Contents

1 Knowledge Graphs

Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Claudius Korzen
University of Freiburg, Germany

1.1 Introduction

Knowledge graphs have become an essential ingredient of information retrieval systems. There is a plethora of specialized research papers as well as surveys on the topic. We here focus on an aspect that has received little attention in existing overview papers so far: efficient indexing of and search on knowledge graphs. We explain the key ideas with many examples, which you can also try out live on a variety of (small and large) knowledge graphs; see https://qlever.cs.uni-freiburg.de/ir-book/. This should make the chapter easy to follow also for non-experts. Where there is more to know, we provide links to relevant surveys.

We also provide a gentle introduction into the basics and an insightful overview of existing knowledge graphs and their main features, strengths, and weaknesses. We have structured this chapter into sections that can be read largely independently from each other, provided that the basics (explained in Section 1.2) are understood. Wherever there is a dependency on a previous section, we point it out. Here is a quick overview over what each section is about.

What is a knowledge graph Knowledge graphs are directed graphs with labeled edges that represent structured knowledge about the world. Each vertex stands for an *entity*, for example *Sanna Marin* (former prime minister of Finland) or *Helsinki* (the capital of Finland). Each directed edge stands for a relation between two entities and the label says what the relation is. For example, there could be an edge from *Sanna Marin* to *Helsinki*, labeled *place of birth*. Equivalently, the information captured by this edge could be formulated as a subject-predicate-object *triple*, namely *Sanna Marin - place of birth - Helsinki*. Much of the beauty of knowledge graphs comes from this simple and universal data model. However, this simple model also implies a number of challenges. For example, how to cast more complex information into this framework (like Sanna Marin being elected prime minister of Finland at a certain time in a certain place) and how to query such information? We answer this and other questions in Section 1.2.

What knowledge graphs are out there The idea behind knowledge graphs is old, but early attempts like CYC have not been widely successful because they were too complex and too ambitious in the kind of knowledge they tried to capture, yet lacked the human power and the data [Lenat 1995]. The World Wide Web has brought us vast amounts of data in electronic form and the possibility of crowdsourcing. As a consequence, the field has been reborn and many large knowledge graphs have been developed over the past fifteen years. Some contain general-purpose knowledge, like Freebase, Yago, DBpedia, or Wikidata. Others contain domain-specific knowledge, like UniProt (proteins), PubChem (chemistry), OpenStreetMap (geodata), or DBLP (bibliographic data). Many big companies maintain their own knowledge graphs, for example: Google, Microsoft, Amazon, Facebook, or Wolfram Alpha. In Section 1.3, we introduce eight popular knowledge graphs and briefly discuss their characteristics, and their strengths and weaknesses.

How to search a knowledge graph The standard query language for knowledge graphs is SPARQL, which is conceptually similar to SQL (the standard query language for databases) but adapted to the simple subject-predicate-object data model. In principle, each knowledge graph can be stored in a standard database and each SPARQL query can be translated into an equivalent SQL query on that database [Chebotko et al. 2009]. But SPARQL is more natural for data modeled as triples. Like SQL, SPARQL allows semantically precise queries, for example: find all the current prime ministers of all countries in the world and when they were elected. In Section 1.4, we discuss the basics of SPARQL and two other popular query languages: *Cypher*, which is strongly related to SPARQL, and *GraphQL*, which is often mistaken as related, but actually is not.[1] SPARQL is conceptually simple, but finding the right entity names and how the sought for information is represented can be extremely hard, even for experts.[2] There are various techniques to alleviate this, notably autocompletion and automatically translating queries written in natural language to SPARQL queries. We discuss these two techniques in Section 1.6.

Engines and indexing The largest knowledge graphs with a coherent schema[3] have tens of billions of triples; see Section 1.3. When combining several such knowledge graphs, we obtain hundreds of billions of triples. Finding information in such vast datasets efficiently is a major challenge, similar to finding information in the around 50 billion pages indexed by today's web search engines. SPARQL engines either build on existing database engines (recall the equivalence of SPARQL and SQL mentioned above), or they precompute index data structures specifically designed for knowledge graphs. In Section 1.5, we describe the key techniques behind building a high-performance SPARQL engine. A popular open-source SPARQL engine built on these techniques is *QLever* [Bast et al. 2022b]. We also discuss three other popular SPARQL engines and their strengths and weaknesses (*Virtuoso*, *Blazegraph*, *Neo4j*).

[1] We include GraphQL because we did not find a lucid explanation anywhere of what GraphQL is and what it is not.

[2] This is not specific for SPARQL: related languages like SQL or CYPHER have the same problem.

[3] Note that generating enormous amounts of triples without a meaningful schema is trivial. For example, one could crawl the web and for every word on every web page, add a triple stating that this word occurs on this web page. That would give a very large knowledge graph, but not a very useful one.

Combination with text search and federated search One of the great strengths of RDF and SPARQL is the easy interoperability of different datasets. In Section 1.7, we provide a brief overview of the various ways to combine a knowledge graph with text data. The easiest way is to have text as objects of selected triples (for example, the name or description of an entity). A more powerful way is to recognize and link entities from a knowledge graph in a separate text corpus. We also provide a brief overview over the benefits and challenges of federated search. In principle, federated search allows queries that gather data from multiple knowledge graphs. For example, we may be interested in the geometric shape (such data is contained in the OpenStreetMap data) of all countries, where a given language is officially spoken (such information is contained in Wikidata). All that is needed for this are triples that relate entity ids from one dataset to entity ids from the other dataset. The challenge is to execute such queries efficiently, exchanging as little data as possible.

What else is there to know about knowledge graphs There are many more uses of and challenges associated with knowledge graphs. In their recent survey, Hogan et al. [2021] give a much broader overview and cover topics such as: graph models, schema representation, reasoning, knowledge graph construction and completion and correction. That survey also lists (in its Table 1) many recent surveys on particular aspects of knowledge graphs. None of these cover the aspects of indexing and search from the ground up like this chapter (which therefore nicely fills a gap in the literature). We here also provide other nuggets of information not found elsewhere, for example, our characterization and critical discussion of several popular knowledge graphs (Table 1.1) and our succinct description of the aforementioned *GraphQL* query language (Section 1.4.3).

The future of knowledge graphs We conclude this chapter with a short glimpse into the future (Section 1.8). In particular, we ponder the question which role knowledge graphs will play given the dramatic progress made in the fields of deep learning and language models. We believe that the two will co-exist for a long time to come, complementing each others weaknesses.

1.2 What is a knowledge graph

In this section, we explain some basic concepts and terminology and introduce a toy knowledge graph, which we will refer back to throughout the chapter.

1.2.1 Our toy knowledge graph, first version

Let us start by giving the toy knowledge graph as a set of triples, which make some statements about a well-known actress and a well-known costume designer and their respective Oscars.

<Meryl_Streep>	*<is_a>*	*<Person>* .
<Meryl_Streep>	*<gender>*	*<Female>* .
<Meryl_Streep>	*<also_known_as>*	*”Mary Louise Streep”* .

```
<Meryl_Streep>           <birth_date>     ”1949-06-22” .
<Meryl_Streep>           <won_award>      <Best_Actress> .
<Ruth_E_Carter>          <is_a>           <Person> .
<Ruth_E_Carter>          <gender>         <Female> .
<Ruth_E_Carter>          <won_award>      <Best_Costume_Design> .
<Best_Actress>           <is_a>           <Oscar> .
<Best_Costume_Design>    <is_a>           <Oscar> .
```

Such a set of triples is called a *knowledge graph*. We already noted that we can equivalently view a set of triples as a graph with directed labeled edges. Sometimes the term *knowledge base* is used as a synonym for the term knowledge graph. In this chapter we exclusively use the term knowledge graph, as it has become much more common. A graph representation of the triples above (and a few additional triples, explained later) is shown in the upper part of Figure 1.1 on page 22.

1.2.2 RDF

Modeling data as a set of subject-predicate-object triples is at the heart of the *Resource Description Framework (RDF)*. Each component of a triple is denoted by a so-called *International Resource Identifier (IRI)*; these are the identifiers in <...> brackets in the example above. An IRI is like a URI (used to give a web page a universally unambiguous name), but with an extended character set (unlike for URIs, most Unicode characters are allowed in an IRI). Objects can alternatively also be strings, called *literals* in RDF. In the example above, the third and fourth triple have a literal object. RDF also has basic support for modeling data types. For example, we could specify that the object of the fourth triple is a date and we will indeed do that in our second (more elaborate) example below. Note that we can also use RDF to specify logical relationships between predicates; for example, that the predicate <*has_author*> is the inverse of the predicate <*is_authored_by*>. For a formal definition of the RDF standard, see https://www.w3.org/TR/rdf11-concepts.

RDF is often called a data *format*, which is not correct. Instead, RDF is a data *model*, and there are various standardized ways to format – or as one often says *serialize* – data that fits that model. Our toy knowledge graph above is given in the very simple and intuitive *N-Triples* format: one triple per line, with a *dot* after each triple (to be read like a full stop terminating a sentence). A related, but more compact format is *Turtle*; we will see an example of that below. The historically first serialization was *RDF/XML*, which is falling out of fashion because it is harder to parse than N-Triples or Turtle.

1.2.3 Our toy knowledge graph, second version

The toy knowledge graph above is simpler than is typical in two important ways. First, typical IRIs are much longer, so that they are universally unambiguous (not just for the particular knowledge graph, but among all knowledge graphs on the planet, just like an URI). Second, in most modern knowledge graphs, IRIs have abstract IDs and the human-readable names and aliases are specified via dedicated predicates. This makes sense because there is typically more than one name for an entity (just think of different languages) and there are different entities that have the same name. Additionally, names can change, while identifiers should be persistent. In Section 1.3, we will introduce *Wikidata*, the largest (and de-facto standard) general-purpose knowledge graph of our time. In Wikidata, Meryl Streep (identified by the URI *<http://www.wikidata.org/entity/Q873>*) has 152 name triples as of this writing, including the following three in English, Latvian, and Ukrainian :

```
<http://www.wikidata.org/entity/Q873> <http://schema.org/name> "Meryl Streep"@en .
<http://www.wikidata.org/entity/Q873> <http://schema.org/name> "Merila Strīpa"@lv .
<http://www.wikidata.org/entity/Q873> <http://schema.org/name> "Меріл Стрiп"@uk .
```

Here, *@en*, *@lv*, and *@uk* are *language tags* that explicitly state the language of a string literal. Full IRIs are long, which makes input files in the N-Triples format large and cumbersome to read. The so-called *Turtle* format alleviates this in several ways. In particular, it allows for the definition of *prefixes* as well as for various useful shortcuts such as specifying multiple triples for the same subject without repeating the subject every time. In Turtle, and using prefixes and IRIs from Wikidata, the four triples about Meryl Streep from Section 1.2.1 (name and alias triples omitted) can be compactly specified as follows.

```
@prefix wd: <http://www.wikidata.org/entity/>
@prefix wdt: <http://www.wikidata.org/prop/direct/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

wd:Q873  wdt:P31   wd:Q5 ;                         # instance of human
         wdt:P21   wd:Q6581072 ;                   # gender female
         wdt:P569  "1949-06-22"^^xsd:dateTime ;    # date of birth
         wdt:P166  wd:Q103618 .                    # won award Oscar for Best Actress
```

Note that the prefixed names are just a shorthand for the full IRIs: for example, *wd:Q873* is exactly equivalent to *<http://www.wikidata.org/entity/Q873>*. Also note the use of the *semicolon* to specify that the following triple uses the same subject as the previous triple.[4] The comments on the right are not part of the data, but serve to clarify what the IRIs stand for in Wikidata.

1.2.4 Reification

Knowledge graphs are strongly related to databases. In fact, we could quite naturally use a standard relational database to store the triples from a knowledge graph. The distinguishing

[4] There is also the *comma* to specify that the following triple uses the same subject *and predicate* as the previous.

feature of the RDF data model is that we do not need different tables with different schemas[5]. There is just one big table with the universal subject-predicate-object schema. This is indeed simple, but gives rise to the question whether all information fits into this simple schema.

