

Chapter

5

Wikidata in the Era of Generative AI: Interfaces, Techniques, and Applications

Marcelo Machado, João Pedro Porto Campos, Alexya Silva, Gabriel Banaggia, and Guilherme Lima

Abstract

We present an introduction to the Wikidata knowledge graph with emphasis on the interfaces and techniques used to build generative AI applications. Wikidata is one of the largest knowledge graphs available on the Web and plays a crucial role today as a provider of authenticated information and equitable representation of world knowledge to AI applications. We start by presenting Wikidata itself, its data model, RDF encoding, and access interfaces, and then shift the focus to techniques and applications that combine LLMs and Wikidata. Two perspectives are explored. In the Wikidata for AI perspective, we discuss how Wikidata can be used to ground and contextualize AI applications, while in the AI for Wikidata perspective, we discuss the ways in which AI models can assist with the construction, enrichment, verification, and curation of Wikidata itself.

Resumo

Apresentamos uma introdução ao grafo de conhecimento Wikidata com foco nas interfaces e técnicas utilizadas para construção de aplicações de IA generativa. O Wikidata é um dos maiores grafos de conhecimento disponíveis na Web e desempenha hoje um papel crucial como fornecedor de informação autenticada e representação equitativa de conhecimento de mundo para aplicações de IA. Começamos apresentando o Wikidata em si, seu modelo de dados, codificação RDF e interfaces de acesso, e em seguida mudamos o foco para técnicas e aplicações que utilizam LLMs juntamente com o Wikidata. São exploradas duas perspectivas. Na perspectiva Wikidata para IA, discutimos como o Wikidata pode ser usado para ancorar factualmente e contextualizar aplicações de IA. Já na perspectiva IA para Wikidata, discutimos as formas pelas quais modelos de IA podem ajudar na construção, enriquecimento, verificação, e curadoria do próprio Wikidata.

5.1. Introduction

Wikidata¹ [Vrandečić and Krötzsch 2014] is a sister project of Wikipedia² and it is also one of largest bases of structured knowledge publicly-available on the Web. Strictly speaking, Wikidata is a knowledge graph or KG, that is, a database structured as a (directed) graph in which nodes represent entities and edges represent relationships between pairs of entities. But there is more to Wikidata than just the knowledge graph. As Wikipedia, Wikidata is a Wikimedia Foundation³ project maintained by a large community of volunteers (at the time of writing 43,357 active users and 326 bots⁴). Since its launch in 2012, Wikidata has grown into one of the largest general-purpose knowledge graphs, providing multilingual, machine-readable descriptions of entities and their relationships which are reused across other Wikimedia Foundation projects. For example, some of the data of the “infoboxes” of Wikipedia come directly from Wikidata [Sáez and Hogan 2018]. Many other projects also reuse Wikidata content including digital libraries, cultural heritage institutions, and countless third-party applications.

More recently, the role of Wikidata as a source of authoritative and equitable knowledge has taken on a new significance with the rise of generative artificial intelligence (AI) and large language models (LLMs). LLMs have exhibited remarkable text manipulation and reasoning abilities, often far surpassing those of traditional natural language processing methods. At the same time, LLMs are inherently stochastic and are known to suffer from hallucinations [Ji et al. 2023], biases [Navigli et al. 2023], and overreliance on outdated information [Mousavi et al. 2026]. So, while LLMs have enabled applications to navigate and process large quantities of text, they still lack grounding: their outputs while seemingly plausible might be factually incorrect, biased, or simply outdated. This is where Wikidata comes in. It provides precisely the kind of structured, authoritative, equitable, and up-to-date knowledge needed to ground the output of LLMs.

But using Wikidata as a source of trusted information for AI systems is just one side of the story. Another interesting possibility is leveraging the natural language understanding and generation capabilities of LLMs to support the development and growth of Wikidata itself. For instance, LLMs can be used to extract information from unstructured sources in order to predict missing entities or relationships in the graph, enrich statements with references and qualifiers, and help in identifying incorrect or misleading statements.

In this chapter, we present Wikidata in detail and discuss how emerging AI techniques can be used to construct richer AI applications using or targeting Wikidata. The chapter is divided into two parts. The first part (Section 5.2) presents the Wikidata data model, RDF encoding, and main access interfaces, including official ones (SPARQL, REST APIs, and data dumps) but also alternatives interfaces (TPF and KIF). The second part (Section 5.3), discusses the intersection between Wikidata and generative AI from the complementary perspectives mentioned to above. In *Wikidata for AI* (Section 5.3.1), we discuss how the knowledge graph can be used to ground and contextualize AI applications. In *AI for Wikidata* (Section 5.3.2), we discuss the ways in which AI models can assist with

¹<https://www.wikidata.org/>

²<https://www.wikipedia.org/>

³<https://wikimediafoundation.org/>

⁴<https://www.wikidata.org/wiki/Special:Statistics>

the construction, enrichment, verification, and curation of the knowledge graph. At the end of the chapter, we discuss some projects maintained by the Wikidata community which aim to ease the integration of Wikidata with generative AI systems. These include Wikidata Embedding and Vector Database projects, for semantic retrieval, and the Wikidata Model Context Protocol (MCP), for interfacing with AI agents.

5.2. The Wikidata Knowledge Graph

The Wikidata knowledge graph is the central repository of structured data for Wikimedia Foundation projects, including Wikipedia, and beyond. Although the user interface (UI) of Wikidata is organized in pages describing particular subjects, under the hood Wikidata is structured as a graph. The building blocks of this graph are entities and statements and the pages we see in Wikidata’s UI are essentially human-friendly way to visualize and edit parts of this graph.

In this section, we take a deep dive into the structure of the Wikidata knowledge graph. We start in by presenting in detail Wikidata’s conceptual data model. We then shift to the technologies on top of which this data model is implemented. One of these technologies is RDF (Resource Description Framework [Kellogg et al. 2026]) which is at the core of the Semantic Web standard⁵. RDF is a graph-based representation language that provides a vocabulary and constructors for describing sets of triples denoting directed graphs. The Wikidata infrastructure leverages RDF and other Semantic Web technologies (such as SPARQL) to encode and provide standard access interfaces to the Wikidata knowledge graph. As we will see, the Wikidata knowledge graph can also be accessed through other non-standard means, such as via data dumps or REST APIs, which might be more convenient depending on the application.

5.2.1. Wikidata Data Model

The data model of Wikidata consists of entities and statements about entities.⁶ In the Wikidata user interface, statements are grouped into “entity pages”. Figure 5.1 depicts a part of the page of entity Q1638425, which stands for the Brazilian city of Lavras⁷, located in the state of Minas Gerais. (At the time of writing, there are more than 123 million entity pages like this one in Wikidata.) Every Wikidata entity has a label, a description, and one or more aliases in a given language. In Figure 5.1, these are shown in the header at the top of the page. So, the English label of entity Q1638425 is “Lavras”, its English description is “town and municipality in the state of Minas Gerais, Brazil”, and it as a single English alias “Arraial de Santana das Lavras do Funil”.

After the header comes the “Statements” section which groups statements about the entity being described. A *statement* consists of two parts: subject and snak. The *subject* is the entity about which the statement is made. The *snak* is the statement’s claim. It associates a property with either a specific value, some unspecified value, or no value.

In Figure 5.1, two statements about the population of Lavras are depicted. These can be read as follows:

⁵https://en.wikipedia.org/wiki/Semantic_Web_Stack

⁶<https://www.mediawiki.org/wiki/Wikibase/DataModel>

⁷<https://www.wikidata.org/wiki/Q1638425>

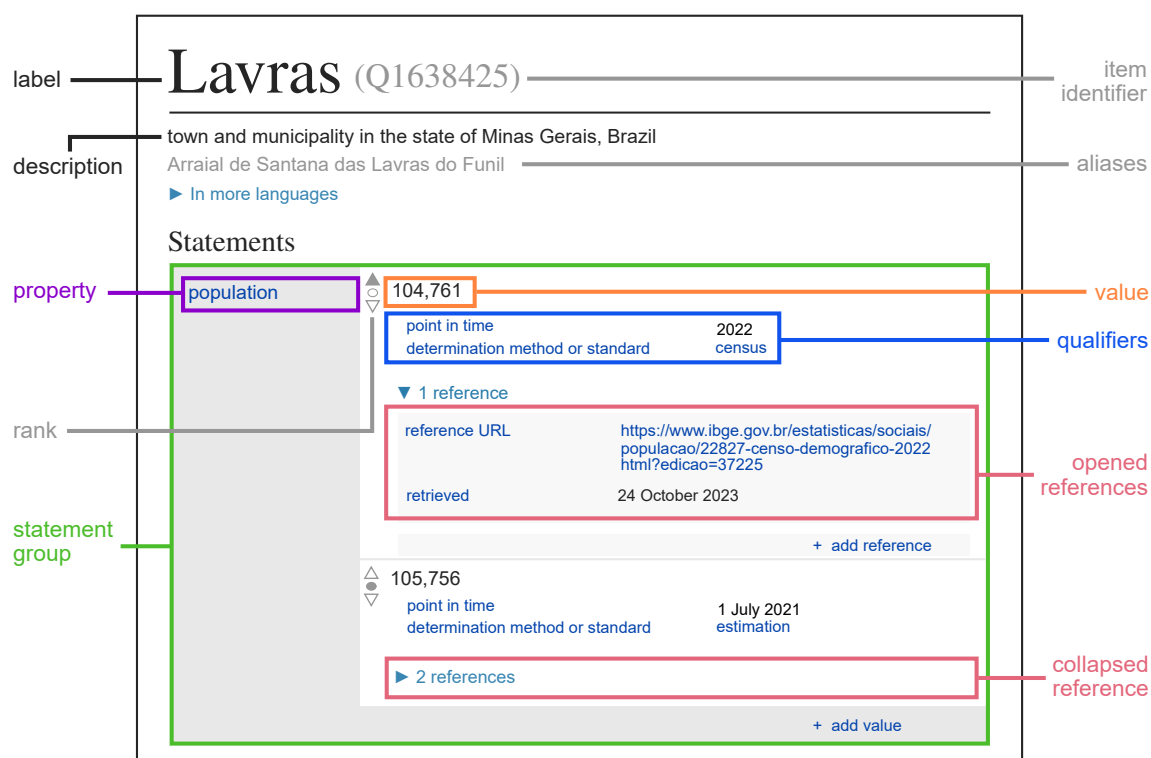


Figure 5.1: Part of Wikidata’s entity page of Lavras (Q1638425). (Adapted from [Odell et al. 2022].)

1. “Lavras has a population of 104,761”; and
2. “Lavras has a population of 105,756”.

The subject of statements (1) and (2) is the same, the entity “Lavras” (Q1638425). Their snak is of the form property-value. The property of both is “population” (P1082). The value of statement (1) is “104,761” and the value of statement (2) is “105,756”. Note that the data model distinguishes between two types of entities: *items* (identified with “Q”) and *properties* (identified with “P”). Lavras (Q1638425) is an example of an item, while population (P1082) is an example of a property. Only properties can occur as the first component of snaks.

Figure 5.1 also shows the qualifiers and references associated with each statement. These appear the statement value in the boxes “qualifiers” and “references”. *Qualifiers* are extra snaks which qualify what is being said by the statement’s main snak. *references* (or *reference records*) are sets of snaks which keep provenance information. In the case of Figure 5.1, the qualifiers convey information which is crucial to interpret the statement, namely, the point in time at which the population counts were performed and method used (census in the case of statement (1) and estimation in the case of statement (2)).

The last piece of metadata associated with statements is the *rank* which can be either “preferred”, “normal”, or “deprecated”. In Figure 5.1, the rank is represented symbolically by the two triangles and circle which occur on the left of the statement’s value. A filled upper triangle means preferred rank; a filled circle means normal rank; and a filled lower

triangle means deprecated rank. As can be seen in Figure 5.1, statement (1) has a preferred rank and (2) has normal rank.

As we said previously, the page depicted in Figure 5.1 is just a human-friendly way to visualize statements about a given subject. Under the hood, statements are actually stored as the edges of a graph, whose nodes are Wikidata entities (items or properties). To concretely describe this graph, Wikidata adopts RDF. It encodes the concepts of the data model (item, properties, statements, snaks, etc.) as an RDF triples, which can be manipulated and queried using other standard Semantic Web technologies. We discuss RDF and the encoding of the Wikidata knowledge graph in RDF in detail next.

5.2.2. Resource Description Framework (RDF)

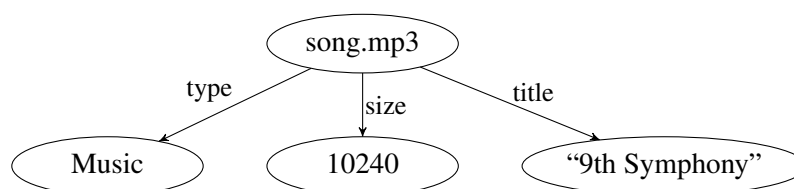
One of the main standards of the Semantic Web is the *Resource Description Framework* (RDF) [Wood et al. 2014]. RDF is a graph-based representation language which provides a vocabulary and constructors for describing sets of triples denoting directed graphs. These triples are called *subject-predicate-object (spo) triples* and have the form

$$\langle \text{subject}, \text{predicate}, \text{object} \rangle.$$

An *spo* triple denotes a labeled edge of a directed graph. For instance, the three *spo* triples below describe the type, size in bytes, and title associated with an audio file:

$\langle \text{song.mp3}, \text{type}, \text{Music} \rangle,$
 $\langle \text{song.mp3}, \text{size}, 10240 \rangle,$
 $\langle \text{song.mp3}, \text{title}, \text{9th Symphony} \rangle.$

Here is a diagram depicting the graph corresponding to these three *spo* triples:



Each triple becomes a directed edge in the graph. The subject of the triple is the source of the edge, the object is its target, and the predicate is the edge label. So, the resulting graph is a directed edge-labeled graph.

Of course, this is just an abstract example. In an actual RDF graph, the components of the triples, that is, the labels of nodes and edges in the graph, must obey a specific format (or *syntax*). A popular plain text syntax for RDF is Turtle (Terse RDF Triple Language) [Carothers and Prud'hommeaux 2014]. Turtle is an alternative to the widely supported but more cumbersome RDF/XML syntax. (All of our RDF examples will be written in Turtle.)

The nodes in an RDF graph can be of three kinds: IRI nodes, literal nodes, or blank nodes. An *IRI node* is a node labeled by an IRI (Internationalized Resource Identifier); an IRI is an URI extended with support for Unicode characters. Here is an example IRI:

<https://pt.wiktionary.org/wiki/maça>

Note the non-ASCII characters “ç” and “ã” above. In an RDF graph, IRIs are used to denote arbitrary *resources*, such as physical objects, documents, images, abstract concepts, numbers, strings, etc.

