    "name": "Hotel-Restaurant La Fontana",
    "address": "-123"
},
{
    "id": "-123",
    "addressCountry": {
      "@type": "Country",
      "name": "DE"
    }
}
```

Some Implementations

- Mongodb
- RocksDB
  - Open Source
  - Developed and maintained by Facebook.
- AllegroGraph

Semantic Annotation vs. Knowledge Graph

- Semantic Annotation: Focuses on publishing machine-understandable annotations for web pages.
  - Efficiently achieved via one-to-one mapping between a JSON-LD document and its corresponding web page.
  - Document stores excel due to straightforward document-to-annotation correspondence.
- Knowledge Graph: Less suitable for document stores.
  - Challenges in leveraging graph data connectedness and supporting reasoning.
  - Querying and reasoning are more efficient when facts are grouped, rather than scattered across multiple documents.

In the end

- Schemaless design with looser consistency checking, supporting heterogeneity and velocity.
- Allows instances to have varied metadata and multiple property values, avoiding null values or duplications seen in relational databases.
- Native JSON support enables **O(1)** access for retrieving instances in JSON-LD format, where each instance is a document.

Graph Databases

- This paradigm represents data and/or the schema as graphs.
- Many implementations support flexible schemas, with some offering data integrity features (e.g., constraints, identity, referential integrity)
- **Graph Data Model**:
  - **Nodes**: Represent entities, with metadata as key-value pairs and labels indicating roles (e.g., instance type).
  - **Edges**: Represent relationships, including direction, start/end nodes, a label for relationship type, and optional property value assertions.
- Many graph database implementations support hosting with flexible schemas to support heterogeneity and velocity.

Types of Graph Databases for KGs

Property graphs

- Graph models describing predicates connecting two entities.
- Nodes: Represent entities, holding key-value pair property assertions and labels indicating roles (e.g., instance type).
- Edges: Represent directed relationships between nodes, with a start/end node, a label for relationship type, and optional property assertions.

Property Graph Characteristics

- No current standard exists; development is primarily industry-driven.
- Neo4J: A widely adopted property graph implementation.
- Query Language: Graphs in Neo4J are queried using the Cypher language.

RDF Triplestores

- Optimized for RDF triple structure: `<subject><predicate><object>`.
- Often called triplestores, following the RDF data model.
- Support reasoning with RDFS and OWL2 via built-in reasoners.
- Use SPARQL 1.1 as the query language; many provide APIs for data retrieval and manipulation.
- Backend implementations vary (e.g., relational databases, document stores), but recent ones natively support RDF graph structure.
- Offer flexible schema, often blurring the line between schema and data for querying.

Standards and Scalability

- Built on W3C standards (RDF, RDFS, OWL, SPARQL), simplifying vendor migration and tooling development.
- Can scale to trillions of triples for knowledge graphs.

VS Property Graphs

- **Triplestores**: Native reasoning support via ontology languages; cumbersome to make statements about statements due to RDF model limitations.
- **Property Graphs**: Natively support statements about relationships but lack built-in reasoning.
- **Extensions**: Named Graphs and RDF-Star allow adding metadata (e.g., temporal/spatial validity, provenance) to triples.

Bridging Graph Models

- Efforts to combine RDF and property graph capabilities:
  - Neo4J Neosemantics Plugin: Supports RDF(S), OWL, SKOS, and SPARQL for property graphs.
  - AnzoGraph: Supports both RDF and property graph data models.
- Ongoing standardization for property graphs and general graph databases.