For example, consider the information about the three Oscars of Meryl Streep, for her roles in the films "Kramer vs. Kramer" (1980), "Sophie's Choice" (1983), and "The Iron Lady" (2012). A naive but **wrong** way to cast this information into triples would be as follows (temporarily using the simpler N-Triples format with short IRIs again):

```
<Meryl_Streep>   <won_award>            <Best_Supporting_Actress> .
<Meryl_Streep>   <won_award>            <Best_Actress> .
<Meryl_Streep>   <won_award_in_year>    "1980" .
<Meryl_Streep>   <won_award_in_year>    "1983" .
<Meryl_Streep>   <won_award_in_year>    "2012" .
<Meryl_Streep>   <won_award_for_film>   <Kramer_vs_Kramer> .
<Meryl_Streep>   <won_award_for_film>   <Sophie's_Choice> .
<Meryl_Streep>   <won_award_for_film>   <The_Iron_Lady> .
```

The problem with this representation is that we lose information: from these triples, we can tell which awards Meryl Streep has won, and for which films, and in which years. But we cannot tell which award belongs to which film and to which year.

This kind of information seems to require an n-ary predicate, where $n > 2$, instead of the binary predicates represented by triples.[6] The corresponding graph would then become a *hypergraph*. For the example above, we could have a 4-ary predicate called <*award*>, expressed in 5-tuples (the fifth component is the predicate name) like these:

```
<award>  <Meryl_Streep>  <Best_Supporting_Actress>  "1980"  <Kramer_vs_Kramer>
<award>  <Meryl_Streep>  <Best_Actress>  "1983"  <Sophie's_Choice>
<award>  <Meryl_Streep>  <Best_Actress>  "2012"  <The_Iron_Lady>
```

We can represent n-ary predicates as triples through a simple technique called *reification*. For each tuple from the n-ary predicate, we simply introduce a *mediator entity*, which connects to each component of the tuple through a suitably chosen binary predicate. In Wikidata, these mediator entities are called *statement nodes* and they have long alphanumerical IDs. For example, the information about Meryl Streep's 2012 Oscar is represented as follows (including the relevant prefix definitions). For compactness, we have abbreviated the IRI of the statement node, the full IRI is *wds:Q873-6ad4311a-47f9-8d9b-7c91-d387e71529ac*.

```
@prefix wd:   <http://www.wikidata.org/entity/>
@prefix wds:  <http://www.wikidata.org/entity/statement/>
@prefix p:    <http://www.wikidata.org/prop/>
```

[5] A *schema* for a table specifies a name and a datatype for each column.

[6] Note that a triple also states the predicate name, hence we need triples, and not only pairs for binary predicates.

```
@prefix ps:    <http://www.wikidata.org/prop/statement/>
@prefix pq:    <http://www.wikidata.org/prop/qualifier/>

wd:Q873               p:P166      wds:Q873-6ad4311a  .   # award statement node
wds:Q873-6ad4311a     ps:P166     wd:Q103618         ;   # which award
                      pq:P585     "2012"             ;   # point in time
                      pq:P1686    wd:Q269810         .   # for work
```

This representation in Wikidata contains several clever ideas. First, the statement nodes have a distinct prefix (*wds:*), and their alphanumerical IDs start with the ID of the entity to which the statement node belongs (*Q873* in this case). Second, the two predicates to and from the statement node have a similar (but not identical) IRI, compared to their "direct" counterpart that connects two entities without a statement node. Namely, the suffix is the same, only the prefix differs (*p:* and *ps:* and *pq:* instead of *wdt:*). Third, the prefix depends on the "role" of the predicate for this particular statement node; predicates with prefix *p:* lead from an entity to a statement node, predicates with prefix *ps:* lead from the statement node to the information that could also have been obtained directly via *wdt:*, and predicates with prefix *pq:* lead to additional information.

Technically, *wdt:P166* and *p:P166* and *ps:P166* (and there is also *pq:P166* and even more variants) are different IRIs. But they all "mean" the same predicate, just in different roles. In the RDF dataset, this is expressed by a "meta" entity *wd:P166*, which is connected to all these variants and to which the name predicate is connected. This elegantly solves the problem of having consistent names for a set of predicates with the same general "meaning".[7]

```
wd:P166   rdfs:label                    "award received" .
wd:P166   wikibase:directClaim          wdt:P166 .
wd:P166   wikibase:claim                p:P166 .
wd:P166   wikibase:statementProperty    ps:P166 .
wd:P166   wikibase:qualifier            pq:P166 .
```

These tricks and subtleties are elegant and the result of a long collective experience in designing knowledge graphs. However, they also make querying a knowledge graph hard, even for expert users. For example, it is asking a lot from a user to understand *and* know that at a certain point in a query, the variant of the predicate with prefix *pq:* is needed. We come back to this important issue when we discuss formal query languages (Section 1.4) and when we discuss how to assist the user in formulating queries in such a language (Section 1.6).

[7] Freebase (a knowledge graph introduced in Section 1.3.2) also has statement nodes, called *mediators*, but there is no mechanism to identify the corresponding different variants of a predicate (like the *P166* variants in the example above) as belonging to the same "meaning".

1.2.5 Other kinds of information

We have seen that complex *n*-ary information can be represented via triples without information loss. But what about more fuzzy information? For example, consider the following sentence about a well-known astronaut:

As Armstrong, whose great-grandfather Friedrich Wilhelm Kötter emigrated from Ladbergen to the United States in 1864, was the first man to walk on the moon in 1969, many citizens of Ladbergen became interested in their American relatives.

This sentence contains various pieces of information about entities that are found in a knowledge graph such as Wikidata (the astronaut Neil Armstrong, the municipality of Ladbergen, the Moon, etc.). But the information is rather complex and not easily cast into triples. For example, which predicate should we use to express that a person set foot on a celestial body or to express that a certain group of people became interested in a certain other group due to certain circumstances? It's not impossible, but a knowledge graph with too many predicates or entities with subtle semantics is impractical. Indeed, early attempts at constructing knowledge graphs (like the aforementioned CYC) aimed at such a comprehensive modeling of knowledge and failed.

It is one of the beauties of natural language that it allows the expression of even complex and fuzzy information in an effortless and flexible manner. Much information is therefore best kept in the form of natural language text. This makes the combination of knowledge graphs and text (which makes statements about entities from a knowledge graph in a looser manner) all the more important. We come back to this in Section 1.7.

1.3 What knowledge graphs are out there

In this section, we introduce and compare prominent knowledge graphs that are publicly available. Note that there are also a lot of *enterprise* knowlegde graphs, that is, knowledge graphs which are used by companies for internal purposes or which are the basis of their business model. However, such knowledge graphs are not publicly available and thus out of scope for this section.

Table 1.1 gives an overview of four *general-purpose* and four *domain-specific* knowledge graphs which are widely used in research and beyond. Each knowledge graph is given together with metrics describing its *size* (in the columns 2-4) and its *content* (in the columns 5-10). The precise definitions can be found under https://qlever.cs.uni-freiburg.de/ir-book.

# Tr.	the number of all triples in the knowledge graph
# S	the number of distinct subjects
# P	the number of distinct predicates
Direct	triples that do not fall into any of the following categories
Reified	triples that contain a mediator node; see Section 1.2.4
Type	triples that specify a type (e.g., via the *rdf:type* predicate)
Label	triples that specify a label, name or title (e.g., via the *rdfs:label* predicate)

Desc.	triples that specify a description (e.g., via the *schema:description* predicate)
Meta	triples that specify meta information (e.g., modification dates)

In the following, we give further details to the listed knowledge graphs, each together with a concise summary of its historical background and some general remarks. A publicly accessible SPARQL endpoint for each of these knowledge graphs is available at https://qlever.cs.uni-freiburg.de.

Name	**# Tr.**	# S	# P	Direct	Reified	Type	Label	Desc.	Meta
Wikidata	**18B**	1.9B	51K	7.8%	23.4%	19.2%	14.7%	15.7%	19.1%
Freebase	**3.1B**	125M	785K	3.3%	10.3%	31.6%	41.8%	0.8%	12.2%
DBpedia	**839M**	43M	55K	43.1%	–	20.2%	7.5%	11.0%	18.2%
YAGO 3	**121M**	15M	100	32.2%	–	20.7%	44.0%	–	3.1%
Uniprot	**112B**	23B	237	31.9%	11.7%	16.8%	10.6%	0.8%	28.0%
PubChem	**14B**	3.8B	395	10.3%	39.6%	22.6%	22.4%	–	4.3%
DBLP	**374M**	48M	68	29.6%	47.4%	15.4%	4.4%	–	3.1%
OSM	**13B**	1.2B	92K	68.7%	21.6%	8.9%	0.7%	–	–

Table 1.1 An overview of four *general-purpose* knowledge graphs (in the upper half of the table) and four *domain-specific* knowledge graphs (in the lower half of the table). An "–" entry means that the respective knowledge graph contains no triples of that category. All of these statistics have been computed using SPARQL queries. To inspect and run those queries, click on the name of the knowledge graph in the first column of the table or visit https://qlever.cs.uni-freiburg.de/ir-book.

1.3.1 Wikidata

Wikidata [Vrandecic and Krötzsch 2014] was released by the *Wikimedia Foundation* in 2012. It is currently the largest, most comprehensive, and also most popular general-purpose knowledge graph. Similarly to Wikipedia, Wikidata will probably keep this status for decades to come. Wikidata uses alphanumerical IRIs following a carefully crafted schema, in particular for its large portion of reified n-ary information; see the examples in Section 1.2. In case of disputed, uncertain, contradictory, or historical statements, Wikidata also provides information as to the *preferred* piece of information. For example, Wikidata contains 34 statements on the population of New York City at different times, with the most current one being marked as preferred via a dedicated triple. On the other hand, Wikidata is still far from complete, even for "popular" information that is available (even in tabular form) in Wikipedia articles. For example, Wikidata has information about only 9 of the 23 Olympic gold medals won by Michael Phelps.

Wikidata is multilingual and provides labels for entities and predicates in more than 400 languages. It is heavily used by different Wikimedia pages; for example Wikipedia, which uses Wikidata for automatically assembling the interlanguage links (that is, the links of a Wikipedia article to equivalent articles in other languages) or some of the infoboxes (that is, the tables that can be found in the upper right of many articles). While many older knowledge graphs (see the following sections) are based on the information from Wikipedia's infoboxes, Wikimedia's long-term goal is the other way round: to automatically assemble *all* infoboxes from Wikidata.[8]

To acquire new data, Wikidata uses a semi-crowdsourced approach. In principle, even unregistered users are allowed to add and update information via the Wikidata website. However, *high-traffic entities* (that is, entities that are used by $\geq$ 500 Wikimedia pages) can only be changed by privileged users.[9] Changes are instantly visible to the public. Wikidata regularly integrates other datasets as well, for example, open government data or the bibliographies of national libraries. Integrating the retired knowledge graph Freebase, which we introduce next, is an on-going process [Tanon et al. 2016].

1.3.2 Freebase

Freebase [Bollacker et al. 2008] was started by a company called Metaweb in 2007. It featured a web interface that allowed (registered) users to add and edit entities in a single click. In July 2010, Freebase was acquired by Google and became Google's knowledge graph [Singhal 2012]. In 2016, Freebase was closed to the public. Around that time, it was the largest general-purpose knowledge graph. Freebase allowed users to add their own predicates, which resulted in an inflated number of predicates of very limited use. Freebase contains 785 K predicates, compared to only 32 K predicates in the current version of Wikidata. However, more than 700 K of these predicates are only used in one or two triples.

Freebase provides labels and descriptions in more than 240 languages. Like Wikidata, Freebase provides a large amount of n-ary information, reified via so-called *mediator* nodes. However, while Wikidata has encoded in its schema which predicates represent the same meaning and which carry the "main" piece of information from an n-ary piece of information (see Section 1.2.4), Freebase does not contain this information. Freebase uses IRIs with alphanumerical IDs for entities, while the IRIs for predicates are human-readable, with a hierarchical structure.[10]

[8] A list of Wikipedia articles that already use a Wikidata-based infobox can be found at https://w.wiki/_2Rq.

[9] https://www.wikidata.org/wiki/Wikidata:Protection_policy

[10] For example, the IRI of Meryl Streep is *fb:m.0h0wc*, while the predicates related to n-ary award information are called *fb:award.award_winner.awards_won*, *fb:award.award_honor.award*, *fb:award.award_honor.year*, etc. The prefix *fb:* stands for <*http://rdf.freebase.com/ns/*>.