The second kind of node which can appear in an RDF graph is the literal node. A *literal node* is any node labeled by a literal, that is, a quoted string denoting a value in-place. In the Turtle syntax, literals are written enclosed in double quotes optionally accompanied with language tags and type tags. Here are three RDF literals (written in Turtle):

```
"Rock",    "maçã"@pt,    "10240"^^xsd:integer.
```

The first is a plain literal, the second is a literal with a language tag (Portuguese in this case), and the third is a literal with a type tag, also called typed literal. The prefix `xsd:`, which occurs in the third literal above, stands for the namespace where the type tag `integer` is defined. We will have more to say about namespaces in a moment.

The third kind of node that can occur in an RDF graph is the blank node. A *blank node* has no specific denotation; it is an unlabeled node which is neither an IRI nor literal node. Blank nodes are often used as dummy nodes whose only purpose is to ease the encoding of complex structures in graphs.

In an RDF triple $\langle s, p, o \rangle$, the subject s must be an IRI or blank node, the predicate p must be an IRI, and the object o must be an IRI, literal, or blank node. When we assert an RDF triple we are saying informally that “the relationship indicated by the predicate holds between the resource denoted by the subject and the resource or value denoted by the object”.

Figure 5.2 presents a possible Turtle encoding for the graph describing the audio file “song.mp3” depicted a few paragraphs ago.⁸ Each edge in the graph (that is, triple in the set) becomes a corresponding triple in the Turtle document (lines 5–7). The subject, predicate, and object parts are separated by spaces and each triple is terminated with a dot (.). Note that IRIs are written between angle brackets $\langle \dots \rangle$.

```
1 @base <http://example.org/data/> .
2 @prefix ex: <http://example.org/#> .
3 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
4 <song.mp3> ex:type ex:Music .
5 <song.mp3> ex:size "10240"^^xsd:integer .
6 <song.mp3> ex:title "9th Symphony" .
```

Figure 5.2: Turtle encoding of the RDF triples describing “song.mp3”.

The directives in lines 1–3 of Figure 5.2 define the namespaces of the document. That is, they specify how relative and prefixed IRIs are to be interpreted. The directive `@base` defines the base IRI of the whole Turtle document. Any relative IRI occurring in the document is to be interpreted as relative to this base IRI. So, for instance, the occurrences of `<song.mp3>` in lines 4–6 of Figure 5.2 are to be interpreted as the absolute IRI:

`http://example.org/data/song.mp3`

⁸Sometimes literals are represented as rectangles when depicting an RDF graph. Incidentally, we did not follow this convention when depicting the graph describing “song.mp3”.

The directive `@prefix` introduces prefix labels which can be used to abbreviate IRIs. So, in Figure 5.2, the labels `ex:` and `xsd:` stand for the following IRI prefixes:

```
http://example.org/#  
http://www.w3.org/2001/XMLSchema#
```

Note that prefixed IRIs are *not* written between angle brackets.

Turtle has a number of other features. Some of these are useful for avoiding tedious repetitions within the document. For instance, Figure 5.3 uses some of these features to define in fewer lines the same graph as Figure 5.2. Note the semicolons “;” in lines 3 and 4 of Figure 5.3. These are used to separate predicate-object pairs referring to the same subject; in this case, the IRI `http://example.org/data/song.mp3`. This type of construction is called a *predicate list*. Note also the typed literal `10240` in line 4. Some common types of literals, such as integers and floating-point numbers, can be written in Turtle in their usual syntaxes.

```
1 @prefix ex: <http://example.org/#> .  
2 <http://example.org/data/song.mp3>  
3   ex:type ex:Music ;  
4   ex:size 10240 ;  
5   ex:title "9th Symphony" .
```

Figure 5.3: Example of Figure 5.2 rewritten using predicate lists.

It is important to realize that an RDF graph is just a data structure, and that Turtle, RDF/XML, and the other syntaxes for RDF are just ways to encode this data structure in a string. More importantly, the meanings of the labels which occur in an RDF graph are not defined in the RDF standard. These labels might indicate something to the person reading them, but to a computer that understands only RDF these labels are meaningless strings. They serve only to identify and distinguish specific components of the graph.

There is another Semantic Web standard, called RDF Schema (RDFS) [Hayes and Patel-Schneider 2014], which is defined on top of RDF and which specifies a basic vocabulary (set predefined terms) and associated meanings. This vocabulary can be found in the namespace:

```
http://www.w3.org/2000/01/rdf-schema#
```

RDFS provides a set of basic terms which can be used to define more specialized terms whose meaning is given by describing their interrelations with other terms. The Wikidata RDF encoding, which we discuss next, reuses very little of standard vocabularies, such as RDFS. Instead it adopts its own vocabulary for most representation needs.

5.2.3. Wikidata RDF Encoding

The RDF encoding of Wikidata’s data model is an extension of the encoding used by Wikibase⁹. The format varies slightly depending on whether it is used in a data dump or observed from Wikidata’s query service. The version we describe here is that of the query service.

⁹https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

Throughout this section we will use two statements about the toxicity of the chemical compound benzene (Q2270) as running examples. These two statements are shown in Figure 5.4 as part of benzene’s entity page. Both concern property median lethal dose, or LD50, (P2240) which is standard toxicity measure used in chemistry.¹⁰ The first statement (1) says that the median lethal dose of benzene is approximately 4,700 milligrams per kilogram when the route of administration is oral and the target taxon is laboratory mouse. Note that route of administration and target taxon are given in the statement’s qualifier. Also, the statement has two references, which are detailed in the figure and which give its provenance.

The second statement (2) shown in Figure 5.4 is similar. It says that the median lethal dose of benzene is approximately 88 milligrams per kilogram when the route of administration is intravenous infusion and defusion and the target taxon is rabbit. Again, the route of administration and target taxon are given in the statement’s qualifiers. The second statement has a single reference which is not detailed in the figure.

The rationale for using these two statements as examples is that they illustrate some representation features of Wikidata which are not present in standard vocabularies, such as RDF, RDFS, and OWL [W3C OWL WG 2012], nor popular knowledge graphs, such as DBpedia [Lehmann et al. 2015]. One of these features is the native support for numerical data values with associated units (mg/kg) and precision ranges (± 1).

benzene (Q2270)

hydrocarbon compound consisting of a 6-sided ring
 C6H6 | [6]annulene | BNZ | cyclohexa-1,3,5-triene | 1,3,5-cyclohexatriene | ...
 ► In more languages

Statements

median lethal dose (LD50) Δ 4,700 ± 1 milligram per kilogram ∇

route of administration oral administration
 applies to taxon laboratory mouse

▼ 2 references

stated in	PubChem
PubChem CID	241
language of work or name	English
title	benzene (English)
retrieved	12 April 2024
reference URL	https://www.cdc.gov/niosh-rtecs/CY155CC0.html

+ add reference

Δ 88 ± 1 milligram per kilogram ∇

route of administration intravenous infusion and defusion
 applies to taxon rabbit

► 1 reference

+ add value

Figure 5.4: Part of Wikidata’s entity page of benzene (Q2270).

¹⁰More precisely, LD50 is a toxicity unit that measures the dose of a substance that is required to kill half the members of a given population.

Back to the RDF encoding, Wikidata adopts a two-level representation of statements. The first level, called *truthy*, keeps a shallow representation of the statement as a single RDF triple. For example, the truthy encoding of statement (1) of Figure 5.4, namely, “benzene (Q2270) has an LD50 (P2240) of 4,700±1 mg/kg”, consists of the single triple:

(wd:Q2270) $\xrightarrow{\text{wdt:P2240}}$ "4700"^^xsd:decimal (†)

The namespace wd: indicates an entity and wdt: indicates a truthy application of a property. Some statements are fully characterized at the truthy level. But, as illustrated by (†), this is not always the case. Note that the unit, lower-, and upper-bounds associated with the value 4700 are not represented in (†). In general, when the statement’s value is a structured data-value, like a quantity or time value, a single literal is used to represent it at the truthy level. This is the so called *simple value* of the statement. For quantity values, the simple value is just the numerical amount.

The second level of the encoding keeps the full representation of the statement. It uses reification to capture the *deep value* of the statement plus its qualifiers, references, and rank. Figure 5.5 depicts the full representation of statement (1) of Figure 5.4 considering only one qualifier and one reference record.

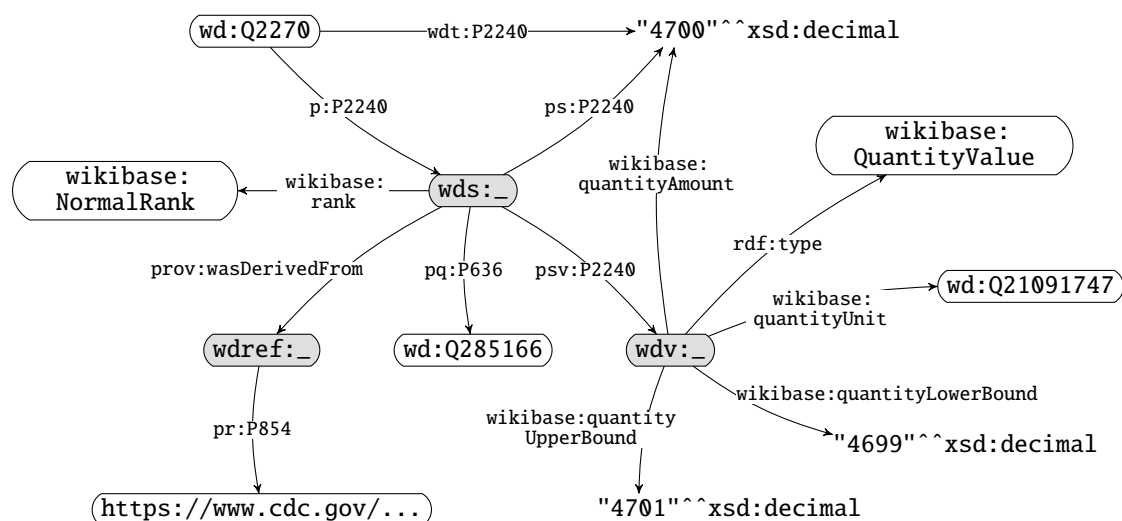


Figure 5.5: RDF representation of the statement “Benzene (Q2270) has an LD50 (P2240) of 4,700 ± 1 mg/kg (Q21091747)” considering only the qualifier “route of administration (P636) is oral administration (Q285166)” and the reference record “reference URL (P854) is <https://www.cdc.gov/niosh-rtecs/CY155CC0.html>”.

In Figure 5.5, the shaded nodes are the reified ones. The single underscore (_) indicates that their ids are opaque (hence not shown in the figure). Node wds:_ represents the statement itself. Predicates p:P2240 and ps:P2240 are used to connect the subject “benzene” (wd:Q2270) to the statement and the statement to its simple value, i.e., the number 4700 in decimal notation.

The deep value of the statement is represented by node wdv:_ . It has type wikibase:QuantityValue and is connected to the unit mg/kg (wd:Q21091747), the lower-

bound 4699, and the upper-bound 4701. The rank of the statement is connected via predicate `wikibase:rank`.

In Figure 5.5, predicate `pq:P636` connects to the statement the qualifier “route of administration” (P636) with value “oral administration” (`wd:Q285166`). Similarly, predicate `prov:wasDerivedFrom` connects to the statement the reference record represented by node `wdref:_` whose content (the snak “reference URL” (P854) with value “<https://www.cdc.gov/...>”) is encoded using predicate `pr:P854` and a (simple) IRI value.

5.2.4. Wikidata Access Interfaces

We now move to the interfaces for accessing the Wikidata knowledge graph. We cover the traditional interfaces first, including the official SPARQL query service, REST APIs, and data dumps. Later, we cover more exotic interfaces, such as the triple pattern fragments (TPF) interface and the one provided by KIF [Lima et al. 2024].

5.2.4.1. SPARQL

So far we have been working with fragments of RDF graphs consisting of some small number of nodes and edges. When working with such small graphs one hardly needs to worry about efficiency. In practice, however, RDF graphs are rarely small. It is not uncommon to find applications that deal with graphs containing millions or even billions of nodes and edges. At the time of writing, for example, the Wikidata knowledge graph contains close to 9 billion triples. At this scale simple tools will not do. One needs specialized RDF databases, called *triplestores*, designed to efficiently store and process large amounts of RDF data. Currently, the Wikidata relies on Blazegraph¹¹, a high-performance RDF triplestore. But there are plans to migrate the infrastructure to other RDF solutions, as the open-source version of Blazegraph is no longer maintained.

SPARQL (SPARQL Protocol and RDF Query Language) [Harris and Seaborne 2013] is to triplestores what SQL is to relational databases. It is a declarative interface for querying and manipulating the contents of the graph stored in the database. There are two basic types of SPARQL queries: graph pattern matching and graph navigation. We discuss each of these two types of queries next, using the RDF graph of Figure 5.6 as an example target graph. This graph describes the film *Taxi Driver*.

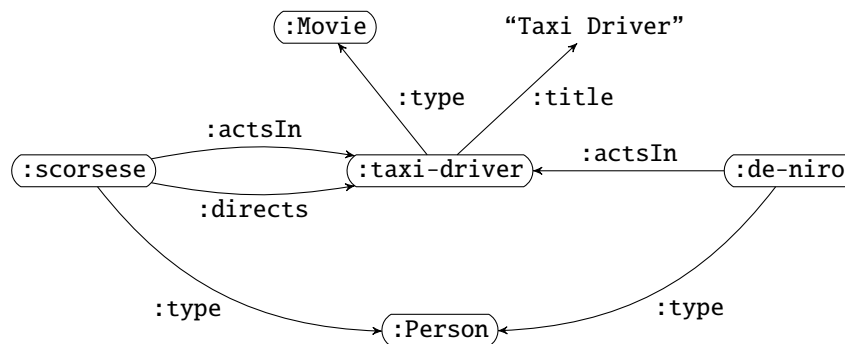


Figure 5.6: Example RDF graph describing the movie *Taxi Driver*.

¹¹<https://blazegraph.com/>

Graph pattern matching A graph pattern matching query is a query that searches for a given (bounded) structural pattern in the graph. In SPARQL, these structural patterns are expressed as *triple patterns*, which are RDF triples in which the subject, predicate, or object may be a variable (name starting with “?”). A *basic graph pattern* (BGP) is the combination of one or more triple patterns.