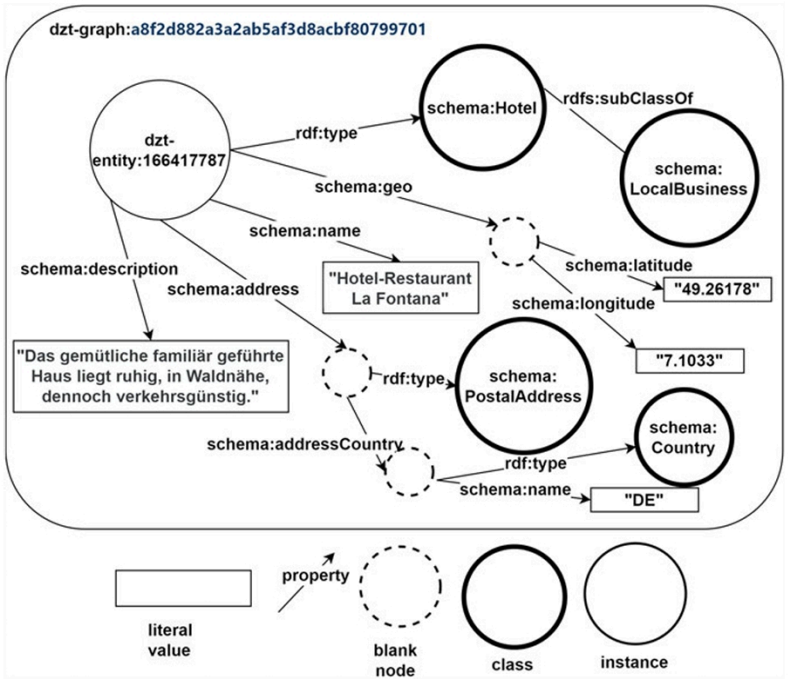
Example

German Tourism Knowledge Graph (GTKG)

Tourism is one of the most critical sectors of the global economy. Due to its heterogeneous and fragmented nature, it provides one of the most suitable use cases for knowledge graphs.

- Integrates tourism data from 16 German states, including events, tours, and points of interest (POIs).
- It is a publicly accessible resource and uses a schema.org-based schema to describe the integrated data.
- It builds around standard semantic technologies like RDF(S), SPARQL, SHACL and widespread technologies like RML.
- Mostly Creative Common license like CC-BY-SA

GTKG summary



An excerpt from the GTKG[1]

[1]German Tourism Knowledge Graph

Storing

- Hosted in a GraphDB Enterprise instance with over 13 million statements.
- Data Loading in GraphDB:
  - RDF graphs can be uploaded via:
    - Graphical User Interface (GUI).
    - API.
    - SPARQL INSERT/INSERT DATA queries.

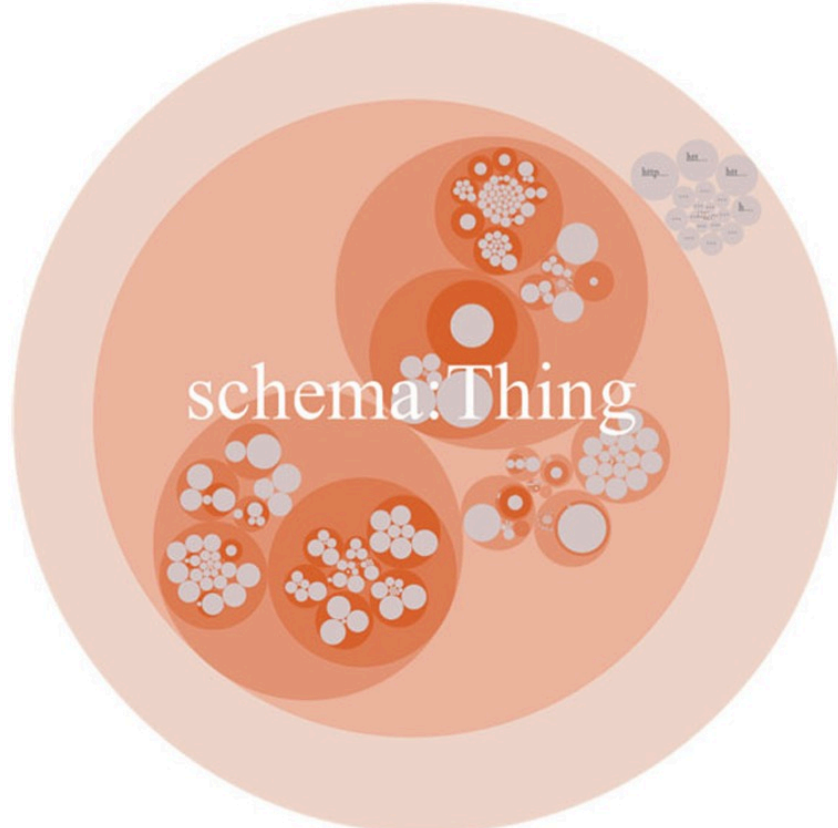
Querying

- Provides a graphical interface and API for executing SPARQL 1.1 queries on RDF graphs.