1.3.3 DBpedia

DBpedia [Lehmann et al. 2015] was released by researchers at the universities of Leipzig and Berlin in 2007. In contrast to Wikidata and Freebase, the content of DBpedia is not created manually but automatically extracted from (semi-)structured data of Wikipedia articles, for example, the infoboxes or the categories. The extracted data are then matched to the DBpedia schema using a rule-based approach. The rules and the schema can be defined and extended by (privileged) users. Since 2018, a similar approach is used to also match statements from Wikidata to the DBpedia schema [Ismayilov et al. 2018]. In Table 1.1, we analyze the *core release* of DBpedia, which only contains statements extracted from Wikipedia.

DBpedia contains many cross-references that connect the entities to equivalent entries in other knowledge graphs and is therefore seen as a *hub* in the LOD cloud. In the core release, the IDs in the IRIs of the entities are the human-readable Wikipedia page IDs (e.g., *Meryl_Streep*). The IDs in the IRIs of the predicates are also human-readable (e.g., *birthDate*). DBpedia contains many different predicates with a similar or even the same meaning, which complicates the formulation of meaningful queries considerably. For example, there are predicates *dbr:birthDate*, *dbo:birthDate*, *dbp:dateOfBirth* and *dbp:datebirth*. This is due to the use of different identifiers in the Wikipedia infoboxes and due to ambiguous rules created by the DBpedia users. Even the union of such predicates is usually far from complete, because the Wikipedia infoboxes are far from complete. DBpedia provides labels for entities and predicates in 125 languages. In principle, it also provides *n*-ary information, but the coverage and quality of that information is far below that of Wikidata or Freebase, again because of the limited information in Wikipedia's infoboxes.

1.3.4 YAGO

YAGO [Suchanek et al. 2008] was released by researchers at the Max Planck Institute for Informatics in 2006. Like DBpedia, it extracts (semi-)structured data from the infoboxes and categories of Wikipedia articles. In contrast to DBpedia, the extraction process is not crowd-sourced but combined with the the concept hierarchy of *WordNet*.[11] YAGO matches the categories of the Wikipedia articles to the concepts of WordNet and thereby establishes a huge[12] hierarchy of classes, but of questionable usefulness. In particular, there are many overly specific classes (for example, *Plant pathogens and diseases by causal agent*), many classes with unclear semantics (for example, there is a concept *entry* which contains a subset of songs), and many canonical classes are missing (for example, there is no category *person*). With respect to its predicates, YAGO's schema is very compact and much clearer than that of DBpedia, but YAGO also contains less information.

[11] WordNet, https://wordnet.princeton.edu, is a lexicon for the English language that provides sets of words and concepts with the same meaning (so-called *synsets*) and their hierarchical relations.

[12] YAGO 3 has five times more classes than Wikidata, but is over a 100 times smaller.

Over time, YAGO was extended by (1) temporal facts extracted from Wikipedia and spatial facts extracted from GeoNames[13] (YAGO 2, released in 2010) and (2) multilinguality (YAGO 3, released in 2015). In 2020, YAGO 4 was released, which, however, does not follow the original idea behind YAGO anymore, as it matches the facts of Wikidata to a schema derived from *schema.org* [Tanon et al. 2020]. For that reason, we do not analyze YAGO 4 in Table 1.1, but YAGO 3. Like in DBpedia, the IRIs of the entities and predicates are human-readable: for example, *yago:Meryl_Streep* (like the Wikipedia article) or *yago:wasBornOnDate*. Labels for entities and predicates are provided in more than 300 languages.

1.3.5 UniProt

UniProt [The UniProt Consortium 2017] is the world's largest protein knowledge graph and was released by the *UniProt consortium* in 2002. The knowledge graph is subdivided into the following parts: the UniProt Archive (*UniParc*), the UniProt knowledge graph (*UniProtKB*) and the UniProt Reference Clusters (*UniRef*). UniParc contains protein sequences in form of strings, together with some basic information like the protein names. UniProtKB contains (1) manually and automatically created annotations for the protein sequences in UniParc, for example: descriptions, classifications, or cross-references to other datasets, and (2) references to (scientific) publications that discuss the protein sequences. UniRef contains clusters of similar protein sequences. There is a lot of *n*-ary information, for example, the provenance of the annotations or metrics describing the quality of the annotations. The names, descriptions and all annotations are given in one language (English). The IDs in the IRIs of the entities are numerical (e.g., the IRI of *SARS-CoV-2* is *uniprot:2697049*), while the IDs in the IRIs of the predicates are human-readable (e.g., *core:scientificName*).

1.3.6 PubChem

PubChem [Fu et al. 2015] is the world's largest chemistry database and was released by the National Institutes of Health (NIH) in 2004. It particularly provides comprehensive information about (1) compounds, substances, proteins, genes, and their interrelations, (2) cross references to equivalent entries in other datasets (e.g., UniProt or Wikidata), and (3) references to (scientific) publications which discuss a given topic or concept. PubChemRDF [Fu et al. 2015], an official part of PubChem, provides the data as an RDF knowledge graph.

The IDs of entity IRIs are alphanumerical; for example, the IRI of Aspirin is *pubchem:CID2244*. Most predicate IRIs are based on alphanumerical IDs (for example, *cheminf:CHEMINF_000477* connects a molecule to its normalized counterpart). Labels and descriptions are only provided in English. PubChem makes heavy use of reification, but in an unusual way. For example, there is a single huge predicate *has-attribute* (with billions of triples) that links to mediator objects with IRIs that are the concatenation of subject and relation name

[13] https://www.geonames.org/

(for example, *descriptor:CID2244_Exact_Mass* or *descriptor:CID2244_Molecular_Formula*). It would be more natural (and also easier for query engines, see Section 1.5) to have a separate predicate for each relation (for example, one predicate for *exact mass* and one for *molecular formula*).

1.3.7 DBLP

DBLP [Ley 2009] is a bibliography database focused on topics from computer science which was started at the University of Trier in 1993. DBLP contains metadata of more than 5 million publications (e.g., their titles, authors, and publication dates) and of more than 2 million authors (e.g., their full names, affiliations, and links to their homepages). For many years, the dataset was distributed in form of a large single XML file. Since 2021, an RDF dump of the data is available as well.

DBLP provides entity labels like the publication titles or the author names only in one language (i.e., in the language in which the entities appear in the literature) and is therefore, from a technical point of view, unilingual. The IDs of the entity IRIs are sometimes alphanumerical and sometimes human-readable. For example, some authors are identified by IRIs with numerical IDs (e.g., *dblp_pid:6304* or *dblp_pid:10012-1*), other authors by IRIs containing their full names (e.g., *dblp_pid:PeterStone*). The IRIs belonging to predicates contain human-readable IDs (e.g., *dblp:authoredBy*). There are also reified statements that provide the authors of a publication in the same order in which they appear on the publication.

1.3.8 OpenStreetMap (OSM)

OSM [OpenStreetMap contributors 2021] is an open source project that was started by Steve Coast in 2004. Its goal is to provide freely accessible map data for the world in a crowd-sourced manner. Each OSM object has a geometric shape (a point, line, or multi-polygon, depending on the nature of the object)[14], as well as structured information in the form of key-value pairs (for example the name of a river or the type of a street). For Table 1.1 above, we have used the OSM2RDF project (https://osm2rdf.cs.uni-freiburg.de/) which converts the OSM data to RDF in a straightforward way, with one triple per shape (using WKT literals)[15], and one triple per key-value pair. We have only included OSM objects with at least one key-value pair. A related project, which provides only a smaller portion of the OSM data, is described on https://wiki.openstreetmap.org/wiki/Sophox.

1.4 How to search a knowledge graph: structured query languages

Knowledge graphs allow semantically precise queries to retrieve information. In this section, we discuss three formal query languages: SPARQL, Cypher, and GraphQL. As it turns out, GraphQL is actually *not* a query language for knowledge graphs, but often mistaken for one

[14] Typical examples are: a park bench (point), a road or river (line), or a country border (multi-polygon).

[15] See https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry

because of its name; hence we briefly discuss it here as well. Angles et al. [2017] give a more detailed account, and they also discuss *Gremlin*, a close relative of Cypher. In Section 1.6, we discuss the important topic of how to make formal query languages accessible to non-expert users (via SPARQL autocompletion and question answering).

1.4.1 SPARQL

SPARQL (*SPARQL Protocol And RDF Query Language*) is the de-facto standard query language for RDF knowledge graphs, as described in Section 1.2. We will explain the most important concepts of SPARQL's SELECT queries by example. For the full specification, see https://www.w3.org/TR/sparql11-overview; be warned though that it's not an easy read.

Example 1. The following SPARQL query asks for all female Oscar winners.

```
SELECT ?x ?award WHERE {
  ?x       <is_a>        <Person>  .
  ?x       <gender>      <Female>  .
  ?x       <won_award>   ?award    .
  ?award   <is_a>        <Oscar>
}
```

Using the toy knowledge graph from Section 1.2.1, the result is the following table with two columns and two rows:

?x	**?award**
<Meryl_Streep>	*<Best_Actress>*
<Ruth_E_Carter>	*<Best_Costume_Design>*

The query consists of a SELECT clause and a WHERE clause. The SELECT clause specifies a sequence of variables, separated by spaces (in the example: *?x ?award*). In SPARQL, variables start with a question mark. In our example, the WHERE clause consists of a sequence of four triples. Each element of a triple can be an IRI (like <is_a> or <Female>), a variable (like *?x* or *?award*) or a literal (there is none in the query above). The result of the query is a table with k columns, where k is the number of variables in the SELECT clause ($k = 2$ in our example). A row of the result table corresponds to an assignment of each variable of the WHERE clause to an IRI or literal. Each assignment must *match* in the sense that each triple of the WHERE clause exists in the knowledge graph, when plugging in the entity or literal for each variable. For the query above on our toy knowledge graph, there are two such assignments.

Example 2. Let us next look at a query with prefixed names, as explained in Section 1.2.3, and reified triples, as explained in Section 1.2.4. The knowledge graph is Wikidata, and we have omitted the definition of the prefixes to save space.[16] Since Wikidata uses IRIs with alphanumerical IDs, we added comments with the canonical names to make it easier to

[16] See https://en.wikibooks.org/wiki/SPARQL/Prefixes for the definitions of all the Wikidata prefixes.

understand the query. The query asks for all Oscars of Meryl Streep and the movies she won them for.

```
SELECT ?award_name ?film_name WHERE {
  wd:Q873      p:P166        ?m .              # Meryl Streep award received
  ?m           pq:P1686      ?film_id .        # for work
  ?m           ps:P166       ?award_id .       # award received
  ?award_id    wdt:P31       wd:Q19020 .       # instance of Academy Awards
  ?film_id     rdfs:label    ?film_name .      # canonical name
  ?award_id    rdfs:label    ?award_name .     # canonical name
  FILTER(LANG(?film_name)="en") .
  FILTER(LANG(?award_name)="en") .
}
```

Click to run query on QLever

The result on the complete Wikidata is the following table with two columns, one for each variable in the SELECT clause, and three rows, one for each of Meryl Streep's Oscars.

?award_name	**?film_name**
"Academy Award for Best Supporting Actress"@en	*"Kramer vs. Kramer"@en*
"Academy Award for Best Actress"@en	*"Sophie's Choice"@en*
"Academy Award for Best Actress"@en	*"The Iron Lady"@en*

The keyword FILTER restricts the result table to rows that fulfill the specified expression. For the query above, we filter the names by language. Otherwise, for each row in the result above, we would have obtained $k_1 \cdot k_2$ rows, where k_1 is the number of names for the award and k_2 is the number of names for the film (one name per language). On the current version of Wikidata, the result would then have $79 \cdot 42 + 76 \cdot 60 + 79 \cdot 52 = 11,986$ rows. This cross-product phenomenon is typical for SPARQL queries, and we come back to it in Section 1.5. Another expression commonly used with filters is *REGEX(?literal, r)*, where *?literal* is a variable that represents a literal. It evaluates to true if *?literal* matches the regular expression *r*; see Section 1.6.1 for examples.