Consider the RDF graph depicted in Figure 5.6, which describes the film *Taxi Driver*. Suppose we want to list all pairs of distinct persons who acted in this film. We can specify this query in SPARQL as follows:

```

1 PREFIX: <http://example.org/#>
2 SELECT ?x1 ?x2
3 WHERE {
4   ?x1 :actsIn ?x3 . ?x1 :type :Person .
5   ?x2 :actsIn ?x3 . ?x2 :type :Person .
6   ?x3 :title "Taxi Driver" . ?x3 :type :Movie .
7   FILTER (?x1 != ?x2)
8 }
```

This query expresses a BGP consisting of six triple patterns. It select all pairs of labels x_1 and x_2 (line 2) with $x_1 \neq x_2$ (line 7) such that, for some label x_3 , each of the following triples occur in the graph:

$\langle x_1, :actsIn, x_3 \rangle$	and	$\langle x_1, :type, :Person \rangle$	(line 4)
$\langle x_2, :actsIn, x_3 \rangle$	and	$\langle x_2, :type, :Person \rangle$	(line 5)
$\langle x_3, :title, "Taxi Driver" \rangle$	and	$\langle x_3, :type, :Movie \rangle$	(line 6)

If we apply this query to the graph of Figure 5.6 we get the following result:

```

1 ?x1      ?x2
2 -----
3 :scorsese :de-niro
4 :de-niro  :scorsese
```

That is, our *answer set* consists exactly of two pairs of labels: one where x_1 is :scorsese and x_2 is :de-niro (line 3 in the result), and another where x_1 is :de-niro and x_2 is :scorsese (line 4 in the result).

Figure 5.7 shows the results of running a similar query over Wikidata. The figure depicts an screenshot of the Wikidata SPARQL Query Service¹². The query in this case references the items Taxi Driver (Q47221), person (Q5), and the properties cast member (PP161) and human (Q5). The truthy namespace wdt: is used for matching over the truthy graph, and the SERVICE clause is used to obtain the labels of the retrieved entities via Wikidata’s custom label service.

Graph navigation The other common kind of query supported by SPARQL are graph navigation queries. These are queries that navigate the topology of the graph through paths

¹²<https://query.wikidata.org/>

The screenshot shows the Wikidata Query Service interface. At the top, there are navigation links: Examples, Help, More tools, and Query Builder. The language is set to English. The query editor contains the following SPARQL query:

```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3
4 SELECT DISTINCT ?x1 ?x1Label ?x2 ?x2Label WHERE {
5   wd:Q47221 wdt:P161 ?x1 . # taxi driver -[cast member]-> ?x1
6   ?x1 wdt:P31 wd:Q5 .      # ?x1 -[instance of]-> human
7
8   wd:Q47221 wdt:P161 ?x2 . # taxi driver -[cast member]-> ?x2
9   ?x2 wdt:P31 wd:Q5 .      # ?x2 -[instance of]-> human
10
11  FILTER (?x1 != ?x2)
12  SERVICE wikibase:label {bd:serviceParam wikibase:language "en".}
13 }

```

The results section shows 156 results in 767 ms. The results are displayed in a table with columns: x1, x1Label, x2, and x2Label. Each cell contains a magnifying glass icon followed by the Wikidata ID and the label.

x1	x1Label	x2	x2Label
wd:Q356303	Albert Brooks	wd:Q3229673	Leonard Harris
wd:Q356303	Albert Brooks	wd:Q3783517	Harry Northup
wd:Q20789791	Peter Savage	wd:Q3229673	Leonard Harris
wd:Q356303	Albert Brooks	wd:Q960676	Joe Spinell
wd:Q356303	Albert Brooks	wd:Q973065	Victor Argo
wd:Q20789791	Peter Savage	wd:Q3783517	Harry Northup

Figure 5.7: Screenshot of the Wikidata SPARQL Query Service showing the results for “All pairs of distinct persons who acted in *Taxi Driver*”.

of potentially arbitrary lengths. A *path query* is a basic navigation query of the form $x \xrightarrow{\alpha} y$ that retrieves all pairs $\langle x, y \rangle$ such that there is a path in the graph from x to y which satisfies a given condition α .

For instance, the path queries

$$x \xrightarrow{\text{:directs}} y \quad \text{and} \quad x \xrightarrow{\text{:actsIn} \circ \text{:type}} y$$

search, respectively, for all nodes x and y such that there is an edge labeled `:directs` from x to y , and for all nodes x and y such that there is path of size two from x to y where the first edge in the path is labeled `:actsIn` and the second edge is labeled `:type`. (Recall that a *path* in a graph is a sequence of edges joining a sequence of nodes.).

We can specify the latter query in SPARQL as follows:

```

SELECT ?x ?y
WHERE {
  ?x (:actsIn/:type) ?y .
}

```

Here the symbol “/” stands for chaining and corresponds to the “o” we used in the abstract notation. If we apply this SPARQL query to the graph of Figure 5.6 we get answer:

?x	?y

:scorsese	:Movie
:de-niro	:Movie

For a more interesting example, consider the path query

$$x \xrightarrow{(:\text{actsIn} \circ :\text{actsIn}^-)^+} y.$$

This query searches for all actors who have a finite collaboration distance¹³, that is, actors x and y such that x and y are co-stars in the same movie (distance 0), or x and y are co-stars of the same z in some movie (distance 1), or x is co-star of some z_1 in some movie and y is co-star of some z_2 in some movie and z_1 and z_2 are co-stars in the same movie (distance 2), and so on. The following SPARQL code implements this query:

```
SELECT ?x ?y
WHERE {
  ?x (:actsIn/^:actsIn)+ ?y .
}
```

Here the “^” denotes the inverse of an edge label (it corresponds to operator “-” used in the abstract notation) and “+” stands for one or more repetitions of the pattern immediately preceding it.

Suppose we add the following triples to the graph of Figure 5.6:

$\langle :de-niro, :actsIn, :cassino \rangle$ and $\langle :sheron-stone, :actsIn, :cassino \rangle$.

If we run the above SPARQL query over this updated graph we get the answer:

?x	?y

:scorsese	:de-niro
:de-niro	:scorsese
:de-niro	:sharon-stone
:sharon-stone	:de-niro
:scorsese	:sharon-stone
:sharon-stone	:scorsese

Figure 5.8 shows the results of running a query similar to the last one over Wikidata. This time we use the QLever SPARQL query interface¹⁴ which, although not being an official Wikidata endpoint, is significantly faster for this type of query.

SPARQL has many more features, such as groups, options, alternatives, filters, etc., which can be combined with basic graph patterns and navigation operators in order

¹³When x is the actor Kevin Bacon this distance is known as the *Bacon distance* or *Bacon degree*. See: https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

¹⁴<https://qllever.dev/wikidata/>

Figure 5.8: Screenshot of QLever SPARQL query interface showing the results for “All actors who have a finite collaboration distance with Robert De Niro”.

to build complex queries. Our goal here was to provide a glimpse of SPARQL queries. For a detailed introduction to SPARQL, see [Hitzler et al. 2010]. Besides SPARQL, there are many other languages for querying graphs. See [Angles 2012] for a general discussion of these. For a practical introduction to graph databases and triple stores, we recommend [Robinson et al. 2013].

5.2.4.2. REST APIs

As an alternative to SPARQL, Wikidata offers REST APIs for searching and fetching information about entities and properties. Three different REST APIs are available [Wikidata 2026a]. The first one is MediaWiki Action API¹⁵, which is intended for obtaining general information about Wikidata entities in JSON format. Currently, requests to the MediaWiki Action API can carry up to 50 entities per request.

Here is an example call to the MediaWiki Action API:

¹⁵<https://wikidata.org/w/api.php>

<https://www.wikidata.org/w/api.php?action=wbgetentities&ids=Q47221&format=json>

This request fetches all data necessary to build the page of entity Taxi Driver (Q47221), namely, its labels, descriptions, and aliases together with all statements (claims) whose subject is the entity Q47221. Here is part of the response to this request:

```
{
  "entities": {
    "Q47221": {
      "modified": "2026-09-12T19:52:54Z",
      "type": "item",
      "id": "Q47221",
      "labels": {
        "en": {
          "language": "en",
          "value": "Taxi Driver"
        },
        ...
      },
      "descriptions": {
        "en": {
          "language": "en",
          "value": "1976 film directed by Martin Scorsese"
        },
        ...
      },
      "aliases": {
        ...
      },
      "claims": {
        "P57": [
          {
            "mainsnak": {
              "snaktype": "value",
              "property": "P57",
              "datavalue": {
                "value": {
                  "entity-type": "item",
                  "id": "Q41148"
                },
                ...
              },
              "datatype": "wikibase-item"
            },
            "type": "statement",
            "id": "q47221$EB940392-6851-48AB-A348-4C16A35471E8",
            "rank": "normal",
            "references": [
              {
                "snaks": {
                  "P248": [
                    {
                      "snaktype": "value",
                      "property": "P248",
                      "datavalue": {
```


Turtle and N-Triples formats. There are two versions of the RDF dumps. The “all” version corresponds to the full graph, which besides information about entities and their relationships carries the qualifiers, references, ranks and other metadata. In contrast, the “truthy” version corresponds to the truthy graph (see Section 5.2.3) which contains only information about the entities and their relationships. The choice between “all” and “truthy” depends on the application, or on how much detail one is willing to lose in exchange for a significant reduction in the size of the graph.

There are also incremental dumps, or dumps of additions and changes, which contain what was added to the graph in the last 24 hours. Applications that need to be kept in sync with the latest version of the graph should use these.

5.2.4.4. Triple Pattern Fragments

We now move to more experimental access interfaces, starting with TPF. TPF (Triple Pattern Fragments) [Verborgh et al. 2016] is a lightweight interface to RDF graphs. They are part of the Linked Data Fragments (LDF) initiative [Verborgh et al. 2014] and were proposed as an intermediate alternative to RDF data dumps and SPARQL endpoints. LDF itself is a framework for the conceptual analysis of Linked Data interfaces. According to LDF, any such interface publishes only parts, or (*linked data*) *fragments*, of a given knowledge graph. These fragments are considered the “units of response” of the interface and are assumed to consist of three things:

1. Data: a subset of the triples of the graph;
2. Metadata: triples describing the data; and
3. Controls: links and forms that can be used to retrieve other fragments of the same or other knowledge graphs.

For example, a data dump of a knowledge graph can be described as single linked data fragment where the data is the whole content of the dump, the metadata consists of things like version, author, etc., and the controls are empty. Similarly, the response to a SPARQL CONSTRUCT query can be seen as a fragment where the data are the resulting triples, the metadata is empty, and the controls are any parameters used to paginate the result, such as limits and offsets.

A TPF interface provides access to an RDF graph based on a single triple-pattern. A *triple-pattern* is a pattern of the form (s, p, o) where s , p , and o are either fixed values (URIs or literals) or anonymous variables. When given a triple-pattern, the TPF interface responds with a fragment in which (i) the data are triples in the graph that match the pattern; (ii) the metadata are an estimate of the total number such triples; and (iii) the controls are a hypermedia form that allows clients to retrieve other fragments matching the same pattern.

Wikidata and some other popular knowledge graphs, such as DBpedia [Lehmann et al. 2015], provide TPF interfaces. Other public TPF interfaces can be found here¹⁸.

We can query the Wikidata TPF interface by submitting a GET HTTP request to an URL such as this one:

¹⁸<https://linkeddatafragments.org/data/>

<https://query.wikidata.org/bigdata/ldf?predicate=wdt:P31\&object=wd:Q5>

This request asks for every triple in Wikidata whose predicate component is `wdt:P31` (instance of) and object component is `wd:Q5` (human). That is, it selects the triples matching the pattern

$$(s, \text{wdt:P31}, \text{wd:Q5})$$

for any value of s . In other words, the triples such that subject s is an instance of human. The response to this specific request is a single linked data fragment containing usually 100 triples matching the pattern. For instance, if you open the above URI in a Web browser, an HTML page with the matched triples will be shown. At the bottom of the page, you will see links (controls) to other pages (fragments) which contain the rest of the result. The actual format of the response depends on the value of the “Accept” header provided to the server in the HTTP request. This can be set to any of the popular RDF serialization formats, such as Turtle and JSON-LD.

Some TPF interfaces also support quad-patterns of the form (s, p, o, g) which take an additional parameter g specifying a named graph of an RDF dataset. If not set, g is assumed to be the default graph. Note that the Wikidata RDF encoding does not use named graphs.

The advantage of TPF over other Linked Data interfaces is that it offers some query capability while being extremely lightweight. For some applications, it is more convenient than a data dump and provides a more reasonable Web API than SPARQL in the sense that it exposes only a limited interface to the underlying database. A further advantage has to do with caching. Because of their restricted syntax TPF requests are more cache-friendly than SPARQL requests. For instance, just by looking at the requested URI, it is straightforward for an HTTP proxy to determine whether two TPF requests are identical. The same cannot be said about SPARQL where there are different ways to write essentially the same query.

5.2.4.5. Knowledge Integration Framework

KIF (Knowledge Integration Framework) [Lima et al. 2024] is an open-source framework for knowledge integration based on Wikidata. It adopts the data model and vocabulary of Wikidata as a lingua franca for integrating heterogeneous knowledge sources. The integration done by KIF is virtual in the sense that the data model and vocabulary translations happen at query time, guided by user-provided mappings.

KIF is in essence a Python library (`kif-lib`)¹⁹ for evaluating queries over abstract engines and obtaining objects of the Wikidata data model as a result. KIF engines are implemented following a plugin architecture. There are various predefined (built-in) engine plugins which can be extended by applications and loaded dynamically into the library. The two basic types of KIF engines are stores and searchers.

Stores A KIF *store* is an interface to a knowledge source, typically but not necessarily a knowledge graph. The most common type of KIF store is the SPARQL store, which reads Wikidata-like statements from a given SPARQL endpoint. To create a SPARQL store, one needs specify the address of the target SPARQL endpoint and the SPARQL

¹⁹<https://pypi.org/project/kif-lib/>

mappings to be used to generate a concrete SPARQL query (i.e., one adhering to the RDF encoding of the target graph). Besides mappings for Wikidata itself, KIF comes with SPARQL mappings for DBpedia [Lehmann et al. 2015], PubChem (RDF) [Kim et al. 2023], UniProt [Consortium 2025], among others. New mappings can be added programmatically.

We can use the KIF command-line interface (CLI), which is distributed together with the library, to run a simple query over a KIF store. For instance:

```
1 $ kif filter --subject='wd.Q(7251)' --property=wd.doctoral_advisor
2 (Statement
3   (Item Alan Turing)
4   (ValueSnak (Property doctoral_advisor)
5     (Item Alonzo Church)))
```

The command on line 1 runs a KIF *filter* (a statement match query) using the default store plugin, *wikidata*, which targets the public Wikidata SPARQL Query Service (WDQS) and uses KIF’s Wikidata SPARQL mappings. This particular filter instructs KIF to search for statements whose subject is *wd:Q7251* (Alan Turing) and property is *wd:P184* (doctoral advisor). To evaluate a filter, the SPARQL store uses the provided mappings to compile it into a SPARQL query and then runs the query over the target endpoint. The resulting bindings are used to construct Wikidata-like statements. The returned statement in this case is shown on lines 2–5 and stands for the assertion “Alan Turing’s doctoral advisor is Alonzo Church”.

As we have seen, in the Wikidata data model, which is adopted by KIF, every statement consists of a subject (which can be an item or property) and a *snak*. The subject of the above statement is the item Alan Turing and its *snak* is a *value snak* (i.e., property-value pair) with property equal to *doctoral advisor* and value equal to the item Alonzo Church.