- Queries run on the default graph (union of all named graphs in the repository) unless a named graph is specified.
- Use FROM `NAMED` or `GRAPH` keywords to specify a named graph, scoping triple patterns to the query's context.

Visualization

Class hierarchy

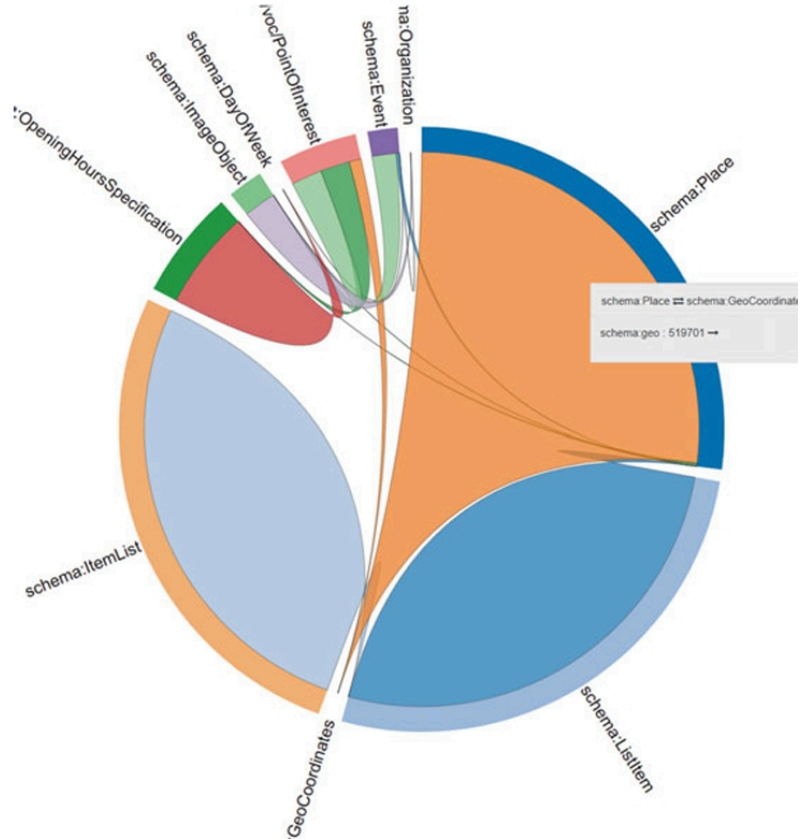
- Displays a Venn-diagram-based view of classes in the RDF dataset.
- Each circle represents a class; smaller circles inside indicate subclasses.
- Circle size reflects the number of class instances.
- Useful for "visual debugging" to identify errors (e.g., stray classes from typos in class IRIs).



shows schema:Thing as the largest circle, a top-level class in schema.org.

Class relationships

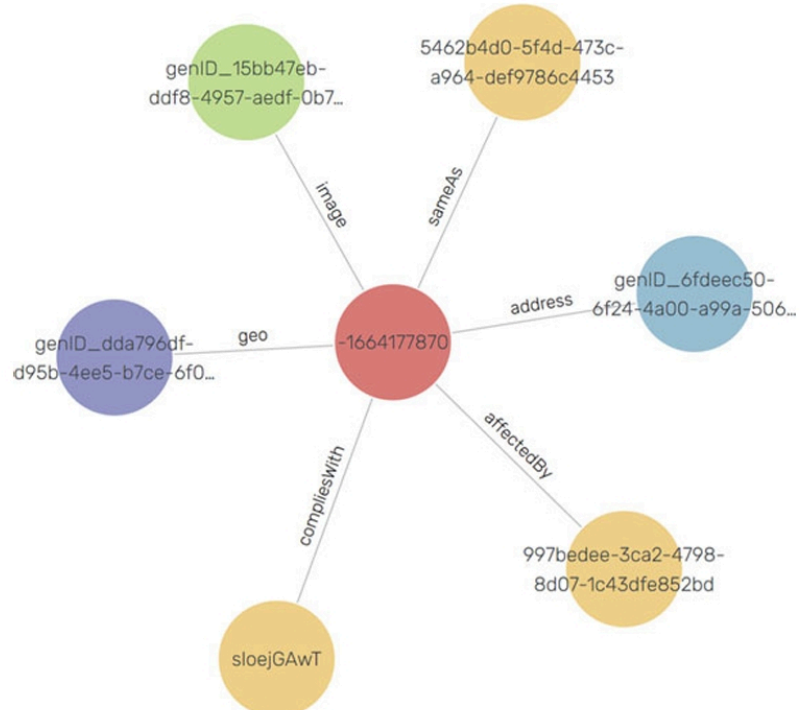
- Shows the number of links between instances of class pairs.