Example 3. Our third query introduces yet more concepts that are explained below. The query asks for all human Oscar winners, sorted by the number of Oscars they won. Again, we have omitted the prefix definitions and added comments.

```
SELECT ?winner_id
          (SAMPLE(?winner_name) AS ?winner)
          (COUNT(?award_id) AS ?count)
WHERE {
  ?winner_id    wdt:P31            wd:Q5 .          # instance of human
  ?winner_id    p:P166/ps:P166     ?award_id .      # award received
  ?award_id     wdt:P31          wd:Q19020 .          # instance of Academy Awards
  ?winner_id    rdfs:label       ?winner_name .       # label
  FILTER (LANG(?winner_name) = “en”) .
}
GROUP BY ?winner_id
ORDER BY DESC(?count) ?winner
```

Click to run query on QLever

At the end of the query, we used so-called *solution modifiers*, with the following semantics. The *GROUP BY* aggregates all rows with the same value for the specified variables (multiple variables are possible). In the result, there will hence be exactly one row for each *winner_id*. For the other variables, we then have to state how we want their values to be aggregated. The *SAMPLE(?winner_name)* means that we want to pick the value of *winner_name* from any of the rows. Note that for the query above, there is only one (English) name per winner, and so all the rows of one winner (with the different Oscars) contain the same name anyway. The *COUNT(?award_id)* counts the number of entries in the specified column. Note that this is exactly the number of rows in the group, independent of the variable. We could have also written *COUNT(DISTINCT ?award_id)*, asking for the number of *distinct* entries in the specified column. The keyword *ORDER BY* asks for an ordering of the result rows, the keyword *DESC* asks for descending order (the default is ascending order), and the two arguments specify the primary and secondary sort key (any number of keys is possible). Counts are sorted numerically, string literals and IRIs are sorted lexicographically.

The *p:P166/ps:P166* in the WHERE clause is called a *predicate path*. It is equivalent to having the two triples *?winner_id p:P166 ?x* and *?x ps:P166 ?award_id*, but it avoids the intermediate variable *?x*. Note that using *wdt:P166* instead of *p:P166/ps:166* would ask for the *distinct* categories of Academy Awards a person won. This is because the predicate *wdt:P166* only knows that Meryl Streep won, for example, an *“Academy Award for Best Actress”*, but not how often. The following is an excerpt from the result table. The full result has over 2000 rows (one per Oscar winner). Understand that it is no coincidence that the winner names in the last entries starts with *Z*.

?winner_id	**?winner**	**?count**
wdt:Q8704	*“Walt Disney”*	*24*
wdt:Q727904	*“Cedric Gibbons”*	*11*
wdt:Q367032	*“Alfred Newman”*	*9*
…		
wdt:Q350704	*“Max Steiner”*	*3*
wdt:Q873	*“Meryl Streep”*	*3*
wdt:Q692550	*“Michael Kahn”*	*3*
…		

wdt:Q16210211	*“Zoltan Elek”*	*1*
wdt:Q220548	*“Zoran Perisic”*	*1*
wdt:Q111342771	*“Zsuzsanna Sipos”*	*1*

Example 4. With our fourth query, we introduce the OPTIONAL keyword and explain its subtle semantics. The query asks for all Canadian astronauts and their Hindi name, if they have one, and if not their Arabic name.

```
SELECT ?astronaut ?name WHERE {
  ?astronaut   wdt:P106   wdt:Q11631 .      # occupation astronaut
  ?astronaut   wdt:P27    wd:Q16 .          # nationality Canadian
  OPTIONAL { ?astronaut rdfs:label ?name . FILTER(LANG(?name)=“hi”) }
  OPTIONAL { ?astronaut rdfs:label ?name . FILTER(LANG(?name)=“ar”) }
}
```

Click to run query on QLever

In the previous three example queries, all results were tables with an entry in every cell. More generally, a table entry can also be UNDEF.[17] To understand the result of the query above, it will be crucial to understand how UNDEF values originate and what their semantics is when joining tables.

According to the SPARQL standard, the two OPTIONAL clauses in the query are processed one after the other, top to bottom. The first OPTIONAL computes a so-called *optional join* between the following two tables. The table on the left is the result of the first two lines of the query. The table on the right is the result of the *{ . . . }* after the first OPTIONAL in the query. We only show three lines of each table; the number in parentheses gives the total number of rows.

All Canadian astronauts (15)

?astronaut	
wd:Q16297	*# William Shatner*
wd:Q1687593	*# Jeremy Hansen*
wd:Q240769	*# Julie Payette*
. . .	

All entities with their Hindi name (700,475)

?astronaut	*?name*
wd:Q10002	*”*इंसकेडी*”@hi*
wd:Q17601	*”*घर्डै प्रांत*”@hi*
wd:Q240769	*”*जूली पयेटे*”@hi*
. . .	

The tables are joined on the columns with equal variable names, which in this case is only *?astronaut*. If an astronaut from the left is found in the table on the right (that is, has a name in Hindi), a row with that astronaut and name will be in the result. If an astronaut on the left is

[17] In the SPARQL standard, a result is more formally defined as a sequence of *bindings* of the variables in the SELECT clause. Each binding corresponds to a table row, with an UNDEF value in each columns where the respective variable is unbound.

not found in the table on the right, we still have a row with that astronaut in the result, but the entry in the column *?name* (which only exists in the right table) will be UNDEF. For readers familiar with SQL, this corresponds to the semantics of a *left outer join* there.

The second OPTIONAL computes an optional join between the following two tables. The table on the left is the result of the first optional join we just described. The table on the right is the result of the *{ ... }* after the second OPTIONAL from our query above. Note that it has mere presentational reasons that the neighboring rows below pertain to the same astronaut (the table on the right has many more rows than the table on the left, but we only show three that belong to an astronaut).

Canadian astronauts + Hindi name (15)

?astronaut	*?name*
wd:Q16297	*UNDEF*
wd:Q1687593	*UNDEF*
wd:Q240769	*"जूली पयेटे"@hi*
...	

All entities + their Arabic name (5,900,123)

?astronaut	*?name*
wd:Q16297	*UNDEF*
wd:Q1687593	*"جيريمي هانسن"@ar*
wd:Q240769	*"جولي باييت"@ar*
...	

These two tables have the same two column names and are hence joined on both columns. The respective first rows match because they are the same, and that row hence appears in the result as well. The respective second rows match because the entities in the first column match (both *wd:Q1687593*) and by definition UNDEF matches any value.[18] The respective third rows do not match, but the row on the left still appears in the result because it is an optional join.[19] The result of this second optional join, which is also the final result of the query, hence contains the following three rows:

?astronaut	*?name*
wd:Q16297	*UNDEF*
wd:Q1687593	*"جيريمي هانسن"@ar*
wd:Q240769	*"जूली पयेटे"@hi*
...	

In total, the final result has 15 rows, one for each Canadian astronaut, with the Hindi name for two of them, the Arabic name for nine of them, and an UNDEF name for four of them.

The query above is a typical use case for OPTIONAL and easy to understand: add Hindi names, with Arabic names as backup (and we could also have English names as further backup). In general, however, joins (optional or not) of tables with UNDEF values are complicated and hard to understand, in particular, when there are multiple join columns with UNDEF values in

[18] Indeed, if the table on the right would contain *k* different rows which all have *wd:Q1687593* in the first column, they would all match the *wd:Q1687593 UNDEF* row from the table on the left.

[19] Without the second OPTIONAL, this row would only appear in the result if the Hindi and Arabic name happen to be identical.

all of them. Note that OPTIONAL is related to UNDEF only in so far, that optional joins are a typical source of UNDEF values, but there are other sources as well.[20]

1.4.2 Cypher (Neo4j)

Cypher is a data representation and query language for a so-called *labeled property graph (LPG)*, defined in the following and often just called property graph. Cypher was developed by Neo4j, Inc. in 2011 and was originally a proprietary language, intended for use with the graph database Neo4j (discussed in Section 1.5). It became an open standard in 2015.

A property graph is very similar to a knowledge graph, with the following differences. In a property graph, each node and each edge can have one or more *properties*. A property is a key-value pair, where the key is an IRI and the value is a literal. Properties correspond to triples that have a literal object. For example, the triple *<Meryl_Streep> <birth_date> "1949-06-22"* can be represented by a property for *<Meryl_Streep>* with key *birth_date* and value *"1949-06-22"*. Each node can also have zero or more labels. Node labels correspond to triples from type-like predicates. For example, the triple *<Meryl_Streep> <is_a> <Person>* can be represented by assigning the label *Person* to the node that represents Meryl Streep in the LPG. All other triples are expressed using relationships. Each relationship is annotated with exactly one *label* that corresponds to the name of the predicate.

The following lines specify a property graph that is a slightly extended version of our toy knowledge graph from Section 1.2.1. Figure 1.1 shows the corresponding property graph and the equivalent knowledge graph.

```
( P1: Person { name: "Meryl Streep",
               also_known_as: "Meryl Louise Streep",
               birth_date: "1949-06-22" } ),
( P2: Person { name: "Ruth E. Carter" }),
( O1: Oscar  { name: "Best Actress" } ),
( O2: Oscar  { name: "Best Costume Design" } ),
( F:         { name: "Female" }),
(P1) – [:GENDER] –> (F),
(P2) – [:GENDER] –> (F),
(P1) – [:WON_AWARD {year: "1983", film: "Sophie's Choice"}] –> (O1),
(P2) – [:WON_AWARD] –> (O2)
```

Note the pictorial character of the syntax. Entities are written in parentheses, resembling a labeled node in the graph. Relationships are written as arrows with the relationship label in square brackets, resembling a labeled box.

Property graphs can be seen as an alternative way to realize *reification*, namely via properties and labels. In knowledge graphs, reification is realized via intermediate nodes. There is an

[20] For example, we can specify an arbitrary sequence of bindings explicity using a VALUES clause.

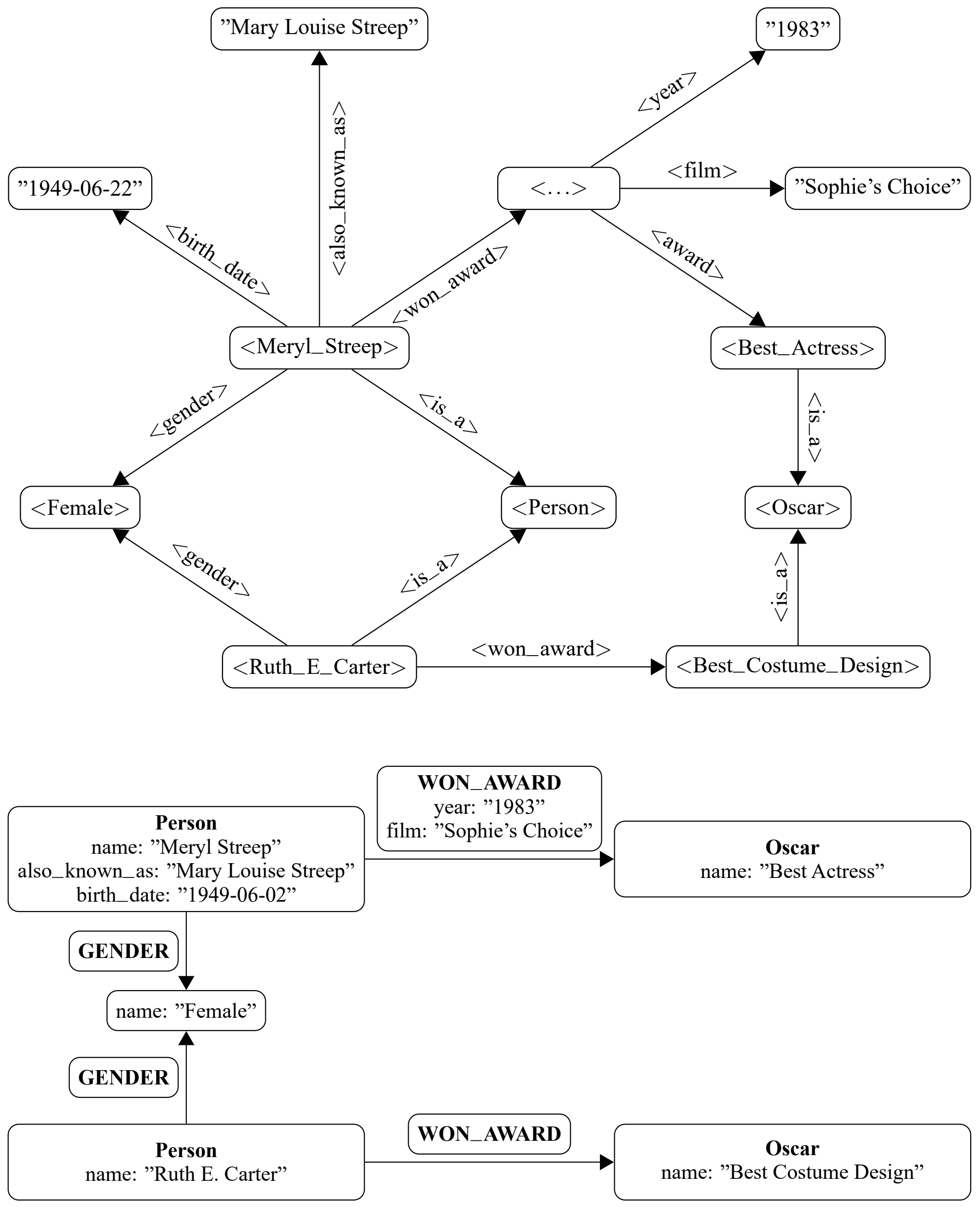


Figure 1.1 Two different representations of the knowledge graph from Section 1.2.1 plus some additional triples including reification: a regular knowledge graph (top) and a property graph (bottom). With their additional features, property graphs are able to represent the same data more densely than a regular graph. However, property graphs are not more powerful with respect to the information they can represent. The empty node in the knowledge graph is a statement node (see Section 1.2.4). Its IRI is omitted here; it is often a variation of the IRI of the subject.