KIF distinguishes between unannotated statements, such as the one above, and annotated statements. The latter, besides a subject and a *snak*, also carry a set of qualifiers, a set of reference records, and a rank. For instance, the filter below matches *annotated statements* (due to the *--annotated* flag) with property *wd:P2054* (density) and value equal to 0.88 grams per cubic centimeter:

```
1 $ kif filter --property=wd.density \
2   --value='Quantity("0.88", wd.Q(13147228))' --annotated
3 (AnnotatedStatement
4   (Item benzene)
5   (ValueSnak (Property density) 0.88 ±0.01 g/cm³)
6   (QualifierRecord
7     (ValueSnak (Property temperature) 20 ±1 °C)
8     (ValueSnak (Property phase of matter) (Item liquid)))
9   (ReferenceRecordSet
10    (ReferenceRecord
11      (ValueSnak (Property HSDB ID) "35#section=TSCA-Test-Submissions")
12      (ValueSnak (Property stated in) (Item Hazardous Substances Data Bank))))
13   NormalRank)
```

The annotated statement shown on lines 2–13 asserts that “benzene’s density is 0.88±0.01 g/cm³”. Its qualifier record (lines 6–8) qualifies this assertion, i.e., says in addition that

this is the case when “the temperature is 20 ± 1 °C” and “the phase of matter is liquid”. Its reference record set (lines 9–12) contains a single reference record (lines 7–9) which indicates the provenance of the statement, namely, the entry with the given HSDB ID in the Hazardous Substances Data Bank. Finally, the value NormalRank for its rank (line 13) is the default one and means “neutral” (neither preferred nor deprecated).

More complex filters So far, we have used simple values to match the components of statements in filters. In KIF, we can also specify the desired subject, property, or value indirectly, by giving a constraint that it must satisfy. For instance:

```
$ kif filter \
  --subject='(wd.shares_border_with(wd.Argentina)|
             wd.shares_border_with(wd.Paraguay))&
             wd.located_in_or_next_to_body_of_water(wd.Atlantic_Ocean)' \
  --property='wd.official_language'
(Statement
 (Item Brazil)
 (ValueSnak (Property official language)
             (Item Portuguese)))
(Statement
 (Item Uruguay)
 (ValueSnak (Property official language)
             (Item Spanish)))
```

This filter matches statements such that (i) the subject “shares border with Argentina *or* Paraguay” *and* “is located in or next to the Atlantic Ocean”; and (ii) the property is wd:P37 (official language). The only matching subjects in this case are wd:Q155 (Brazil) and wd:Q77 (Uruguay) and their official languages, as indicated by the statements above, are wd:Q5146 (Portuguese) and wd:Q1321 (Spanish).

Besides single-hop checks, KIF supports constraints involving *n*-hop property paths. These can be combined via conjunction (&) and disjunction (|) operators, as shown above. Simple datatype checks and projection on the statement components (subject, property, value) are also supported.

Targeting other graphs To query a graph other than Wikidata, all we need to do is change the store plugin. For instance:

```
1 $ kif filter --store=dbpedia --subject='db.r("Alan_Turing")'
2   --value='(db.almaMater/db.city)(db.r("Princeton,_New_Jersey"))'
3 (Statement
4   (Item dbr:Alan_Turing)
5   (ValueSnak (Property dbo:doctoralAdvisor)
6             (Item dbr:Alonzo_Church)))
7 (Statement
8   (Item dbr:Alan_Turing)
9   (ValueSnak (Property doctoral advisor)
10             (Item dbr:Alonzo_Church)))
```

The --store=dbpedia argument instructs KIF to run the filter using the dbpedia store plugin, which targets the public DBpedia SPARQL endpoint and uses KIF’s DBpedia SPARQL mappings. The filter in this case matches statements whose subject is

dbr:Alan_Turing and value is “someone whose dbo:almaMater is located in the dbo:city of dbr:Princeton,_New_Jersey”.

Despite querying DBpedia, as expected, we get as a result two Wikidata-like statements (lines 3–10) asserting that “Alan Turing’s doctoral advisor is Alonzo Church”. We say these statements are “Wikidata-like” because although following Wikidata syntax, their components mostly have DBpedia URLs. This happens because KIF’s DBpedia SPARQL mappings do not attempt to convert entities in the DBpedia namespace into that of Wikidata. The exception are properties. If there exists a Wikidata property which is said to be the same as (`owl:sameAs`) a given DBpedia property in the DBpedia graph, then the generated SPARQL query will match it. This is why on line 9 we get a statement whose property is `wd:P184` (doctoral advisor) even though we are querying DBpedia.

Searchers Besides stores, the other kind of engine in KIF are searchers. A KIF *searcher* is an interface to a similarity search method within a namespace. For example, given a string, the `wikidata-wapi` search plugin looks in the Wikidata namespace for entities with a similar label or alias:

```
1 $ kif search --search=wikidata-wapi 'pizza' --item
2 (Item pizza)
3 (Item Mariagrazia Pizza)
4 (Item Pizza Hut)
```

The `--search=wikidata-wapi` argument specifies the search plugin to be used, “pizza” is the search string, and `--item` indicates that we want to search for items. The top three items found by the search are shown on lines 2–4.

In addition to Wikidata, KIF comes with built-in plugins to search for entities in the namespaces of DBpedia and PubChem.

5.3. Wikidata in the Era of Generative AI Models

The rapid development of LLMs has changed the way users interact with information systems. Instead of formulating queries in a formal language or navigating predefined interfaces, users can express their information needs in natural language and expect AI systems to interpret their requests, retrieve relevant information, and generate natural language answers. This shift in interaction mode has renewed interest in KGs. Their structured and explicitly represented knowledge can complement the information encoded in LLMs, providing external sources of factual knowledge that can be retrieved and inspected. At the same time, the ability of LLMs to interpret and generate natural language creates new opportunities for enriching, curating, verifying, and maintaining KGs.

Wikidata is particularly well positioned in this ecosystem because it combines broad domain coverage with an openly accessible, collaboratively maintained, and continuously evolving representation of structured knowledge. Moreover, Wikidata can be accessed through multiple interfaces and representations, supporting different ways of querying and consuming its knowledge. This flexibility is particularly relevant for AI applications, where the requirements for accessing a KG may vary across systems and use cases.

Two complementary perspectives are useful for understanding the relationship

between Wikidata and AI. In *Wikidata for AI*, the Wikidata KG is used to contextualize and ground AI systems. In *AI for Wikidata*, AI models assist with knowledge completion, verification, and curation. Together, these perspectives can form a feedback loop in which Wikidata supports AI systems which in turn help improve Wikidata.

5.3.1. Wikidata for AI: Why KGs Matter for Generative AI?

LLMs are trained on vast quantities of unlabeled data, rich in world knowledge [Youssef et al. 2023]. However, this knowledge is embedded in the model’s parameters, making it difficult to access it through deterministic queries [Li et al. 2025b]. More importantly, a model may generate statements that are linguistically convincing but factually incorrect. This phenomenon, commonly referred to as hallucination, illustrates a fundamental limitation of relying solely on the model’s internal knowledge.

RAG Retrieval-Augmented Generation (RAG) addresses part of the aforementioned limitations by complementing the model’s parametric knowledge with information obtained from an external source [Lewis et al. 2020]. Rather than requiring the model to answer entirely from what was learned during training, a retrieval component first identifies potentially relevant evidence and provides it as context for generation. The model then use this retrieved information as context when formulating its response.

KGs serve as an interesting source of information for this purpose. While a conventional document-oriented RAG system retrieves passages of text, a KG exposes knowledge as entities connected by explicit relationships. This difference becomes important when answering questions that require reasoning. For instance, consider the following question:

Which researchers work on the Semantic Web and are affiliated with universities?

A language model may be able to produce a plausible list of researchers from its internal knowledge, but there is no guarantee that the listed people are correctly identified, the relationships hold, and the resulting set is complete. However, answering this reliably is not simply a matter of retrieving relevant passages from documents. It requires identifying people who are researchers working on the Semantic Web, determining their affiliations, and connecting these facts. A KG makes these connections explicit and allows them to be traversed and evaluated through a formal query (see Figure 5.9).

Wikidata is particularly useful in this setting not because it is complete or error-free, but because it makes knowledge available in a structured and inspectable form. As discussed in Section 5.2, its data model represents knowledge through items, properties, and statements, which can be further enriched with qualifiers and references. Qualifiers provide additional context about a statement, while references indicate the sources supporting it. For example, when answering the question above, a statement about a researcher’s affiliation can include qualifiers indicating the period during which the affiliation holds. This enables an AI system not only to retrieve the relevant information, but also to inspect the context and provenance associated with it. Moreover, because Wikidata is continuously maintained by a large community of contributors, changes to the underlying knowledge graph are reflected in subsequent queries, allowing AI systems to generate answers that

evolve as the knowledge itself is updated.

KGQA What we described above is the goal of a classic task in natural language processing, called Knowledge Graph Question Answering (KGQA). A KGQA system takes natural language questions as input and attempts to answer them by retrieving and reasoning over a knowledge graph, such as Wikidata. KGQA pipelines are usually employed in the implementation of chatbots, either as standalone applications or as components of domain-specific question-answering systems. One advantage of KGQA over other forms of QA is that the existence of an external representation of knowledge makes it possible to separate the process of interpreting natural language from the processes of obtaining relevant information. This separation is useful in cases where the output of the system is not necessarily a natural language answer.

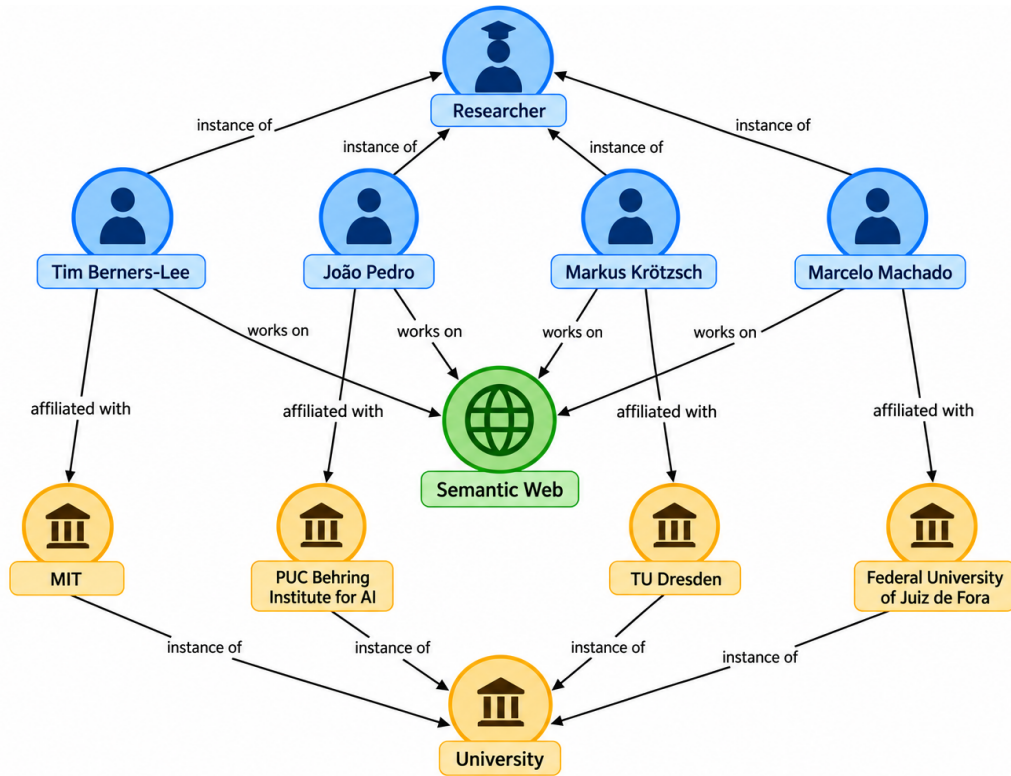


Figure 5.9: Knowledge graph representing researchers, their affiliations, and their relationship with the Semantic Web.

KIF-QA [Machado et al. 2025, Campos et al. 2025], for example, is a system that answers simple one-hop questions by retrieving statements from Wikidata (and other KGs). Rather than relying on the language model to provide the factual answer itself, the system uses it to interpret the natural language question and construct a structured query, while the KG provides the factual knowledge required to answer it. This illustrates a broader class of applications in which LLMs serve as an interface to structured knowledge, with the KG remaining responsible for providing the underlying facts. The pipeline of KIF-QA consists of five main steps (see Figure 5.10).

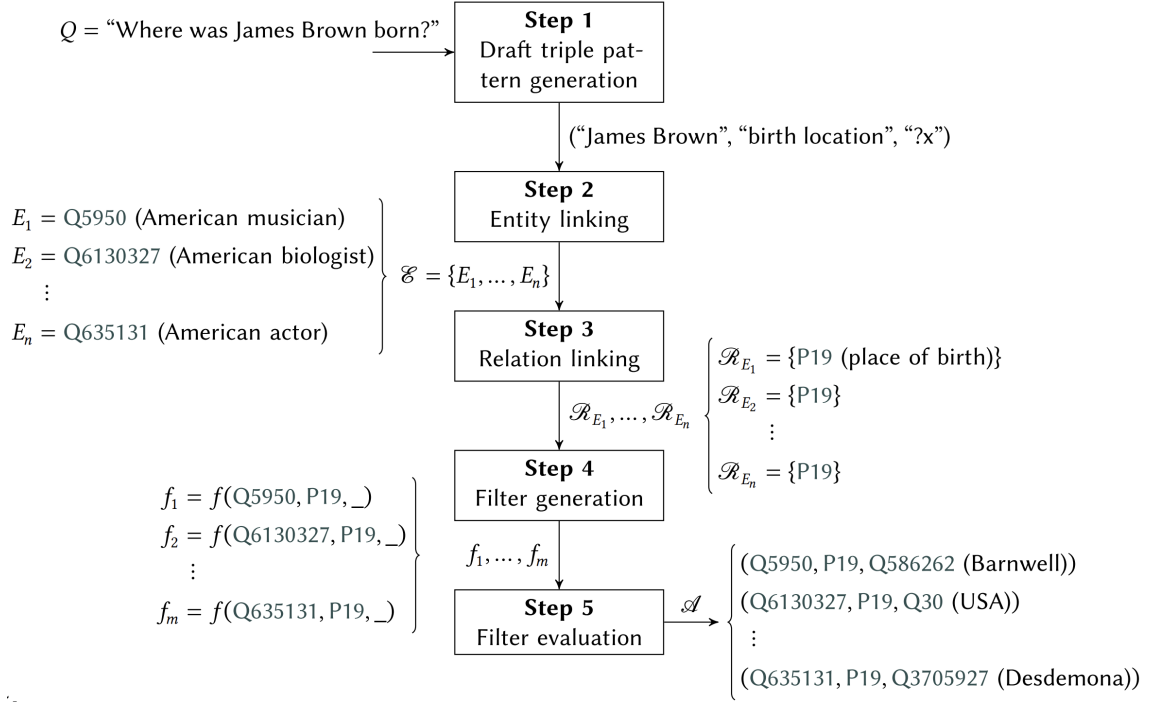


Figure 5.10: A typical KQGA pipeline for transforming natural language questions into structured queries over a knowledge graph [Machado et al. 2025].