- Highlights highly connected class pairs, revealing key types in the knowledge graph.



indicates ~520K links from schema:Place to schema:GeoCoordinates via the schema:geo property.

Visual graph

- Provides a graph-based view for a specific resource IRI or a subgraph from a custom SPARQL query.
- Displays incoming and outgoing edges for a selected node.
- Configurable and interactive, allowing users to expand additional nodes.



visualizes showing 1-hop connections by default.

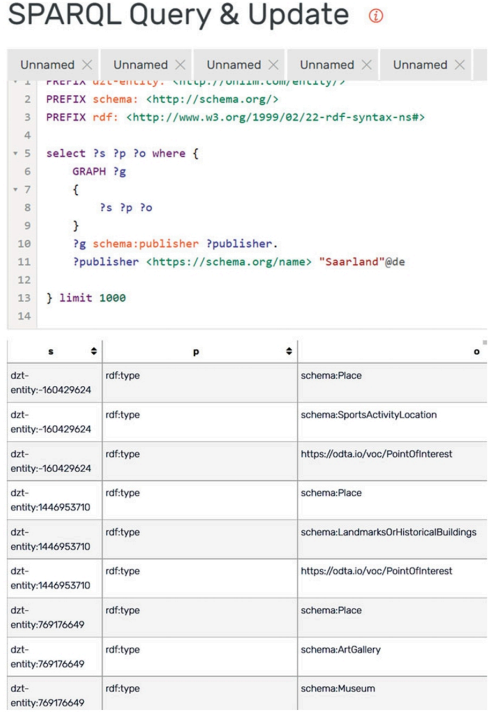
Reasoning

- GraphDB Reasoner:
  - Based on W3C recommendations, infers new facts using forward chaining entailment rules.
  - Employs a materialization strategy, applying inference rules repeatedly to explicit statements until no new implicit statements are produced.
  - Inferred statements are generated at load time, leading to longer loading times but faster query execution.
- Reasoning Profiles:
  - Supports RDF(S), RDFS-Plus, and OWL2 profiles.
  - **OWL-Horst**: Non-W3C profile with restrictions on OWL RL for improved reasoning performance on large datasets.
  - **RDFS-Plus**: Non-W3C profile used in the GTKG, supports subsumption reasoning for types/properties and reasoning for inverse/transitive properties.

Reasoning

- Virtual Named Graphs:
  - Two built-in named graphs:
    - Explicit Named Graph: Contains only explicitly loaded triples.
    - Implicit Named Graph: Contains only inferred triples.
  - Queries can specify whether to include only explicit statements or both explicit and implicit statements.

Query Example



Query

In the end

Relational database	Document database	Graph database (triplestore)
Fixed schema.	No fixed schema.	No constraining, rigid schema(still they can have ontologies).
Schema adaptations require a change of the database schema.	Documents in a collection can have different properties; adding properties possible without affecting already existing documents.	Allows to easily integrate data using different schemas (ontologies); adding properties does not affect existing graphs as in RDB.
Data are organized in different tables.	Documents are organized in collections.	Triples can be organized in named graphs.
JOINs required to combine data from different tables.	Allows embedding documents; JOINs required when using identifiers to link to other documents.	JOINs are required, but they are handled differently than relational databases.
No native reasoning.	No native reasoning.	Native logical reasoning.