important difference, however: one cannot specify a link between two properties or a property and a node. For example, in Figure 1.1, we cannot link the property value *Sophie's Choice* to a node representing that film. If needed, one could also introduce intermediate nodes in property graphs. However, typical queries would then be processed very inefficiently. This is explained in Section 1.5, where we discuss a typical query engine for property graphs (Neo4j).

Similar to SPARQL, Cypher contains different keywords for formulating queries. The *MATCH* keyword is used to describe the search pattern, and *WHERE* is used to further filter or constrain the pattern. These two correspond to what is written in the *WHERE* clause of a SPARQL query. Indeed, it makes no difference for the semantics of a Cypher query, whether a constraint is specified in the *MATCH* clause or in the *WHERE* clause. The *RETURN* clause specifies the desired output and corresponds to the *SELECT* clause of a SPARQL query. The following example shows two equivalent Cypher queries to find all Oscar winners in 1983 and the movies they won them for.

```
MATCH (x:Person) − [rel:WON_AWARD] −> (:Oscar)
WHERE rel.year="1983"
RETURN x.name rel.film
```

```
MATCH (x:Person) − [rel:WON_AWARD { year: "1983"}] −> (:Oscar)
RETURN x.name rel.film
```

Another query language for property graphs is *Gremlin*. Similar to Cypher, it organizes data in nodes and directed edges and distinguishes between "relationships", "properties" and "labels". However, the syntax of how to create and query a graph differs from Cypher; see the aforementioned survey by Angles et al. [2017].

1.4.3 GraphQL (Facebook)

GraphQL ("Graph Query Language") is an application query language which was developed for building and querying APIs. However, due to its name, it is often mistaken as a query language for knowledge graphs. GraphQL was created by Facebook in 2012 for internal use and released as an open project in 2015.

GraphQL has a *schema*, like a knowledge graph, but with an important difference. In a knowledge graph, the schema is a property of the data and the syntax of a SPARQL query is independent of that schema (and the same is true for property graphs and Cypher). In GraphQL, the schema is defined by the user (in accordance with the underlying data), and it determines the semantics and the structure of the queries that are allowed. Figure 1.2 provides an example. The schema can be implemented in any language even though it is most widely used with JavaScript. You can find a list of tools and libraries for various programming languages on GraphQL's website.[21]

[21] https://graphql.org/code/

```
type Actor {            type Oscar {            type Movie {          type Query {
  id: ID!                 id: ID!                 id: ID!               actor(id: ID!): Actor
  name: String!           category: String!       name: String!       }
  movies: [Movie]!        forWork: Movie!         year: Int
  oscars: [Oscar]         year: Int             }
}                       }
```

Figure 1.2 An example of a GraphQL schema. The first three items are regular *Object Types*. Each object type has so-called *fields*, here for example "id" or "name". These fields specify what can appear in a part of a GraphQL query that operates on our Object Type. The last item is a special type that every schema needs, the *Query Type*. It defines the *entry point* of a GraphQL query (see Figure 1.3). "ID", "String" and "Int" are built-in *scalar* types. An exclamation point indicates that this is a required field. Squared brackets represent an array.

A GraphQL query looks like a nested JSON map, where the values are either nested JSON maps or omitted. Using the example query in Figure 1.3, we will now explain how the response is produced. For each *field* in the query (for example "actor" or "name"), there has to be a function in the schema, called a *resolver (function)*. Most implementations provide simple default resolvers; other (more complicated) resolvers have to be implemented by the user. Each field produces either a scalar value (like *String* or *Int*) or an object value (like *Movie* or *Oscar*). If it produces an object value, the query has to contain a sub-selection of fields which apply to that object. This is indicated by a new set of curly brackets in the query, for example after "oscars". Each query starts with a Query type field or *entry point* (here: "actor"). Usually, the corresponding resolver function accesses a database and constructs objects from it. For example, the resolver function *actor(id: ID!)* finds the actor with the correct ID in the database and returns an *Actor* object. This object is then passed to the resolver function(s) of the next field(s) (here: "name" and "oscars"). This continues until scalar values are reached. The answer is a JSON in the same format as the query with the values filled out with the scalar values returned by the resolver functions.

Despite the name, GraphQL's purpose and scope is quite different from query languages like SPARQL or Cypher. First, due to the application-specific schema and the lack of global IRIs, GraphQL is not intended to combine data from different sources. Second, typical GraphQL queries are less expressive. In particular, only tree queries are possible and functions like filtering, sorting and grouping are not supported by default. In principle, users can implement any additional feature using a suitable resolver function.

```
query {
  actor(id: ”873”) {
    name
    oscars {
      category
      year
    }
  }
}
```

```
{
  ”data”: {
    ”actor”: {
      ”name”: ”Meryl Streep”
      ”oscars”: [
        { ”category”: ”Best Supporting Actress”
          ”year”: 1980 }
        { ”category”: ”Best Actress”
          ”year”: 1983 }
        { ”category”: ”Best Actress”
          ”year”: 2012 }
      ]
    }
  }
}
```

Figure 1.3 An example of a GraphQL query (left) and the response (right). Note that the schema in Figure 1.2 knows what value type each field returns. For example, “name” returns a String, “year” an Int, and “oscars” returns an array of *Oscar* objects.

1.5 Engines and indexing

Knowledge graphs contain up to tens of billions of triples (Section 1.3 gave several examples), and they continue to grow. In this section, we explain how data of such scale can be processed and queried efficiently. We focus on three techniques: representation via monotonic *object identifiers*, indexing via *triple permutations*, and query planning via *cost estimation*. Several SPARQL engines have been built based on these core ideas, in particular the QLever SPARQL engine (https://qlever.cs.uni-freiburg.de), which provides the SPARQL endpoints for the knowledge graphs discussed in Section 1.3. We also discuss three other popular SPARQL engines (*Virtuoso*, *Blazegraph*, and *Neo4j*) and their respective strengths and weaknesses. Ali et al. [2021] provide a much broader overview of techniques and list more than a hundred SPARQL engines together with their feature sets. However, in that survey each particular technique is described only briefly and rather formally, whereas our explanations here are more elaborate and also suitable for newcomers to the field.

1.5.1 Object identifiers

Call the set of distinct IRIs and literals of a knowledge graph its *vocabulary*. Many query engines work with *object identifiers (OIDs)*, which map each IRI or literal from the vocabulary to a unique integer ID. Each OID is stored in a fixed number of bytes, typically either 6 bytes (enough for 281 trillion OIDs) or 8 bytes (corresponding to a *word* on a 64-bit machine). The input data is then stored using OIDs, and for a given SPARQL query, the (relatively few) IRIs and literals are also mapped to OIDs. Most of the query processing can then be carried out efficiently with tables of fixed-size integers, instead of with tables of strings. To produce the final result, the OIDs are mapped back to the corresponding strings.

To illustrate this, consider the following toy knowledge graph and the subsequent mapping of its IRIs and literals to OIDs:

<Neil_Armstrong>	*<Profession>*	*<Astronaut>*
<Neil_Armstrong>	*<Nationality>*	*<USA>*
<Neil_Armstrong>	*<Date_of_birth>*	*"1930-08-05"*
<Liu_Yang>	*<Profession>*	*<Astronaut>*
<Liu_Yang>	*<Nationality>*	*<China>*

<Astronaut>	*#0*	*<Liu_Yang>*	*#3*	*<Profession>*	*#6*
<China>	*#1*	*<Nationality>*	*#4*	*<USA>*	*#7*
<Date_of_birth>	*#2*	*<Neil_Armstrong>*	*#5*	*"1930-08-05"*	*#8*

These OIDs have an additional important property, namely, that they are *monotonic*. Informally, this means that the "natural" order of the IRIs and literals is preserved. Formally, there must be a total and easy-to-compute order $<_{\text{OID}}$ on the OIDs, such that for arbitrary strings s_1, s_2 from the vocabulary, it holds that $\text{oid}(s_1) <_{\text{OID}} \text{oid}(s_2)$ if and only if $s_1 < s_2$, where $<$ is the order of IRIs and literals as defined by the SPARQL standard. In the example above, $<_{\text{OID}}$ is simply the natural integer order of the OIDs.[22]

Monotonic OIDs allow a particularly efficient implementation of operations that need to consider the order of IRIs and literals according to the SPARQL standard. This includes: ORDER BY (sorting the result using one or several variables as sort key), FILTER with relational expressions like $\leq$ or $\geq$, and REGEX expressions that check whether a given entity or literal starts with a given prefix. It should be noted that ORDER BY clauses are very frequent in SPARQL queries, and as we will see in Section 1.6.1, an efficient processing of prefix REGEX expressions is crucial for efficient autocompletion for SPARQL queries.

Using monotonic OIDs, these operations can be computed using only the OIDs, which is much cheaper than working with the underlying IRIs or literals, provided that $<_{\text{OID}}$ is easy to compute. This is trivial for the example above, where $<_{\text{OID}}$ is just the normal integer order, but in practice, matters are a bit more complicated. For example, QLever encodes integers, floating point numbers and dates directly in its 8-byte OIDs, using 4 bits to specify the datatype and 60 bits for the actual value. It then looks like already the trivial integer order on such OIDs fulfills the monotonicity property, but that is not true. For example, when encoding (signed) integers in the standard way using two's complement, the negative numbers come *after* the positive

[22] Section 15.1 of the SPARQL standard does not define a total order on all possible RDF terms. However, it does specify that IRIs come before literals, hence *"1930-08-05"* correctly gets the largest OID in the example.

numbers in the trivial OID order. For floating point numbers and dates, the situation is even more complicated. Still, with clever bit fiddling, it is possible to implement a $<_{\text{OID}}$ order that can be computed very efficiently and has the monotonicity property.

When the knowledge graph is static and given in advance, monotonic OIDs are straightforward to compute. When the knowledge graph is updated, or when new entities or literals are introduced as part of a query,[23] maintaining monotonic IDs becomes more challenging. Of the four SPARQL engines discussed in detail below, only QLever implements monotonic OIDs, while Virtuoso, Blazegraph, and Neo4j do not.