We can summarize each of the steps of the pipeline of Figure 5.10 as follows:

- *Step 1: Draft triple pattern generation.* Given a simple question Q in natural language, KIF-QA prompts the LLM to generate a draft triple pattern (s, p, o) using text labels for the known components (s - p or p - o) and the variable “?x” for the unknown component (either s or o). When available, examples of “(question, draft triple pattern)” pairs are also provided in the prompt.
- *Step 2: Entity linking.* Given a draft triple pattern (s, p, o) , KIF-QA performs entity linking on its *topic entity label*, i.e., the known s or o . The system searches for entities with a similar label or alias in the namespace of the target graph. The result is a set of candidate entities accompanied by their descriptions. The LLM then *disambiguates* the topic entity label against the candidate entities, i.e., it selects among the candidates those entities whose label and description best match the topic entity label and the text of the question. The result of this step is a refined set $\mathcal{E}$ of *candidate topic entities*.
- *Step 3: Relation linking.* To resolve the predicate component p of the draft triple pattern (s, p, o) , KIF-QA proceeds as follows. For each candidate topic entity E in $\mathcal{E}$, the system obtains candidate relations R_i and R_j such that, for some x , $R_i(E, x)$ or $R_j(x, E)$ holds in the graph. The LLM then *disambiguates* the predicate label p against the label and description of each candidate relation R_k , the label and description of E , and the text of question Q . The result of this step is, for each topic entity E , a refined set $\mathcal{R}_E$ of *candidate relations for E*. This set is constructed such

that, for each R_k in $\mathcal{R}_E$, it is possible to determine whether E occurs as the source or target of R_k in the graph.

- *Steps 4 and 5: Filter generation and evaluation.* For each candidate topic entity E in $\mathcal{E}$ and candidate relation R_k in $\mathcal{R}_E$, KIF-QA generates a SPARQL query with a single triple pattern (filter) $f(E, R_k, _)$ if E occurs as the source of R_k , or a filter $f(_, R_k, E)$ if E occurs as the target. The system then executes all filters generated for the candidate entities in parallel. The resulting statement set $\mathcal{A}$ constitutes the *answer set* associated with the original natural language question Q .

Fact checking A similar pipeline can be employed in fact checking applications. For example, Wikidata has been used to verify facts generated by LLMs in a geospatial question-answering pipeline [Norouzi and Hitzler 2025], distinguishing generated facts for which supporting information can be retrieved from Wikidata from those for which no corresponding information is found. More generally, rather than determining whether a claim is true or false in general, such a system can check whether the KG contains a statement supporting the claim. The LLM interprets the natural language claim and, together with the entity linking process described above, translates it into a formal SPARQL query. In particular, an ASK query can be used to determine whether a matching statement exists in the KG, returning a Boolean value indicating whether the queried fact is represented. For example, given the claim “Tim Berners-Lee is an employee of the World Wide Web Consortium”, the fact-checking pipeline could generate the following SPARQL query targeting Wikidata:

```
ASK {
  # ("Tim Berners-Lee", "employer", "World Wide Web Consortium")
  wd:Q80 wdt:P108 wd:Q37033 .
}
```

This particular query evaluates to true, indicating that Wikidata contains a statement supporting the fact.

In addition to query-level checks, verification can also be performed against Wikidata’s ontology constraints. Chepurova et al. [Chepurova et al. 2024] propose an LLM-based knowledge extraction pipeline that uses Wikidata’s property constraints to verify extracted triples. For example, consider the triple:

⟨Tahiti Honey, screenwriter, Frederick Kohner⟩.

After linking its entities to Wikidata, the approach checks whether the types of the subject and object are compatible with the constraints defined for the screenwriter (P58) property. Such constraints specify which types of entities are expected in each position of a triple. For example, a property such as screenwriter (P58) could require its subject to be an entity representing a film or audiovisual work and its object to be an entity representing a person. The types of the linked items are determined from their instance of (P31) and subclass of (P279) hierarchies, allowing the system to assess whether the types of Tahiti Honey (Q20949975) and Frederick Kohner (Q2656718) satisfy the corresponding subject and object constraints. In this case, Wikidata is used not only as a source of factual knowledge, but also as a semantic schema for validating the knowledge produced by LLMs.

Summary Taken together, the previous use cases illustrate a broader pattern in the integration of Wikidata and language models. Wikidata can serve not only as a source for retrieving factual information, but also as a structured layer for grounding, verifying, constraining, and supporting the generation of knowledge-based responses. At the same time, the different approaches demonstrate that LLMs can interact with such structured knowledge at several layers, from interpreting natural language questions to generating queries and validating extracted or generated information.

5.3.2. AI for Wikidata: How Generative AI can support KGs?

In Section 5.3.1, we discussed how Wikidata can be used as an external source of structured knowledge to support AI systems. In this section, we present a complementary perspective. Here LLMs are no longer used just to interpret a query and retrieve existing knowledge. Instead, they become a collaborative component in the process of producing, enriching, and validating structured knowledge that is fed back to the KG itself.

5.3.2.1. Knowledge Graph Completion

KGs usually operate under a open-world assumption. This means that any missing relationships or facts are treated as unknown rather than false—a semantic choice that makes KGs are inherently incomplete [Ji et al. 2021]. Wikidata is no different: by its nature it is incomplete, even more so because it attempts to model the real world, which is dynamic, vast, and continuously evolving. As knowledge evolves, so does the need for automated approaches to identify and complete missing knowledge in KGs.

Knowledge graph completion generally refers to the task of predicting missing entities or relationships in a KG [Chen et al. 2020]. Early KG completion approaches relied primarily on the structural information encoded in the graph, including embedding- and path-based methods [Li et al. 2025a]. Subsequent approaches incorporated additional sources of information, such as textual descriptions and neural representations. More recently, LLM-based approaches have reformulated KG completion as a language model task, leveraging the semantic and linguistic capabilities of LLMs [Ibrahim et al. 2024].

An example system implementing KG completion pipeline is LLM Store [Machado et al. 2024b, Machado et al. 2024a]. Although not primarily designed for this purpose, LLM Store illustrates how the completion task can be preformed over Wikidata using an LLM as a knowledge source. LLM Store was developed as a KIF plugin (Section 5.2.4.5). It takes a structured query in a SPARQL-like syntax, translates it into a natural-language prompt and sends it to an LLM. The LLM answer is then translated back into structured Wikidata-like statements. Interestingly, LLM Store may be thought as a counterpart to KIF-QA [Machado et al. 2025, Campos et al. 2025]: while KIF-QA starts from a natural-language question and uses an LLM to construct a structured query whose answer is retrieved from the KG, LLM Store starts from a structured query pattern and uses an LLM to generate candidate facts that are translated into Wikidata statements.

For example, consider the query about Ada Lovelace’s (Q7259) occupation (P106):

```
SELECT ?o WHERE {  
  wd:Q7259 wdt:P106 ?o .  
}
```

LLM Store translates it into a prompt which is then fed into an LLM. In this case, the LLM is asked to predict the missing value (?o) for the occupation of Ada Lovelace. Here is the actual prompt sent to the LLM by LLM Store:

```
# SYSTEM
You are a helpful and honest assistant that resolves a TASK. Please, respond
→ concisely, with no further explanation, and truthfully.

# USER
TASK: Fill in the gap to complete the relation:
Ada Lovelace occupation ____

The output should be only a list containing the answers, such as `["answer_1",
→ "answer_2",..., "answer_n"]`. Do not provide any further explanation and
→ avoid false answers. Return an empty list, such as `[]`, if no information is
→ available.
```

The LLM may predict “mathematician” as one of the missing values. However, for this prediction to become structured knowledge, the generated string must be linked to the corresponding Wikidata item. This is precisely the entity linking step used in KIF-QA (see Figure 5.10). In this case, the missing value is resolved to Q170790 (mathematician) and the resulting statement corresponds to the triple:

$$\langle \text{wd:Q7259}, \text{wdt:P106}, \text{wd:Q170790} \rangle.$$

LLM Store is capable of handling more complex queries involving constraints on the subject or the object. For example, consider the query below which asks for the entity representing the “official sign language of Brazil”:

```
SELECT ?o WHERE {
  # ("Brazil", "official language", ?o)
  wd:Q155 wdt:P37 ?o .
  # (?o "instance of", sign language)
  ?o wdt:P31 wd:Q34228
}
```

This query gives rise to the following user prompt:

```
...
# USER
TASK: Fill in the gap to complete the relation:
Brazil official language ____ where ____ is an instance of sign language
...
```

For Wikidata-like KGs, LLM Store leverages the property constraints to improve the process of linking items and constructing the resulting statements. For instance, the property date of birth (P569) is used to link an item (usually a person) to a date value (its date of birth). So, if during completion for the value of this property, an LLM generates something other than a date value, LLM Store can detect the inconsistency and trigger a new generation round or report the error. Also, because the expected value is a literal

(date value), when an actual date value is generated, LLM Store knows it can skip the item-linking process entirely.

The KG completion approach we just discussed can be naturally complemented with RAG (see Section 5.3.1) to increase the likelihood of the LLMs generating sound answers. In fact, LLM Store offers a native solution for RAG which uses a plugin mechanism to connect to external sources at query time. Custom plugins can be developed by users, but LLM Store comes with a set of predefined plugins targeting popular sources, such as Wikipedia.

We conclude this section on knowledge graph completion by remarking that one of the main limitations of existing approaches, including LLM Store’s, is that they largely focus on completing only the basic components of a statement, namely, its subject, property, or object. More comprehensive approaches could be developed to generate and validate other components, such as qualifiers and references.

5.3.2.2. Knowledge Graph Construction

“Completion” is only one way in which LLMs can contribute to KGs. LLMs can also be used to extract information from unstructured sources in bulk and represent it as a graph, according to the schema (data model and vocabulary) of a target KG. In this setting, the task is not to infer a missing component of an existing statement in a KG, but rather to construct a graph from scratch, by extracting a set of statements from unstructured sources.

The implementation of KG construction pipeline typically involves three steps: (i) extracting the information from the sources, (ii) grounding the extracted entities and relationships in the target KG’s schema, and (iii) validating the resulting statements before possibly incorporating them back into the target KG [Zhong et al. 2023]. We detail these steps and how LLMs can assist with each of them next (see Figure 5.11).

Extraction The extraction step is usually performed through open information extraction (oIE) in which the LLM is instructed to return a list of *spo* triples from the input text. Usually no predefined schema used at this point, that is, the admissible relations and entity types are unconstrained. The instruction to the LLM is typically accompanied by examples, so that the model returns triples in a consistent format rather than in free-form prose. For example, consider the instruction below, we will use this as the running example throughout this section.

SYSTEM

Your task is to extract triples of the form `[Subject, Relation, Object]` from a ↪ given piece of text.

EXAMPLE

[examples go here]

USER

Extract triples from the following text:

Albert Einstein was born in Ulm on March 14, 1879.

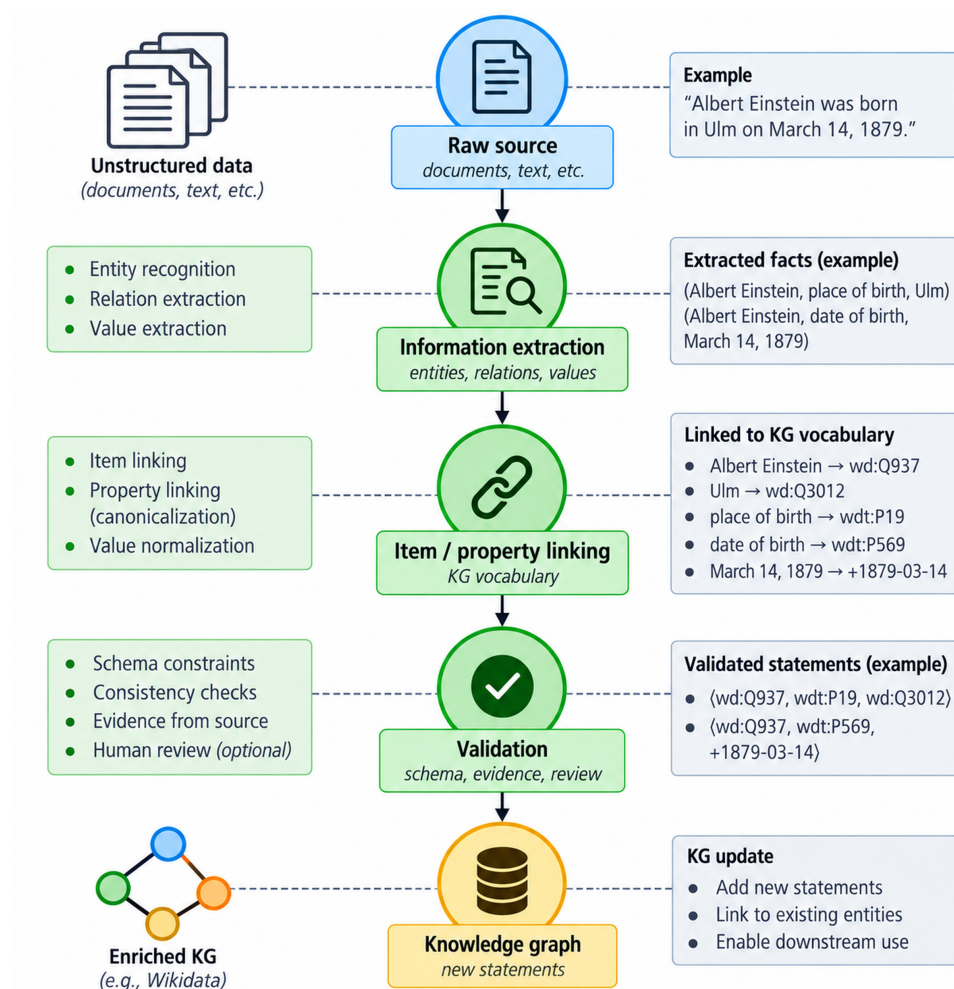


Figure 5.11: LLM-assisted pipeline for constructing KGs from unstructured data.

In this case, the model might respond with the triples:

⟨Albert Einstein, born in, Ulm⟩,
⟨Albert Einstein, born on, March 14, 1879⟩.

The EDC (Extract-Define-Canonicalize) [Zhang and Soh 2024] is a proposal that instantiates the design we just described. In its first phase, EDC prompts an LLM to extract draft triples freely, independently of any target schema. The advantage of schema-free extraction is that it allows for both matching the results against a preexisting target schema or creating a new schema on-the-fly. The disadvantage is that the resulting triple is still a triple of surface forms, that is, “born in” rather than P19, “Albert Einstein” rather than Q937, etc. This means that different fragments of the input text describing the a fact may yield different surface forms for the same entities. Also, a prompt that asks only for a core triple components might discard important contextual information (the kind of thing that goes into qualifiers and references).