1.5.2 Triple permutations

A key idea used in most high-performance SPARQL engines is to store the set of given triples multiple times, but in different permutations. The six possible permutations are *SPO*, *SOP*, *PSO*, *OSP*, *POS*, and *OPS*, where *S* stands for subject, *P* stands for predicate, and *O* stands for object. For example, the *SPO* index and *PSO* index relating to the knowledge graph from Section 1.5.1 stores the triples as follows:

SPO index			*PSO index*		
S=#3	*P=#4*	*O=#1*	*P=#2*	*S=#5*	*O=#8*
S=#3	*P=#6*	*O=#0*	*P=#4*	*S=#3*	*O=#1*
S=#5	*P=#2*	*O=#8*	*P=#4*	*S=#5*	*O=#7*
S=#5	*P=#4*	*O=#7*	*P=#6*	*S=#3*	*O=#0*
S=#5	*P=#6*	*O=#0*	*P=#6*	*S=#5*	*O=#0*

In the *SPO* index, the triples are sorted by subject, then predicate, then object. In the *PSO* index, the triples are sorted by predicate, then subject, then object. Note that in the examples above (and also in the following examples), the IDs are prefixed by "#" and "*S*=", "*P*=" or "*O*=", depending on their position in a triple. This is just for the sake of explanation; in reality, only the integer IDs are stored. In fact, the first column is not stored explicitly, because it consists of long sequences of the same ID. Instead, we store only the second and third column (row-wise), and have a separate map that for each ID of the first column, stores the index of the first row pertaining to that ID. For the *SPO* index above, the map for *S* would be $\{3:0;\ 5:2\}$, and the table would be $[(4,1),(6,0),(2,8),(4,7),(6,0)]$.

To understand how these indexes can be used for efficient query processing, consider the following SPARQL query:

SELECT ?person ?nationality WHERE {
?person <Nationality> ?nationality .
} ORDER BY ?person

[23] For example, this can happen when using VALUES or SERVICE; see Section 1.7.3.

To get all matches for the triple in the *WHERE* clause, we could use either the *PSO* or the *POS* index. In both of them, the matches for *?person ?nationality* form a contiguous segment, marked blue in the following example (recall from the OID table in Section 1.5.1 that #4 stands for *<Nationality>*). Since the SPARQL query asks for the results ordered by *?person*, we take the *PSO* index, because in it the matching rows are already ordered by *S*, which stands for *?person* in this case. Note that this only works when using monotonic OIDs, otherwise we would still have to sort the result afterwards. Retrieving a segment from one of the precomputed indexes is called a *SCAN* operation.

PSO index			*Result Table with OIDs*		*Result Table with IRIs*	
P=#2	*S=#5*	*O=#8*	*#3*	*#1*	*<Liu_Yang>*	*<China>*
P=#4	***S=#3***	***O=#1***	*#5*	*#7*	*<Neil_Armstrong>*	*<USA>*
P=#4	***S=#5***	***O=#7***				
P=#6	*S=#3*	*O=#0*				
P=#6	*S=#5*	*O=#0*				

Things get more interesting when SPARQL queries contain two or more triples. For an illustration, consider the following SPARQL query:

```
SELECT ?person ?nationality ?profession WHERE {
  ?person <Nationality> ?nationality .        T1
  ?person <Profession> ?profession .          T2
}
```

For each individual triple, the result is a table with two columns: *?person ?nationality* for T_1 and *?person ?profession* for T_2. Just like for the previous query, each of these tables can be obtained with a SCAN operation on either the *PSO* index or the *POS* index. Since the two triples have the same subject variable *?person*, we need to join these two tables on their respective first columns. We therefore perform the SCAN on the *PSO* index, so that both columns are already sorted by their respective join columns, and we can perform an efficient linear-time merge JOIN. This works with any order of the OIDs, in particular also for non-monotonic OIDs. Here is an illustration, where the triples in blue correspond to T_1 and the triples in green to T_2.

PSO index			*Result Table*			*Result Table with IRIs*		
P=#2	*S=#5*	*O=#8*	*#3*	*#1*	*#0*	*<Liu Yang>*	*<China>*	*<Astronaut>*
P=#4	***S=#3***	***O=#1***	*#5*	*#7*	*#0*	*<Neil Armstrong>*	*<USA>*	*<Astronaut>*
P=#4	***S=#5***	***O=#7***						
P=#6	***S=#3***	***O=#0***						
P=#6	***S=#5***	***O=#0***						

1.5.3 Query planning

In the examples above, we picked the index such that the necessary JOIN operations could be carried out as efficiently as possible. In complex SPARQL queries, there are many such choices to be made, not only the choice of index for each SCAN operation, but also the order of the JOIN operations. Consider the following SPARQL query, which asks for all married couples in the knowledge graph who were born on the same date.

SELECT ?person_1 ?person_2 ?birth_date WHERE {
  ?person_1 <Date_of_birth> ?birth_date . T_1
  ?person_2 <Date_of_birth> ?birth_date . T_2
  ?person_1 <Spouse> ?person_2 . T_3
}

Let us look at two possible query plans to compute the result of this query. We assume that we always *materialize* (fully compute) the intermediate result tables.[24] The first query plan is as follows.

(1) Scan the *POS* index for the sets of triples relating to T_1 and T_2 (in any order).
(2) Join both sets of triples on *?birth_date*.
(3) Sort the result from step 2 by *?person_1* and *?person_2*.
(4) Scan the *PSO* index for the set of triples relating to T_3.
(5) Join the results from steps 3 and 4 on two columns: *?person_1* and *?person_2*.

This query plan is expensive, because the intermediate result from the JOIN operation in step 2 is huge and therefore expensive to compute (if we have n different days, and k people are born on each of these, the result has $n \cdot k^2$ rows). Here is an alternative query plan:

(1) Scan the *PSO* index for the sets of triples relating to T_1 and T_3 (in any order).
(2) Join both sets of triples on *?person_1*.
(3) Sort the result from step 2 by *?person_2* and *?birth_date*.
(4) Scan the *PSO* index for the set of triples relating to T_2.
(5) Join the results from steps 3 and 4 on two columns: *?person_2* and *?birth_date*.

This query plan is cheaper because any fixed person has only a single date of birth and only few spouses (if any), and thus the result from step 2 is much smaller than for the first query plan.

But how to determine the cheapest query plan, without executing all possible query plans in advance? The typical approach is to *estimate* the cost of some or all query plans.[25] In the following, we describe an approach that estimates the following statistics for each intermediate result table:

(1) The size s of the table, that is, the number of rows.

[24] This can be avoided in special cases, for example when a SPARQL query has a *LIMIT k* clause, and k rows of the result can be obtained without exploring all matches for the various parts of the query.

[25] The standard approach is via dynamic programming, which we do not described in detail here. That way, we can discard a subset of the query plans before we have fully estimated their cost, in case the estimate for a partial plan is already more expensive than the estimate for a full plan we have already found.

(2) For each column i, the number d_i of distinct elements in that column.
(3) For each column i, the average multiplicity m_i of an element in that column.

Trivially, $d_i \cdot m_i = s$ for each column i, so that one of these three quantities is redundant. It's still good to have all three defined because it makes the following formulas easier to understand.

The basic operations of each query plan are the index SCANs. We assume that we have the exact values of s, d_i, and m_i for the results of these SCANs (they can be easily precomputed during the indexing). We next describe how we estimate these values for the results of a JOIN operation, given the estimates for the input tables. Note that for full SPARQL support, there are also many other kinds of operations for which we would need such estimates, like sorting a table by a particular column or filtering a table by a particular criterion. We omit them here for brevity.

Without loss of generality, assume that the join column is the first column in both input tables and in the result table. Let T' and T'' be the two input tables, and let s' and s'' be their sizes, and let d'_1, d''_1, m'_1, and m''_1 be the distinctness and multiplicity of their respective join columns.

Then, we can estimate the distinctness and multiplicity of the join column of the result table as follows, where α is a correction factor in the range $[0,1]$ explained below

$$d_1 = \alpha \cdot \min(d'_1,\ d''_1) \text{ and } m_1 = m'_1 \cdot m''_1\,.$$

Using this, we get the following estimation for the size of the result table

$$s = d_1 \cdot m_1 = \alpha \cdot \min(d'_1,\ d''_1) \cdot m'_1 \cdot m''_1\,.$$

Let us explain the intuition behind these formulas.

Distinctness Assume that all values of the join column of one table (without loss of generality, let us call it T'') occur in the join column of the other table (without loss of generality T'). Then, the distinctness d''_1 of table T'' cannot be larger than the distinctness d'_1 of table T' and the distinctness of the result table T is equal to the distinctness of T''. In other words, $d = \min(d'_1, d''_1)$, which matches the formula with $\alpha = 1$. The result table might be smaller when the join column of T'' contains values that do not occur in T'. This is reflected by the tuning constant α. The QLever SPARQL engine uses $\alpha = 0.7$. The following tables show an example where $m'_1 = m''_1 = 1$.

T'	$s' = 3$
#14	#15
#38	#42
#57	#13
$d'_1 = 3$	$d'_2 = 3$
$m'_1 = 1$	$m'_2 = 1$

T''	$s'' = 2$
#14	#97
#57	#55
$d''_1 = 2$	$d''_2 = 2$
$m''_1 = 1$	$m''_2 = 1$

	T	$s = 2$
#14	#15	#97
#57	#13	#55
$d_1 = 2$	$d_2 = 2$	$d_3 = 2$
$m_1 = 1$	$m_2 = 1$	$m_3 = 1$

Multiplicity Assume that each value in the join column of T' has multiplicity m'_1 and each value in the join column of T'' has multiplicity m''_1.[26] Then, the number of rows in T for each value contained in the join columns of both input tables is $m'_1 \cdot m''_1$. Since *all values* in the result table occur $m'_1 \cdot m''_1$, this is also true on average, i.e. $m_1 = m'_1 \cdot m''_1$. The following tables show an example where $d'_1 = d''_1 = 1$.

T'	$s' = 3$
#57	#15
#57	#42
#57	#13
$d'_1 = 1$	$d'_2 = 3$
$m'_1 = 3$	$m'_2 = 1$

T''	$s'' = 2$
#57	#97
#57	#55
$d''_1 = 1$	$d''_2 = 2$
$m''_1 = 2$	$m''_2 = 1$

T	$s = 6$	
#57	#15	#97
#57	#15	#55
#57	#42	#97
#57	#42	#55
#57	#13	#97
#57	#13	#55
$d_1 = 1$	$d_2 = 3$	$d_3 = 2$
$m_1 = 6$	$m_2 = 2$	$m_3 = 3$

Other columns Since T itself can be the input to another JOIN operation on any column, we also need an estimation of the distinctness d_j and multiplicity m_j for each column j (not just of the join column). In the following, we will explain how to estimate d_j, which also gives us an estimation for m_j since we already have an estimation for the total size s of T and $m_j = s/d_j$ for any column j. Again, without loss of generality, we can make the following assumptions (just as illustrated in the examples above):

(1) The join column is the first column in both input tables and in the result table.
(2) The column we are interested in, originates from the second column of table T'.
(3) The column we are interested in, is the second column in the result table (that is $j = 2$).

Then we estimate the distinctness in column $j = 2$ of the result table as

$$d_2 = \min(d'_2,\, m'_1 \cdot \alpha \cdot \min(d'_1,\, d''_1))\,.$$

Let us also understand the intuition behind this formula.

Case 1: Assume d'_2 is small, in particular smaller than $\alpha \cdot m'_1 \cdot \min(d'_1,\, d''_1)$. Using the above formula, our estimation will be $d_2 = d'_2$. Note that d_2 can never be larger than d'_2, because only two things can happen when column 2 from T' gets copied to T: values get lost (because not all values in the join column of T' may be contained in the join column of T'') and values get duplicated (because of the "cross-product effect", like in the first example above). Hence, in this case our estimation is good, because d'_2 is small by our assumption and it is an upper bound for d_2.

[26] Note that, by definition, this is true *on average*. The closer all these multiplicities are to the average, the better our estimation tends to be.

Case 2: Now assume d'_2 is large. We can estimate the number of distinct elements in the join column of T' that make it to T as $\alpha \cdot \min(d'_1,\, d''_1)$, just like for our distinctness formula above. Each of these "bring" m'_1 values from column 2 on average. Since d'_2 is large, most of them are distinct and $d_2 = m'_1 \cdot \alpha \cdot \min(d'_1,\, d''_1)$ is a good estimation.