Grounding The grounding step is responsible for transforming surface-form triples into KG statements. This transformation usually involves two related tasks: entity linking and relation linking. Entity linking attempts to map the surface forms in subject and object positions to existing KG entities, while relation linking (or canonicalization) attempts to map surface forms in predicate position to properties in the target KG.

So, for example, in a KB construction pipeline targeting Wikidata, the surface form “Albert Einstein” can be linked to Q937, “Ulm” to Q3012, “born in” to P19 (place of birth), and “born on” to P569 (date of birth). The resulting statements of our running example can thus be represented as:

```
⟨wd:Q937, wdt:P19, wd:Q3012⟩,  
⟨wd:Q937, wdt:P569, "+1879-03-14T00:00:00Z"⟩.
```

The approach adopted in the extraction stage of a KB construction pipeline often determines how difficult the grounding stage will be. At one extreme, closed information extraction (cIE) assumes a fixed KG schema and generates triples that conform to this schema. At the other extreme, oIE (the setting we have been describing) imposes no constraints and attempts build the graph from scratch. Each of these extremes fail in a characteristic way. Closed extraction is precise but brittle: it is hard to adapt it to a new domain and it tends to silently discard whatever the schema cannot express. Open extraction is flexible but produces graphs whose nodes and edges are surface forms: as the graph grows, synonymy, coreference, and predicate variation accumulate and erode the properties that motivated building a KG in the first place (precision, interpretability, and logical consistency) [Chepurova et al. 2026].

A compromise between the two extremes, cIE and oIE, is what is sometimes called ontology-guided extraction. In ontology-guided extraction, one extracts freely, as in oIE, and only then align the surface forms against a given schema. This separation of concerns motivates the treatment of grounding as an independent, first-class stage, rather simply post-processing. And it is also why a vocabulary such as Wikidata’s remains useful even for systems that never intend to contribute statements to Wikidata itself. Table 5.1 summarizes the three approaches we just described.

Setting	Schema	Typical failure mode
Closed IE (cIE)	Fixed, given a priori	Low recall; costly domain adaptation
Open IE (oIE)	None; induced from text	Redundancy; inconsistent surface forms
Ontology-guided	External, applied after extraction	Mismatch between text and ontology

Table 5.1: Positions along the information extraction spectrum for KG construction.

The previously mentioned EDC (Extract-Define-Canonicalize) [Zhang and Soh 2024] proposal supports a mixture of three approaches, cIE, oIE, and ontology-guided. If one provides a target schema, EDC becomes ontology-guided, as it aligns the open triples against the schema after extraction. But if no target schema is provided, EDC builds and canonicalizes its own schema on-the-fly. In either case, the pipeline is the same. EDC first uses an LLM to generate a natural-language definition for each relation induced by

the extraction phase. These definitions are then used to (i) identify semantically related elements in the schema, and (ii) validate whether an extracted relation can be replaced by one of the candidate relations.

For example, given the sentence “Alan Shepard was a member of the Apollo 14 crew”, EDC extracts the surface-form triple

⟨Alan Shepard, participated in, Apollo 14⟩.

The relation “participated in” in this case is assumed to have its usual meaning (definition), namely, that the subject took part in the event specified by the object entity. During canonicalization, EDC compares this definition to those of the relations in the target schema and identifies, say, “mission”, as a suitable candidate. EDC then uses the LLM to verify whether the replacement of “participated in” by “mission” is appropriate in the given context, and if so, and produces the canonical triple

⟨Alan Shepard, mission, Apollo 14⟩.

This example illustrates that, rather than requiring the full target schema during extraction, EDC performs the alignment after the initial extraction, allowing the same pipeline to be used with both a target scheme or an initially empty schema which is generated on the fly.

Grounding natively against Wikidata Wikontic [Chepurova et al. 2026] is an interesting proposal for grounding surface-forms against Wikidata. Instead of treating Wikidata as one of many possible target schemas, it uses Wikidata as the backbone of the grounding method. More specifically, Wikontic uses a custom ontology derived from Wikidata which contains 2,464 factual properties such that, for each of these, Wikontic records the subject and object type constraints with entity types expanded recursively via instance of (P31) and subclass of (P279) relationships, so that a constraint declared for a superclass, such as “audiovisual work”, still applies to its subclasses, such as “film”.

The custom Wikidata-derived ontology is used by Wikontic to guide both extraction and alignment, which are carried out more thoroughly than in the generic proposals discussed previously, such as EDC. In particular, Wikontic’s extraction stage does not stop at the bare triple: it uses the LLM to generate statement qualifiers (time, location, or other conditions under which the statement holds). So, given the sentence “In 2010, Christopher Nolan directed the science fiction movie Inception”, Wikontic might generate as a candidate the surface-form triple & qualifier:

⟨Nolan, directed, Inception⟩
qualifier: ⟨point in time, 2010⟩

with the additional constraints that “Nolan” is an instance of “human” and “Inception” is an instance of “film”.

In Wikontic, entity and relation linking are driven by type constraints rather than by semantic similarity definitions alone. Candidate types for both subject and object are retrieved from a dense index and selected by the LLM, with supertypes added so that constraints declared at a higher level of abstraction still apply. Wikidata’s own constraints

are then used to enumerate every relation that can possibly connect the chosen types, with inverses (such as “directed” vs “director”) also considered. Only the admissible set together with the source text are submitted to the LLM for a final decision. So, in the previous example, Wikontic’s approach repairs both the surface form and the direction of the candidate, yielding

⟨Inception, director, Christopher Nolan⟩
qualifier: ⟨point in time, 2010⟩

Differently from EDC, Wikontic handles entity deduplication. Subject and object mentions are linked to existing entries construction that share a compatible type in the current graph under. This is done using embeddings over previously seen aliases. If a match is found, the mention is replaced by the canonical label and its surface form is stored as a new alias; otherwise a new entity is created. This is what makes the merging of “Nolan” and “Christopher Nolan” possible in the example above.

Note that although we said that Wikontic is an example of an ontology-guided system, the “entities” it produces are not Wikidata items: instead they are canonical names together with aliases, and the final entity-linking step is delegated to users. Also, although Wikontic generates qualifiers it does not attempt to generate references or ranks.

Validation The grounding stage produces well-formed statements, but not necessarily correct statements: the LLM may link a surface-form to the wrong entity, select a property that only approximates the relation in the source, or generate a value the source does not support. Candidate statements should therefore be validated before being added to the target graph [Zhong et al. 2023, Chen et al. 2024].

Validation draws on at least four complementary sources of evidence. The first one are *schema constraints*, such as domain and ranges of properties. For instance, P19 (place of birth) expects a human as subject and a geographic place as object. These constraints have been formalized in SHACL and SPARQL [Ferranti et al. 2024] and used to verify LLM-extracted triples [Chepurova et al. 2024]. The second source of evidence for statement validation is the *existing knowledge* in the graph. For instance, a generated statement might contradict a statement that already exists in the graph, which serves as evidence to deem the generated statement invalid. The third source of evidence is the *source text* which was used to produce the statement as, in principle, the former should entail the latter. Finally, the fourth source of validation is *human review*, which is costly but might be necessary in mission-critical applications.

Note that statements that fail validation need not be discarded. They can be flagged as such and recorded together with the constraint violated and the input text that was used to generate them.

5.3.2.3. Verifying and Curating Wikidata with LLMs

So far, we discussed knowledge graph completion and construction. A complementary line of work uses LLMs not to add new knowledge, but to assess and improve knowledge that is already present in Wikidata. Quality assurance at Wikidata’s scale is not a new problem. For over a decade, the graph has relied on machine-learning classifiers trained

on edit metadata and content features to catch vandalism and low-quality edits in near real time [Heindorf et al. 2016]. What LLMs add to this picture is not merely a faster classifier, but a qualitatively different kind of check. They can be fed a statement alongside a piece of text and asked to decide on whether the text supports the statement. Alternatively, they can be asked to evaluate a statement against the ontology in order to determine if the combination is sound. These checks depend on language and semantic understanding rather than edit-history features alone.

Verifying statements against external evidence One way to check a statement is to ask whether the world, as described in some independent source, actually supports it. Adam and Kliegr [Adam and Kliegr 2025] propose a method in which an LLM does not rely on its own internal knowledge to verify a statement. Instead, the statement is compared against externally retrieved grounding text, allowing the system to produce a traceable justification for its assessment rather than an unexplained verdict. This distinction matters. An LLM asked in isolation whether “Tim Berners-Lee is an employee of the World Wide Web Consortium” is true answers from the same parametric knowledge that makes LLMs unreliable narrators of fact in the first place (Section 5.3.1). Conversely, an LLM asked to compare the statement against a retrieved passage performs a task closer to reading comprehension, where a wrong verdict can at least be traced back to a specific piece of evidence. The approach is demonstrated on Wikidata by automatically identifying statements that require verification, prioritizing the pool of candidates a human curator would otherwise have to review from scratch.

Verifying statements against the ontology A second, complementary check does not need an external document at all. It asks whether a statement is internally consistent with the ontology it is supposed to belong to. Wikidata’s property constraints specify expectations about how a property may be used, such as the subject and object type constraints introduced in Section 5.3.1. A violation of these constraints is a structural signal that a statement may be wrong, independently of whether it is factually true. Formalizing these constraints does not by itself require an LLM. Ferranti et al. [Ferranti et al. 2024] encode Wikidata’s property constraints in SHACL and SPARQL, providing a systematic, fully symbolic way to identify violations across the graph. Nevertheless, this formalization is what makes LLM-based curation checkable rather than merely plausible-sounding. It supplies explicit, machine-checkable criteria against which a statement can be evaluated, whether it is authored by a human editor or proposed by an LLM-based construction pipeline (Section 5.3.2.2).

The two checks are not alternatives so much as complementary signals that recent systems combine directly. MAKGED [Li et al. 2025c], for instance, detects erroneous triples by having several LLM-backed agents examine a candidate from different angles. Each agent is conditioned on a different bidirectional view of the triple’s local subgraph structure, settling disagreements through a multi-round discussion rather than relying on a single opaque score. This is a step beyond earlier, purely structural error detectors. Those models score a triple’s plausibility from graph embeddings alone and struggle precisely on cases that look structurally normal but are semantically wrong [Liu et al. 2024]. Replacing a director with another plausible-looking director is exactly the kind of error that structure alone misses but a language-aware check catches. Combining a structural signal with an

LLM’s reading of an entity’s textual description, as MAKGED does, is one way to close that gap. Comparing a statement against retrieved external text, as Adam and Kliegr do, is another. Both methods replace a single confidence score with something a curator can inspect.

Curating structure, not just statements LLMs can also be used to address problems in the semantic structure of the KG itself, rather than in any one statement. WiKC [Peng et al. 2024] combines LLMs with graph-mining techniques to refine the Wikidata taxonomy. The approach identifies problematic taxonomic relations and performs operations such as cutting spurious links or merging classes that were only superficially distinct, producing a cleaned version of the taxonomy. This extends the role of LLMs beyond the statement-level completion and verification discussed in the previous sections. It supports the maintenance of the class hierarchy that those very statements are typed against. This hierarchy forms the *instance of* and *subclass of* backbone that both KG completion (Section 5.3.2.1) and KG construction (Section 5.3.2.2) rely on to generalize constraints across levels of abstraction. An error in the taxonomy propagates silently into every statement typed under it, making this structural layer a target worth curating in its own right.

From proposal to insertion: what automation cannot yet close These approaches also highlight an important distinction between AI-assisted curation and fully automated editing. Because LLMs can generate plausible but incorrect statements, their output cannot be assumed to be suitable for direct insertion into Wikidata. The very hallucination risk that motivates using Wikidata to ground LLM outputs elsewhere in this chapter reappears on the writing side, once an LLM is itself proposing what the graph should say [Lavrinovics et al. 2025]. [Regino and dos Reis 2025] offer a direct test of whether an LLM can be trusted to validate a triple before it is inserted, rather than just completing or extracting one. They evaluate LLMs on four concrete curation tasks: class and property alignment, URI standardization, semantic consistency, and syntactic correctness. Their results are informative less for a particular accuracy number than for where the models struggle. They highlight challenges with domain generalization across ontologies, semantic drift over longer chains of reasoning, and cases that still require a human in the loop even when the model performs well on individual tasks. Consequently, the more defensible role for LLMs in this setting is as assistants that propose candidate statements, identify likely errors, retrieve supporting evidence, or prioritize items for human review. This aligns exactly with the role played by the statement-level and structural checks described above. Wikidata’s constraints, references, and collaborative editing mechanisms provide complementary safeguards for evaluating any AI-generated contribution before it becomes part of the graph.

Taken together, these approaches illustrate the complementary nature of the two perspectives introduced above. In Wikidata for AI, the LLM acts primarily as an interface through which structured knowledge can be interpreted, queried, and incorporated into an AI application. In AI for Wikidata, the LLM becomes a component in the construction and maintenance of the knowledge itself, generating, extracting, enriching, or validating statements. The interaction is therefore not a one-way flow of information from the KG to the model, but a feedback loop in which structured knowledge can support AI systems while AI can assist in expanding and improving the KG.

5.3.3. Wikidata Infrastructure for AI Applications

The interaction between Wikidata and AI applications depends not only on techniques, but also on infrastructure that makes the knowledge graph accessible to modern AI systems. While conventional interfaces provide precise mechanisms for querying and retrieving Wikidata, applications based on LLMs and AI agents often benefit from semantic retrieval and more flexible forms of interaction.

In this section, we introduce two community-driven initiatives that address these requirements: the Wikidata Vector Database and the Wikidata Model Context Protocol. They provide complementary capabilities for semantic retrieval and agentic interaction, extending the traditional access mechanisms discussed in Section 5.2.4. Together, they illustrate how the Wikidata ecosystem is evolving to accommodate the needs of generative AI applications.

5.3.3.1. The Wikidata Vector Database

In both directions of collaboration between Wikidata and AI, the applications mentioned in previous sections have demonstrated the need to query Wikidata to capture entities present in the graph. While traditional mechanisms generally serve this purpose, they are often limited to structured searches or exact keyword matching; this hinders interaction with applications that primarily handle natural language, where terms may differ grammatically yet share the same meaning. To bridge this gap between traditional structured search and continuous semantic spaces, the *Wikidata Vector Database* (Wikidata VDB)²⁰ operationalizes semantic access to Wikidata by storing and serving dense vector representations of items and properties [Wikimedia Deutschland 2026].

Wikidata VDB is developed as the core operational layer of the *Wikidata Embedding Project*, which is an initiative led by Wikimedia Deutschland in collaboration with Jina AI²¹ and DataStax²² intended to support AI and machine-learning applications [Wikimedia Deutschland 2025]. While the traditional Wikidata keyword search, via CirrusSearch²³, is strictly bound to surface text and SPARQL is precise and accurate, but its steep learning curve and complexity make it challenging for many users to leverage, vector representations allow semantic search even when queries differ syntactically or orthographically from canonical labels.

The basic idea is simple: information associated with Wikidata entities is transformed into textual representations and then encoded as dense vectors. Queries can be embedded in the same space, allowing the system to retrieve semantically related entities even when the wording of the query does not exactly match the labels stored in Wikidata. For example, a user might search for “scientists who investigated the relationship between heredity and evolution” without knowing the name of the relevant person or the precise vocabulary used in Wikidata. Semantic retrieval can identify entities whose textual representations are conceptually close to the query.

²⁰<https://wd-vectordb.wmcloud.org>

²¹<https://jina.ai/>

²²<https://www.ibm.com/products/datastax>

²³<https://www.mediawiki.org/wiki/Help:CirrusSearch>

KIF-QA, for example, uses the traditional Wikidata API to retrieve candidate entities during the entity and relation linking processes. As the authors noted, KIF-QA may fail to answer correctly if the LLM generates a surface form that, does not yield any can–

To search for Wikidata items by their title on a given site, use [Special:ItemByTitle](#).

Advanced search:

Search in:

There were no results matching the query. You may [create a new item](#) for "freakstyle".

(a) Searching the Wikidata traditional lookup interface.

 **Wikidata Vector Database**

Language: ☒ ALL ☐ AR ☐ DE ☐ EN ☐ FR Search type:

Scope:

Rerank:

 This tool is in early testing, and results may be incomplete or inaccurate. Your queries are sent to a third-party service (JinaAI) for processing, and we store them for up to 90 days for quality improvements. We welcome your feedback! Please help us improve by filling out [our survey](#).

Freakstyle (Q4354676)

2002 video game

Similarity Score: 0.8071854

Source: Vector Search

(b) Searchin the Wikidata Vector Database interface.

Figure 5.12: Comparison of Wikidata interfaces for the term “freakstyle” (with typo). The traditional lookup interface returns no results (a), whereas the Vector Database interface retrieves relevant results (b).

didate entities in Wikidata. For instance, the paper reports that Llama 3.3 misrecognized “freakstyle” as “freakstyle” (with ‘a’) in the question “who published the game freakstyle” (from SimpleQuestionsQA dataset [Bordes et al. 2015]). Consequently, a keyword-based search cannot retrieve the correct entity from Wikidata (see Figure 5.12a). However, as shown in Figure 5.12b, semantic search through the Wikidata VDB can retrieve the correct entity despite the typo as dense embeddings capture conceptual context beyond brittle surface orthography.

5.3.4. Data Serialization and Vector Indexing. Turning a symbolic entity into something an embedding model can consume requires a text representation of it, and the choice of representation matters as much as the choice of model. Each item’s label, description, aliases, and claims are concatenated into a short, human-readable paragraph, so that the text given to the embedding model reads like a compact entity summary rather than a bare list of triples. For complex entities with extensive claim networks, a statement-level chunking strategy splits claims into smaller segments while maintaining the core metadata (label, description, aliases, and *instance of* [P31]) across all chunks to fit model context windows. Furthermore, Wikidata properties (PIDs) are encoded using a dual representation: one treats the property itself as an entity, combining its own label, description, and aliases; the other pairs the property’s label and description with a handful of example statements that use it, so that a property such as P166 (award received) is represented alongside concrete instances of its use, giving the embedding model usage context that the property’s bare definition would not convey on its own. All text passages are encoded using *Jina Embedding V3* (`retrieval.passage` mode), a multilingual embedding model supporting more than 100 languages and input sequences of up to 8,192 tokens. The resulting embeddings are indexed in *DataStax Astra DB*, partitioned separately by entity type (items vs. properties) and language. For unsupported languages, input queries are automatically translated into English via the MinT machine translation engine (based on NLLB-200) before searching across language shards.

5.3.5. Hybrid Retrieval Pipeline. Rather than replacing lexical lookup, the database combines vector-based retrieval with traditional keyword search through a hybrid execution pipeline (Figure 5.13). When a query is received, vector candidate retrieval (via *Jina Embedding V3* `retrieval.query`) and keyword retrieval (via *CirrusSearch* with stop-word removal and logical OR expansion) run in parallel. The system can filter results dynamically using parameters such as `instanceof` or `exclude_external_ids`. Candidates from both channels are merged using Reciprocal Rank Fusion (RRF) [Cormack et al. 2009], computed as:

$$\text{RRF}(e) = \sum_{l \in L} \frac{1}{k + r_l(e)} \quad (1)$$

where L is the set of ranked result lists, $r_l(e)$ is the rank position of entity e in list l , and $k = 60$ is a smoothing constant. Optionally, candidates undergo secondary reranking using *Jina Reranker v2 Base Multilingual*, a cross-encoder model that scores query-entity alignment at the cost of slightly higher latency.

5.3.6. Accessing Wikidata VDB. The database is exposed through a small REST API. Three endpoints cover the two ways one might want to use it: searching for new candidates,

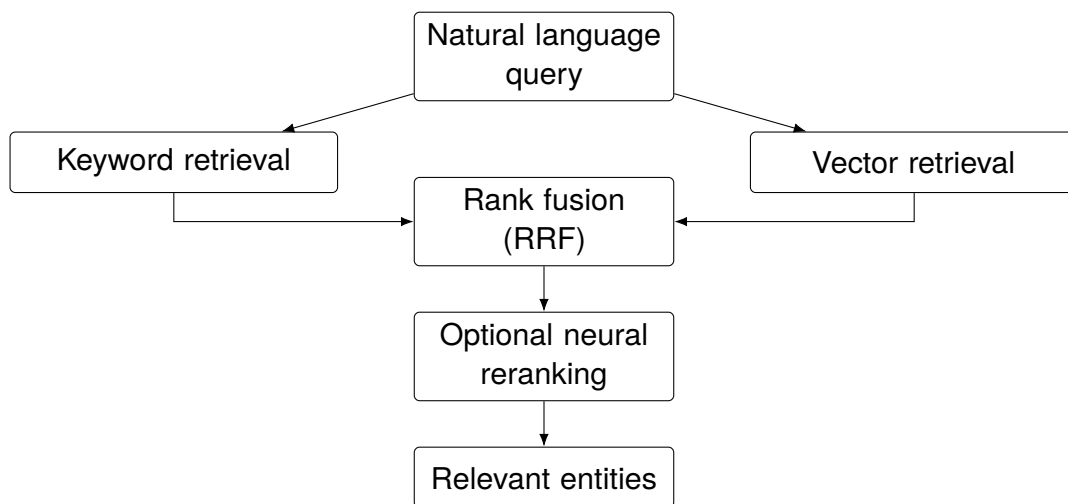


Figure 5.13: Hybrid pipeline to access Wikidata through lexical and semantic search.

and scoring a fixed list of candidates one already has.

- GET `/item/query/` — hybrid vector-and-keyword search over item shards, returning QIDs ranked by their fused score.
- GET `/property/query/` — the same hybrid search restricted to property shards, returning PIDs.
- GET `/similarity-score/` — given a text query and an explicit list of QIDs and/or PIDs, computes and returns similarity scores only for those entities, without retrieving any new candidates.

A request for items related to the earlier example query can be issued as:

<https://wd-vectorddb.wmcloud.org/item/query/?query=scientists+who+investigated+heredity+and+evolution&lang=en>

which returns a ranked list of QIDs, such as `wd:Q1035` (Charles Darwin) together with their vector-similarity (0.72215676). An `instanceof` parameter can restrict item search to instances of a given class, and property search accepts an `exclude_external_ids` flag to drop properties whose datatype is `external-id`, which are rarely useful for semantic matching.

5.3.7. Limitations. The project describes itself as an alpha release, and two limitations follow directly from that status. Coverage is bounded to roughly 30 million items, those connected to at least one Wikipedia article, which excludes the long tail of Wikidata items with no corresponding Wikipedia page. And unlike the weekly and incremental dumps discussed in Section 5.2.4.3, the embeddings are not refreshed on a regular cadence: at the time of writing they are built from a May 2026 Wikidata snapshot, so an application relying on the vector database alone may miss statements added or edited since then.²⁴

²⁴In a LinkedIn post Wikidata profile posted that coverage for more Wikidata items is being prioritized, targeting September 2026: <https://lnkd.in/dzG4NbHt>

The project’s own documentation is explicit about the intended use that follows from this:

The vector search is primarily designed for exploration and discovery. It helps users and researchers identify related entities that might not appear through traditional keyword search. The results are not intended to serve as direct answers but as starting points for further investigation and analysis. [Wikimedia Deutschland 2026]

5.3.7.1. Wikidata MCP: Wikidata as an Agentic Tool Environment

The transition from static Retrieval-Augmented Generation (RAG) to agentic AI paradigms introduces a critical requirement: Large Language Models (LLMs) need standardized, dynamic interfaces to query and explore external knowledge environments. While LLMs are generally capable of generating syntactically valid SPARQL queries, they frequently fail in zero-shot execution over Wikidata due to a lack of domain-specific schema knowledge, unfamiliarity with entity identifiers (QIDs) and property predicates (PIDs), and complex taxonomic hierarchies. To bridge this gap, the Wikimedia community developed the *Wikidata MCP* server [Wikidata Community 2026], an implementation of Anthropic’s open-standard Model Context Protocol (MCP) [Anthropic 2024]—frequently described as the “USB-C for AI” tools. By standardizing communication, MCP transforms Wikidata from a passive database accessed via hardcoded programmatic endpoints into an interactive *tool environment* that an LLM agent can actively explore.

Operationalizing agentic access via a standardized HTTP streaming interface (`streamable_http`) hosted at <https://wd-mcp.wmcloud.org/mcp/>, the service allows client environments to register the endpoint directly within their system configuration:

```
{
  "mcpServers": {
    "wikidata": {
      "type": "streamable_http",
      "url": "https://wd-mcp.wmcloud.org/mcp/"
    }
  }
}
```

Crucially, the interface operates in a read-only state, establishing a formal governance boundary that permits multi-turn graph exploration without exposing the underlying Knowledge Graph to unauthorized edits or operational risks. Within this secure execution context, the server exposes a specialized suite of tools designed to mirror a human researcher’s exploratory workflow rather than forcing immediate query synthesis:

- `search_items(query, lang)`: Executes hybrid (vector and lexical) retrieval over Wikidata items to discover candidate entity identifiers (QIDs) from natural language queries.
- `search_properties(query, lang)`: Performs hybrid retrieval over properties to identify relevant relation predicates (PIDs) prior to constructing structured queries.

- `get_statements(entity_id, include_external_ids, lang)`: Retrieves direct graph assertions (triplets) for an entity, deliberately excluding qualifiers and references to provide a fast, low-latency structural overview.
- `get_statement_values(entity_id, property_id, lang)`: Fetches full claim details for a specific entity-property pair, including statement qualifiers, provenance references, and deprecation statuses when fine-grained metadata is required.
- `get_instance_and_subclass_hierarchy(entity_id, max_depth, lang)`: Traverses class relationships via *instance of* (P31) and *subclass of* (P279) up to a configured depth (defaulting to 5), validating ontological boundaries before query generation.
- `execute_sparql(sparql, K)`: Executes the compiled SPARQL query against the Wikidata Query Service (WDQS), returning up to K result rows or diagnostic syntax error messages.

The true architectural value of this modular suite lies in facilitating an autonomous reasoning loop, replacing rigid single-step prompt-to-SPARQL generation with iterative context gathering. As illustrated in Figure 5.14, an agent processing a complex prompt such as “*Who are the presidents of France?*” does not guess identifiers; it first invokes `search_items` and `search_properties` to resolve France (Q142) and president (Q191954), verifies class boundaries using `get_instance_and_subclass_hierarchy`, and only then constructs and executes the SPARQL query via `execute_sparql`. This multi-stage interaction ensures ground-truth verification and eliminates hallucinations, as facts are anchored strictly to deterministic graph execution. If execution yields syntax errors or empty result sets, the feedback loop routes the error message back to the LLM for self-correction.

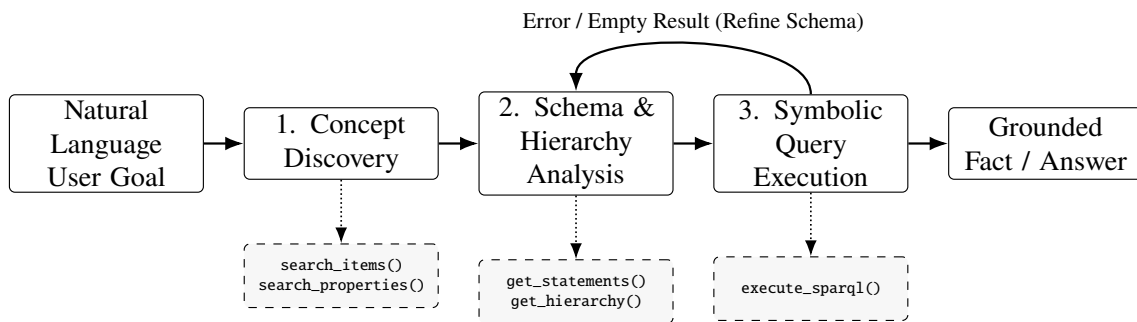


Figure 5.14: Agentic reasoning workflow enabled by Wikidata MCP: moving iteratively from fuzzy entity discovery to schema verification and deterministic SPARQL execution.

Beyond standalone research, the deployment of Wikidata MCP establishes a foundational bridge between structured open data and the broader developer ecosystem. The protocol has enabled integrations within agentic execution frameworks such as LangChain, as well as specialized workflow plugins like the OpenCode Wikidata SPARQL workflow (bringing direct Wikidata access to over 7.5 million developers). Furthermore, this paradigm

extends beyond Wikidata itself: through *Wikibase MCP* (wb-mcp.wmcloud.org), third-party Wikibase instances can register dedicated endpoints, extending standardized, agentic graph exploration to private enterprise knowledge bases.

5.4. Conclusion

This chapter has presented Wikidata as a KG and examined its role in the emerging ecosystem of generative AI. We first introduced the main concepts of the Wikidata data model and its RDF representation, showing how entities, statements, qualifiers, references, and ranks are represented as structured knowledge. We then discussed the different mechanisms through which this knowledge can be accessed, ranging from SPARQL and REST APIs to data dumps, Triple Pattern Fragments, and KIF. These interfaces expose the same underlying knowledge at different levels of abstraction and support different requirements in applications that consume or integrate Wikidata.

We then turned to the intersection between Wikidata and generative AI from two complementary perspectives. In *Wikidata for AI*, Wikidata can provide an external source of structured and inspectable knowledge that complements the parametric knowledge of LLMs. We discussed how this role appears in RAG, KGQA, and systems such as KIF-QA, where the language model is used to interpret a natural-language question while the KG provides the factual information. In these settings, Wikidata is not merely an additional text source: its explicit entities and relationships make it possible to retrieve, inspect, and query knowledge in ways that are difficult to guarantee from the model alone.