1.5.4 Further improvements

We have described three key techniques: object IDs, triple permutations, and cost estimation for query planning. These three already form a solid basis for building a high-performance SPARQL engine, but there are still a lot of challenges to address. Here is an incomplete list:

(1) Monotonic object identifiers for dynamic knowledge graphs, in particular, new values.
(2) SPARQL queries with a *LIMIT* that don't require full materialization of the result.
(3) Compression of the indexes and of the vocabulary.
(4) Caching of intermediate results and making the query planner aware of cached results.
(5) Efficient support of other SPARQL features like *OPTIONAL*, *UNION*, *MINUS*, …
(6) Efficient support of predicate paths (explained in Example 3 of Section 1.4.1).

For an overview of techniques, we again refer the reader to the survey by Ali et al. [2021]. For an open-source query engine implementing all of these, see https://github.com/ad-freiburg/qlever, the basic architecture of which is described in Bast and Buchhold [2017] and [Bast et al. 2022b].

1.5.5 Virtuoso

Virtuoso [Erling and Mikhailov 2009], released by *OpenLink Software* in 1999, is the most widely used SPARQL engine in research.[27] It is written in C, and based on a database system (SPARQL queries are translated to SQL), but provides full RDF and SPARQL support. In particular, it provides the official SPARQL endpoint for knowledge graphs as large as UniProt (80B triples); see Section 1.3. Virtuoso uses 8-byte OIDs for IRIs and long literals, while literals with ≤ 12 characters are stored without indirection. The OIDs do not have the monotonicity property described above. The following triple permutations are stored: *PSOG*, *POGS*, *SP*, *OP*, and *GS*. Here, *G* denotes the knowledge graph (SPARQL has support for multiple graphs, which can be addressed separately or altogether in a query). The index can be partitioned and distributed over several machines. Query planning is limited to the determination of the join order, where cost estimates are obtained via sampling the data. Query plans can be analyzed only for the translated SQL query. Intermediate results are materialized only if necessary; see the discussion above.

1.5.6 Blazegraph

Blazegraph (formerly known as *Bigdata*) [Systap 2013] is an open source database engine released by *Systap* in 2006. It is the query engine behind the official Wikidata query service[28];

[27] There is a commercial version and an open-source version; the latter is usually used in research.

[28] https://query.wikidata.org/

it was selected at the time due to Wikimedia's open-source requirements. However, they are currently looking for a replacement because of Blazegraph's inactivity. Blazegraph also uses object IDs with 8 bytes and indexes the triples in three permutations (*SPO*, *POS*, *OSP*), using a B+ Tree for each. A fourth dimension (like the graph name *G* above) is supported optionally, in which case there are six indices. Blazegraph also supports the full SPARQL standard and can be distributed over multiple machines. Blazegraph is written in *Java* and clearly lags behind in performance compared to Virtuoso for many typical queries. A recent performance comparison against Virtuoso and QLever on a large set of queries can be found in [Bast et al. 2022b].

1.5.7 Neo4j

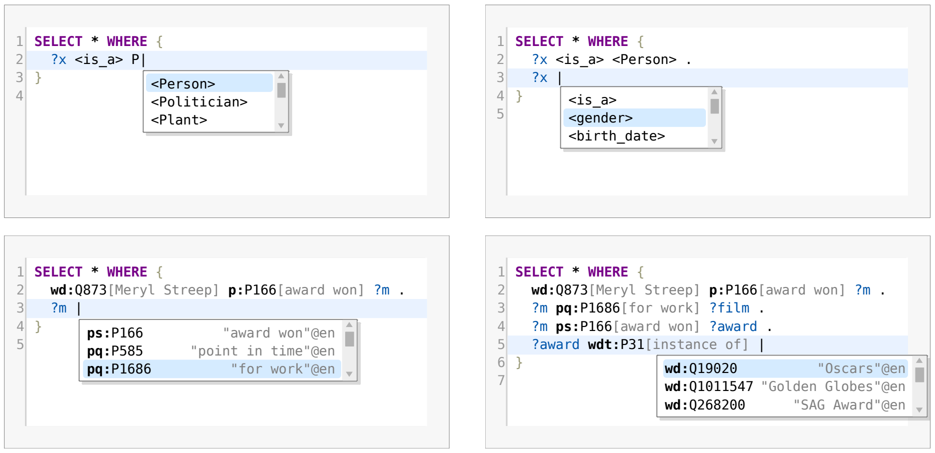
Neo4j was started by *Neo4j, Inc.* in 2007, and is the most widely used graph database for labeled property graphs in industry. Neo4j is written in Java and supports Cypher, a variant of SPARQL, described in Section 1.4.2. In principle, Neo4j can be used to index and query knowledge graphs, but not generally efficiently so. In a nutshell, Neo4j has a record for each node (entity), and for each record, stores pointers to the properties and relationships of that node; see Figure 1.1. This design is efficient for retrieving information for a small set of entities and their neighborhood, or for traversing the graph starting from such a set of entities. It is extremely inefficient for queries that involve joins of large result sets. For example, consider a query that asks for all persons of female gender. A typical SPARQL engine processes this query by fetching precomputed lists for all entities of type *Person* and for all entities with gender *Female*,[29] and then joining these two lists. Neo4j processes this query by iterating over all nodes with label *Person*[30] and for each node follow pointers to the relationship *gender* and check whether the adjacent node says *Female*.

1.6 How to search a knowledge graph: assisting the user

In Section 1.4.1 we have seen how to write formal SPARQL queries. SPARQL is conceptually easy, because queries can be formulated as lists of triples, just like the data. However, formulating the right query is often very hard in practice, because it requires knowing the schema of the underlying knowledge graph as well as knowing the IRIs and literals relevant for the query. There are literally hundreds of user interfaces to help users to explore RDF data or formulate SPARQL queries. In this section, we focus on the two most natural and most widely used forms of assistance: SPARQL Autocompletion (Section 1.6.1) and Question Answering, that is, translating a natural language question into the corresponding SPARQL query (Section 1.6.2).

[29] These would both be segments of the *POS* or *OPS* index, assuming that the corresponding triples are stored via predicates that stand for "type" and "gender", and object that stands for "Person" and "Female", respectively.

[30] Assuming that the types are represented via labels, which would be the natural thing to do in a property graph.



```
SELECT * WHERE {
  ?x <is_a> P|
}
          <Person>
          <Politician>
          <Plant>
```

```
SELECT * WHERE {
  ?x <is_a> <Person> .
  ?x |
}
      <is_a>
      <gender>
      <birth_date>
```

```
SELECT * WHERE {
  wd:Q873[Meryl Streep] p:P166[award won] ?m .
  ?m |
}
      ps:P166         "award won"@en
      pq:P585      "point in time"@en
      pq:P1686         "for work"@en
```

```
SELECT * WHERE {
  wd:Q873[Meryl Streep] p:P166[award won] ?m .
  ?m pq:P1686[for work] ?film .
  ?m ps:P166[award won] ?award .
  ?award wdt:P31[instance of] |
}
                  wd:Q19020           "Oscars"@en
                  wd:Q1011547 "Golden Globes"@en
                  wd:Q268200       "SAG Award"@en
```

Figure 1.4 Four screenshots of the SPARQL autocompletion in action, with three suggestions each. Top-left and top-right: Examples 1 and 2 on a knowledge graph like that of Section 1.2.1, where the IRIs are directly understandable for a human. Bottom-left and bottom-right: Example 3 and a continuation not described in the text, for a query on Wikidata, where IRIs are alphanumeric and names are obtained via dedicated predicates. Note that the grey names in square brackets are not part of the formal SPARQL query, but just there to help the user understand the query better.

1.6.1 SPARQL Autocompletion

Autocompletion is a natural way to address the two problems mentioned above (having to know the schema and the right IRIs and literals). In this subsection, we describe how to aid a user in incrementally constructing a SPARQL query by providing suggestions after every keystroke. The suggestions have the following two important properties:

(1) The suggestions are *context-sensitive* in the sense that they continue the part of the query already typed in a meaningful way.
(2) The suggestions are *ranked* by some measure of relevance, so that the continuation sought by the user is as high up in the list of suggestions as possible.

These properties will become very clear in the examples below. Bast et al. [2022b] provide a formal definition, together with an extensive quality and performance evaluation and an account of related work on the topic. A demo of an autocompletion system based on these ideas, for many of the knowledge graphs from Section 1.3, is available at https://qlever.cs.uni-freiburg.de. Figure 1.4 provides four screenshots.

Since the suggestions should come from the knowledge graph, for which the user is trying to formulate their query, it turns out that the suggestions can themselves be formulated as SPARQL queries, called *autocompletion (AC) queries* in the following. This is a very elegant idea, because it allows the realization of autocompletion without additional software,[31] using only the SPARQL engine that is already there.

In the following we explain this via three examples. To understand these queries, you should be familiar with the SPARQL concepts presented in Section 1.4.1. For an interactive experience, it is crucial that the AC queries are processed fast, which is a major challenge when the knowledge graph is large. We disregard the aspect of efficiency here and only mention that a SPARQL engine along the lines of Section 1.5 can be extended to process AC queries fast. Such an extension is described by Bast et al. [2022b].

Example 1. Assume that we have typed the body of the SPARQL query in Example 1 in Section 1.4.1 until before the first object; see below. The _ symbol marks the cursor position and the user has typed the prefix *"P"*. The desired object at this position is *<Person>*.

*?x <is_a> P_*

The following AC query computes a table containing all objects *?entity* (these are IRIs) and their name *?name* (these are literal strings), such that the name starts with *P* and the triple *?x <is_a> ?entity* exists. The table is sorted in descending order of the number of such triples for each *?entity*.

```
SELECT ?entity ?name (COUNT(?x) AS ?score) WHERE {
  ?x <is_a> ?entity .
  BIND (STR(?entity) AS ?name) .
  FILTER REGEX(?name, "^P")
} GROUP BY ?entity ?name ORDER BY DESC(?score)
```

Click to run query on QLever

The keyword BIND assigns a value, in this case *STR(?entity)*, to a new variable, in this case *?name*. The first three result rows for that query look as follows. The numbers are from a simplified version of Freebase, called *Freebase Easy* [Bast et al. 2014].[32]

```
<Person>        "Person"       3970825
<Politician>    "Politician"   127809
<Plant>         "Plant"        60459
```

Note that for this knowledge graph, each name is simply the corresponding IRI converted to a string. This is not generally the case, as we can see in Figure 1.4 and Example 3 below. Also note that the desired entity appears in the first row of the table, and that, by construction, all suggested entities lead to a non-empty result.

[31] Except of course the code for the actual user interface. But there is no need for a separate program that reads or preprocesses the knowledge graph in any way.

[32] A SPARQL endpoint for Freebase Easy, with which the results for our examples here can be reproduced, can also be found under the aforementioned https://qlever.cs.uni-freiburg.de.

Example 2. Now assume that we have typed the SPARQL query a little bit further. The desired predicate at the cursor position is *<gender>* and the user has typed no prefix yet to narrow down the search.

```
?x <is_a> <Person> .
?x _
```

The following AC query gives us a ranked list of *predicates* that lead to a non-empty result. The score for each predicate is the number of persons (that is, entities matching the first triple) that have a triple with that predicate. If a person has several triples with the same predicate, we only count the predicate once, hence the DISTINCT.

```
SELECT ?entity ?name (COUNT(DISTINCT ?x) AS ?score) WHERE {
  ?x <is_a> <Person> .
  ?x ?entity [] .
  BIND (STR(?entity) AS ?name) .
} GROUP BY ?entity ?name ORDER BY DESC(?score)
```

Click to run query on QLever

The square brackets *"[]"* are used instead of introducing a new variable that remains unused in the rest of the query. The first three result rows for this AC query are as follows:

<is_a>	*"is_a"*	*3970825*
<gender>	*"gender"*	*2276146*
<birth_date>	*"birth_date"*	*1915167*

The desired predicate comes second in this table and, again by construction, the AC query only returns predicates that lead to a hit. The suggestions are ranked by how often each of them occurs with the set of entities defined by the part of the query already typed. Because of this, we did not have to type a single letter here to get very good suggestions.

Example 3. Our last example is based on Wikidata, where entities have alphanumerical IRIs, and the name is obtained via a dedicated predicate *rdfs:label*. Assume that we have typed the body of the SPARQL query in Example 2 in Section 1.4.1 this far:

```
wd:Q873 p:P166 ?m .
?m _
```

Again, the cursor is at the position of the predicate and the user has not yet typed a prefix at this position. Recall that *wd:Q873* stands for *Meryl Streep* and *p:P166* connects this entity to all statement nodes *?m* pertaining to one of her awards. The desired token is *pq:P1686*, which leads us to the awarded films.