In the complementary *AI for Wikidata* perspective, the language model is instead used as a component in the construction and maintenance of the KG. We discussed approaches for KG completion and construction, where LLMs can predict missing information or extract candidate statements from unstructured sources, as well as approaches for verification and curation. Examples such as LLM Store illustrate how an LLM can be used to generate candidate knowledge, while other approaches use external evidence, ontology constraints, or graph structure to identify potentially incorrect statements. These examples also highlight an important limitation: the ability of LLMs to generate plausible information does not make their output reliable enough for unrestricted insertion into a collaboratively maintained KG. Human review, provenance, structural constraints, and other forms of validation therefore remain important parts of the process.

Finally, we examined recent infrastructure developed around Wikidata to support AI-oriented use cases. The Wikidata Vector Database extend traditional lexical retrieval with semantic retrieval, while the Wikidata MCP provides an interface through which AI agents can dynamically search, inspect, explore, and query the graph. These mechanisms do not replace Wikidata’s traditional interfaces. Rather, they provide additional layers through which AI systems can interact with the same underlying structured knowledge, combining semantic retrieval with precise graph-based querying when necessary (Table 5.2 summarizes the Wikidata access interfaces).

Taken together, these perspectives show that the relationship between Wikidata and generative AI is bidirectional. Wikidata can provide AI systems with structured, multilingual, inspectable, and provenance-aware knowledge, while AI models can assist with discovering, constructing, verifying, and maintaining knowledge in the graph. The

Access mechanism	Paradigm	Typical role in AI systems
Keyword search	Lexical retrieval	Exact entity names, labels, and known terminology
SPARQL	Symbolic querying	Precise filtering, joins, aggregation, and graph reasoning
Vector Database	Semantic retrieval	Semantic search, candidate generation, RAG, and graph exploration
Wikidata MCP	Agentic tool use	Dynamic search, graph exploration, query execution, and orchestration

Table 5.2: Complementary mechanisms for accessing Wikidata in AI applications.

goal, therefore, is not to replace one technology with the other, but to combine their complementary capabilities. At the same time, this integration must preserve the properties that make Wikidata valuable as a collaborative KG, particularly the explicit representation of knowledge, provenance, openness, and community-based curation.

Declaration on Generative AI

During the preparation of this work, the authors used generative AI tools for machine translation from Portuguese to English, grammar and spelling checking, and to assist in paraphrasing and improving the writing style. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

- [Adam and Kliegr 2025] Adam, D. and Kliegr, T. (2025). Traceable llm-based validation of statements in knowledge graphs. *Information Processing & Management*, 62(4):104128.
- [Angles 2012] Angles, R. (2012). A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177. IEEE.
- [Anthropic 2024] Anthropic (2024). Introducing the model context protocol. <https://www.anthropic.com/news/model-context-protocol>. Accessed: 2026-08-27.
- [Bordes et al. 2015] Bordes, A., Usunier, N., Chopra, S., and Weston, J. (2015). Large-scale simple question answering with memory networks. *arXiv preprint arXiv:1506.02075*.
- [Campos et al. 2025] Campos, J. P. P., Machado, M., Lima, G., and da Silva, V. T. (2025). Uma ferramenta web baseada em llms para respostas a perguntas simples sobre bases de conhecimento heterogêneas. In *Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia)*, pages 159–162. SBC.
- [Carothers and Prud’hommeaux 2014] Carothers, G. and Prud’hommeaux, E. (2014). RDF 1.1 Turtle. W3c recommendation, W3C. <http://www.w3.org/TR/2014/REC-turtle-20140225/>.

- [Chen et al. 2024] Chen, H., Shen, X., Lv, Q., Wang, J., Ni, X., and Ye, J. (2024). SAC-KG: Exploiting Large Language Models as Skilled Automatic Constructors for Domain Knowledge Graph. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4345–4360.
- [Chen et al. 2020] Chen, Z., Wang, Y., Zhao, B., Cheng, J., Zhao, X., and Duan, Z. (2020). Knowledge graph completion: A review. *IEEE Access*, 8:192435–192456.
- [Chepurova et al. 2026] Chepurova, A., Bulatov, A., Burtsev, M., and Kuratov, Y. (2026). Wikontic: Constructing wikidata-aligned, ontology-aware knowledge graphs with large language models. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8304–8319.
- [Chepurova et al. 2024] Chepurova, A., Kuratov, Y., Bulatov, A., and Burtsev, M. (2024). Prompt me one more time: A two-step knowledge extraction pipeline with ontology-based verification. In *Proceedings of TextGraphs-17: Graph-based Methods for Natural Language Processing*, pages 61–77.
- [Consortium 2025] Consortium, T. U. (2025). Uniprot: the universal protein knowledge-base in 2025. *Nucleic Acids Research*, 53(D1):D609–D617.
- [Cormack et al. 2009] Cormack, G. V., Clarke, C. L. A., and Büttcher, S. (2009). Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 758–759. ACM.
- [Ferranti et al. 2024] Ferranti, N., De Souza, J. F., Ahmetaj, S., and Polleres, A. (2024). Formalizing and validating Wikidata’s property constraints using SHACL and SPARQL. *Semantic Web*, 15(6):2333–2380.
- [Harris and Seaborne 2013] Harris, S. and Seaborne, A. (2013). SPARQL 1.1 query language. W3c recommendation, W3C. <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.
- [Hayes and Patel-Schneider 2014] Hayes, P. and Patel-Schneider, P. (2014). RDF 1.1 semantics. Recommendation, W3C. <http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/>.
- [Heindorf et al. 2016] Heindorf, S., Potthast, M., Stein, B., and Engels, G. (2016). Vandalism detection in Wikidata. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM)*, pages 327–336. ACM.
- [Hitzler et al. 2010] Hitzler, P., Krötzsch, M., and Rudolph, S. (2010). *Foundations of Semantic Web Technologies*. Chapman & Hall.
- [Ibrahim et al. 2024] Ibrahim, N., Aboulela, S., Ibrahim, A., and Kashef, R. (2024). A survey on augmenting knowledge graphs (KGs) with large language models (LLMs): models, evaluation metrics, benchmarks, and challenges. *Discover Artificial Intelligence*, 4(1):76.

- [Ji et al. 2021] Ji, S., Pan, S., Cambria, E., Marttinen, P., and Yu, P. S. (2021). A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE transactions on neural networks and learning systems*, 33(2):494–514.
- [Ji et al. 2023] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.
- [Kellogg et al. 2026] Kellogg, G., Hartig, O., Champin, P.-A., and Seaborne, A. (2026). RDF 1.2 concepts and abstract data model. <https://www.w3.org/TR/rdf12-concepts/>. Publicado em 7 abr. 2026. Versão desta data: <https://www.w3.org/TR/2026/CR-rdf12-concepts-20260407/>. A Recomendação vigente continua sendo RDF 1.1. Acesso em: 25 ago. 2026.
- [Kim et al. 2023] Kim, S., Chen, J., Cheng, T., Gindulyte, A., He, J., He, S., Li, Q., Shoemaker, B. A., Thiessen, P. A., Yu, B., Zaslavsky, L., Zhang, J., and Bolton, E. E. (2023). Pubchem 2023 update. *Nucleic Acids Research*, 51(D1):D1373–D1380.
- [Lavrinovics et al. 2025] Lavrinovics, E., Biswas, R., Bjerva, J., and Hose, K. (2025). Knowledge graphs, large language models, and hallucinations: An NLP perspective. *Journal of Web Semantics*, 85:100844.
- [Lehmann et al. 2015] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. (2015). Dbpedia – a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195.
- [Lewis et al. 2020] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.
- [Li et al. 2025a] Li, D., Liang, Y., Qian, S., Wu, H., Jia, W., Fu, Y., and Sun, Y. (2025a). A review of background, methods, limitations and opportunities of knowledge graph completion. *Computer Science Review*, 58:100809.
- [Li et al. 2025b] Li, M., Zhao, Y., Zhang, W., Li, S., Xie, W., Ng, S. K., Chua, T.-S., and Deng, Y. (2025b). Knowledge boundary of large language models: A survey. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5131–5157.
- [Li et al. 2025c] Li, Y., Huang, Y., Qi, G., Feng, J., Hu, N., Zhai, S., Xue, H., Chen, Y., Shen, R., and Wu, T. (2025c). Harnessing diverse perspectives: A multi-agent framework for enhanced error detection in knowledge graphs. In *Database Systems for Advanced Applications (DASFAA 2025), Part VI*, volume 15991 of *Lecture Notes in Computer Science*, pages 446–458. Springer.

- [Lima et al. 2024] Lima, G., Rodrigues, J., Machado, M., Soares, E., Fiorini, S. R., Thiago, R., Azevedo, L. G., da Silva, V. T., and Cerqueira, R. (2024). KIF: A Wikidata-Based Framework for Integrating Heterogeneous Knowledge Sources. *arXiv preprint arXiv:2403.10304*.
- [Liu et al. 2024] Liu, X., Liu, Y., and Hu, W. (2024). Knowledge graph error detection with contrastive confidence adaption. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 8824–8831.
- [Machado et al. 2025] Machado, M. d. O. C., Porto Campos, J. P., Lima, G. A. F., and Silva, V. T. d. (2025). KIF-QA: Using Off-the-Shelf LLMs to Answer Simple Questions over Heterogeneous Knowledge Bases. In *Proceedings of the 5th Wikidata Workshop*, volume 4108 of *CEUR Workshop Proceedings*. CEUR-WS.org.
- [Machado et al. 2024a] Machado, M. d. O. C., Rodrigues, J. M. B., Lima, G., and da Silva, V. T. (2024a). LLM Store: A KIF Plugin for Wikidata-Based Knowledge Base Completion via LLMs. In *KBC-LM/LM-KBC@ ISWC*.
- [Machado et al. 2024b] Machado, M. d. O. C., Rodrigues, J. M. B., Lima, G., and Silva, V. T. d. (2024b). LLM Store: A KIF Plugin for Wikidata-Based Knowledge Base Completion via LLMs. In *Proceedings of the 4th Workshop on Knowledge Base Construction from Texts: The LM-KBC Challenge @ ISWC 2024*, volume 3853 of *CEUR Workshop Proceedings*. CEUR-WS.org.
- [Mousavi et al. 2026] Mousavi, S. M., Alghisi, S., and Riccardi, G. (2026). LLMs as Repositories of Factual Knowledge: Limitations and Solutions. *IEEE Transactions on Audio, Speech and Language Processing*, 34:2213–2226.
- [Navigli et al. 2023] Navigli, R., Conia, S., and Ross, B. (2023). Biases in Large Language Models: Origins, Inventory, and Discussion. *J. Data and Information Quality*, 15(2).
- [Norouzi and Hitzler 2025] Norouzi, S. S. and Hitzler, P. (2025). Knowledge-Enhanced Geospatial QA: Integrating Wikidata Fact Verification with LLMs. In *Proceedings of the AAAI Symposium Series*, volume 5, pages 334–339.
- [Odell et al. 2022] Odell, J., Lemus-Rojas, M., and Brys, L. (2022). *Wikidata for Scholarly Communication Librarianship*. IUPUI University Library. DOI: 10.7912/9Z4E-9M13.
- [Peng et al. 2024] Peng, Y., Bonald, T., and Alam, M. (2024). Refining Wikidata taxonomy using large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM)*. ACM.
- [Regino and dos Reis 2025] Regino, A. G. and dos Reis, J. C. (2025). Can LLMs be knowledge graph curators for validating triple insertions? In *Proceedings of the Workshop on Generative AI and Knowledge Graphs (GenAIK)*, pages 87–99, Abu Dhabi, UAE. International Committee on Computational Linguistics.
- [Robinson et al. 2013] Robinson, I., Webber, J., and Eifrem, E. (2013). *Graph Databases*. O’Reilly.

- [Sáez and Hogan 2018] Sáez, T. and Hogan, A. (2018). Automatically generating wikipedia info-boxes from wikidata. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 1823–1830, Republic and Canton of Geneva, CHE. International World Wide Web Conferences Steering Committee.
- [Verborgh et al. 2014] Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Manens, E., and Van de Walle, R. (2014). Web-scale querying through Linked Data Fragments. In Bizer, C., Heath, T., Auer, S., and Berners-Lee, T., editors, *Proceedings of the 7th Workshop on Linked Data on the Web*, volume 1184 of *CEUR Workshop Proceedings*. CEUR.
- [Verborgh et al. 2016] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., and Colpaert, P. (2016). Triple pattern fragments: A low-cost knowledge graph interface for the web. *Journal of Web Semantics*, 37-38:184–206.
- [Vrandečić and Krötzsch 2014] Vrandečić, D. and Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85.
- [W3C OWL WG 2012] W3C OWL WG (2012). OWL 2 Web Ontology Language document overview (second edition). Recommendation, W3C. <http://www.w3.org/TR/2012/REC-owl2-overview-20121211/>.
- [Wikidata 2026a] Wikidata (2026a). Wikidata:Data access. https://www.wikidata.org/w/index.php?title=Wikidata:Data_access. Revisão 2497420605. Acesso em: 25 ago. 2026.
- [Wikidata 2026b] Wikidata (2026b). Wikidata:Database download. https://www.wikidata.org/w/index.php?title=Wikidata:Database_download. Revisão 2479770906. Acesso em: 25 ago. 2026.
- [Wikidata Community 2026] Wikidata Community (2026). Wikidata: Mcp. <https://www.wikidata.org/wiki/Wikidata:MCP>. Accessed: 2026-08-27.
- [Wikimedia Deutschland 2025] Wikimedia Deutschland (2025). Wikidata embedding project. https://www.wikidata.org/wiki/Wikidata:Embedding_Project. Accessed: 2026-08-27.
- [Wikimedia Deutschland 2026] Wikimedia Deutschland (2026). Wikidata vector database. https://www.wikidata.org/wiki/Wikidata:Vector_Database. Accessed: 2026-08-27.
- [Wood et al. 2014] Wood, D., Lanthaler, M., and Cyganiak, R. (2014). RDF 1.1 concepts and abstract syntax. W3c recommendation, W3C. <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [Youssef et al. 2023] Youssef, P., Koraş, O., Li, M., Schlötterer, J., and Seifert, C. (2023). Give me the facts! a survey on factual knowledge probing in pre-trained language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 15588–15605.

- [Zhang and Soh 2024] Zhang, B. and Soh, H. (2024). Extract, define, canonicalize: An llm-based framework for knowledge graph construction. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 9820–9836. Association for Computational Linguistics.
- [Zhong et al. 2023] Zhong, L., Wu, J., Li, Q., Peng, H., and Wu, X. (2023). A comprehensive survey on automatic knowledge graph construction. *ACM Computing Surveys*, 56(4):1–62.