The following AC query gives us a list of predicates, along with their English[33] names, that lead to results at this point. The score is analogous to that of the previous example.

[33] Note that we could easily have names in another language here.

```
SELECT ?entity ?name (COUNT(DISTINCT ?m) AS ?score) WHERE {
  wd:Q873 p:P166 ?m .
  ?m ?entity [] .
  ?entity_normalized ?tmp ?entity .
  ?entity_normalized rdfs:label ?name FILTER (LANG(?name) = "en")
}
GROUP BY ?entity ?name
ORDER BY DESC(?score)
```

Click to run query on QLever

The third and fourth triple connect a predicate to its label on Wikidata, see Section 1.2.4 for details. The first three result rows for this AC query look as follows:

ps:P166	*"award received"@en*	*33*
pq:P585	*"point in time"@en*	*23*
pq:P1686	*"for work"@en*	*10*

These rows tell us that Wikidata knows about 33 awards of Meryl Streep, 23 points in time for these awards, and 10 works awarded. Our desired predicate comes third, and again all suggestions are context-sensitive by construction. Without these suggestions it would require extremely intimate knowledge of Wikidata to know that we need the predicate suffixes *P166* and *P1686* and the prefixes *ps:* (which stands for the main entity of a statement node) and *pq:* (which stands for additional properties of a statement node).

1.6.2 Question Answering

Context-sensitive autocompletion is great, but it still requires basic knowledge of the SPARQL syntax. Ideally, the user can ask questions in natural language (just as they would ask a human, without any constraints on the formulation), which is then either translated automatically to a SPARQL query with the corresponding semantics, or it is decided that no such SPARQL query exists (because the data is not in the knowledge graph).

Our running example in this section is the following question and corresponding SPARQL query (referring to a simplified knowledge graph like the one from Sections 1.2.1 and 1.4.1, not Wikidata):

What role does Meryl Streep play in The Iron Lady?

```
SELECT ?target WHERE {
  <Meryl_Streep> <film_performance> ?m .
  ?m <film> <The_Iron_Lady> .
  <character> ?target
}
```

In this subsection, we explain the basic principles behind a typical system that solves this problem. The system is called *Aqqu* [Bast and Haussmann 2015], and a demo is available at https://aqqu.cs.uni-freiburg.de. Diefenbach et al. [2018] provide a broader overview of techniques and systems for question answering on knowledge graphs. We divide this task into the following four steps, which we will explain in detail below:

(1) Find entities from the knowledge graph that are mentioned in the question.
(2) Generate candidate SPARQL queries that contain these entities.
(3) Compute a *feature vector* for each candidate.
(4) Find the best candidate(s) by ranking the feature vectors.

Step 1: Find entities from the knowledge graph mentioned in the question
The input question typically refers to one or several entities from the knowledge graph (assuming that it can be answered from the knowledge graph at all). Since we don't have an interpretation of the question at this point, we identify *all* possible entity mentions (or at least so many, that we can be reasonably sure that the desired entities are among them) and the respective entity in the knowledge graph. This task is known as *entity linking*; more about this in Section 1.7.2. We assume that our entity linker does not only provide us with a single entity for each mention, but with a probability distribution over several entities; this will be used in Step 3.

Entity linking is hard because the same entity can be referred to by many different names, and the same piece of text can mean many different entities, depending on the context. For example, there are many persons called *Meryl*, and depending on the context, *Iron Lady* could refer to the movie or to Margaret Thatcher personally. If we assume that the question is well-formed, we can use a *part-of-speech (POS) tagger* to identify those parts of the question that could be entity mentions. For our example, a perfect POS tagger would identify "role", "Meryl Streep" and "Iron Lady". Without a POS tagger, any subsequence of words could be a potential entity match.

Unfortunately, this method does not work well enough with predicates. There is just much more variation in language in expressing predicates than in referring to entities, and the match between the verb(s) in the question and the predicate in the query can be extremely fuzzy. In our example, the words "role does ... play" (not even consecutive in the question) correspond to the predicates <*film_performance*> and <*character*> in the SPARQL query. But the question could also be formulated differently, for example, "Who is Meryl Streep in The Iron Lady?". We deal with the quality of the predicate matches in Step 3.

Step 2: Generating candidates
We generate SPARQL queries, in which at least one entity from Step 1 appears as a subject or object. Since there is a huge number of such queries, we only consider SPARQL queries of a particular form, called *template*. Aqqu uses three templates, which are shown in Figure 1.5. They correspond to relatively simple SPARQL queries and they are sufficient to answer most

simple questions. Let us explain how to generate query candidates from the third template. First, take any pair of entities e_1 and e_2 that were found in step (1) and whose mention text in the question does not overlap. For each pair, find all predicates r_1, r_2, r_3 and mediator entities *m* such that the triples (e_1 r_1 *m*), (*m* r_2 e_2) and (*m* r_3 *x*) exist in the knowledge graph, where *x* represents an arbitrary entity. Each assignment to e_1, e_2, r_1, r_2 and r_3 yields a query candidate. For example, in the third row in Figure 1.5, <Meryl_Streep> was assigned to e_1, <Margaret_Thatcher> to e_2 and <film_performance>, <character> and <film> were assigned to r_1, r_2 and r_3, respectively. This step generates many *bad* query candidates, in the sense that their result is not the correct answer to the question (the first three examples in Figure 1.5). However, the correct query should be among the candidates as well (last example in Figure 1.5). The next two steps address the problem of finding the correct query among the candidates.

Step 3: Computing feature vectors
For each of the query candidates from the last step, we generate a feature vector. Each of these features gives some indication about how likely this query candidate is to correctly answer the question. As an example, we will discuss three different features. A real system has more features; Aqqu for example has 23.

Entities (Ent): How well do the entities match the question? We make use of the probabilities that the named-entity linker from Step 1 provides. For each entity *e* that matched a part of the question *s*, compute the probability $\mathrm{p}\,(e|s)$ that *s* actually referred to the entity *e*, for example:

$$\begin{aligned}\text{Prob}(\text{<The\_Iron\_Lady\_(film)>} \,|\, \text{“The Iron Lady”}) &= 0.74\\ \text{Prob}(\text{<Margaret\_Thatcher>} \,|\, \text{“The Iron Lady”}) &= 0.17\\ \text{Prob}(\text{<Meryl\_Streep>} \,|\, \text{“Meryl Streep”}) &= 0.98\end{aligned}$$

For the final score, aggregate these probabilities for all entities that appear in a query. For our example in Figure 1.5 we added them up.

Predicates (Pred): How well do the predicates match? In a first approach, we would like to measure how similar the words in the question are to the words of the predicates in the query. We can do this by either considering literal matches or derivations, for example lemmatization (e.g., “played” $\rightarrow$ “play”) or synonyms (e.g., “role” and “character”). We compute a score, for example, by counting the number of matches or determining the percentage of predicate words covered in the query. However, as discussed in Step 1, this often does not work with predicates. We will now explain a more elaborate strategy for fuzzy predicate matching in more detail. Assume we want to find words that describe the relationship <Meryl_Streep> has to *?target* in the second example in Figure 1.5. In order to find such *indicator words*, we replace <Meryl_Streep> with a variable and run the following query

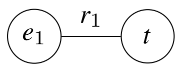
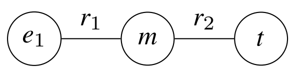
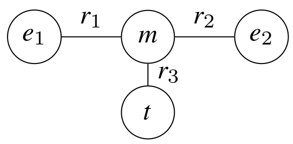
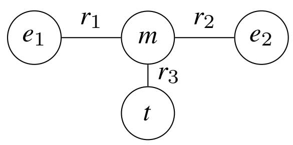
Example Candidate	**Template**	**Features**
<Meryl_Streep> <won_award> ?target		Ent: 0.98 Pred: 0 Cov: 0.2
<Meryl_Streep> <film_performance> ?m . ?m <character> ?target		Ent: 0.98 Pred: 1.3 Cov: 0.4
<Meryl_Streep> <film_performance> ?m . ?m <character> <Margaret_Thatcher> . ?m <film> ?target		Ent: 1.15 Pred: 1.3 Cov: 0.7
<Meryl_Streep> <film_performance> ?m . ?m <film> <The_Iron_Lady_(film)> . ?m <character> ?target		Ent: 1.72 Pred: 1.3 Cov: 0.7

Figure 1.5 Four examples of query candidates generated for the question in Figure **??**. The last row contains the query that correctly answers the question. Each template represents queries with a particular kind of structure. The nodes e_i represent entities found in Step 1 and r_i represent predicates that are connected with the entities as illustrated. The nodes m are intermediate nodes, representing a variable in the query candidates. The nodes t represent the *?target* variable, which contains the answer that this candidate gives to the question. The last column shows the feature vectors, which give some indication for how good the match between the candidate and the question is; see Step 3 for more information.

```
SELECT ?x ?target WHERE {
  ?x <film_performance> ?m .
  ?m <character> ?target
}
```

Among others, we get a result row where *?x* is <Leonardo_DiCaprio> and *?target* is <Billy_Costigan>. For each row in the result, look for all sentences (e.g., on Wikipedia) that contain both entities:

In Scorsese's The Departed, DiCaprio played the role of Billy Costigan.

Now, consider each verb or noun between the entity occurrences as an indicator word. In other words, we have learned that "play" or "role" are indicator words for the relationship

<film_performance> plus <character>. This is an instance of *distant supervision*, i.e. using external data to learn something useful for our task. Compute a score for how good an indicator word describes the relation, depending on how often it was found. Then, for each query candidate sum up the scores for all words from the question that match a relation. As a result, the last three examples in Figure 1.5 get a higher score for this feature than the first one (since no indicator words of <won_award> appear in the question).

Coverage (Cov): How much of the question is covered by the query? Generally, the more parts of the question are matched to entities or predicates in a query candidate, the better the candidate is. Hence, we compute the percentage of words covered by dividing the number of words that are matched by the total number of words in the question. For example, the third candidate in Figure 1.5 covers 7 out of 10 words:

What <u>role</u> does <u>Meryl</u> <u>Streep</u> <u>play</u> in <u>The</u> <u>Iron</u> <u>Lady</u>?

Note that we could also give weights to different words, if we are able to identify important parts of a question. In our example, the words not covered ("what", "does" and "in") are not very important since they do not carry a lot of meaning. It is therefore not important to cover them and we could assign lower weights to them.

The scores in the feature vectors can be *misleading* in the sense that the correct query does not get the best score. For example, questions can contain additional information that should not be matched in the query. Consider the question "What role does Meryl Streep play in the British drama film The Iron Lady?". The part "British drama film" is irrelevant and queries that match this part are probably wrong.

Step 4: Ranking

There is usually no candidate with a feature vector that is better than all other feature vectors in all components. Therefore, we need an algorithm to determine the "best" feature vector.

A simple approach would be to turn each feature vector into a real-values score by taking a linear combination of its components. For example:

$$s(c) = 5 \cdot c_{\text{Ent}} + 3 \cdot c_{\text{Pred}} + 4.5 \cdot c_{\text{Cov}},$$

where c is a query candidate and c_x is the score that c got for feature x. With good weights, this approach can work reasonably well, but finding good weights manually is hard, especially when there are many features.

More sophisticated approaches try to *learn* to determine the best feature vector among a given set. To learn scores, we need training samples of the kind *feature vector* $\rightarrow$ *score*. Note that we can generate them from training samples of the kind *question* $\rightarrow$ *answer* in the following way. Using the previous steps, for each question in the ground truth, we generate query candidates and the feature vector for each candidate. Mark a feature vector as positive sample, if the corresponding query candidate returns the correct answer according to the ground truth and mark all other feature vectors as negative samples. The problem with this approach is that the same feature vector can be good for one (hard) question and bad for another (easy) question. To address this issue, we transform the problem into a binary classification problem and *learn to compare* two feature vectors. We generate pairs of one "positive" and one "negative" feature vector and we mark which one is better. In the QA system, for all pairs of feature vectors, we let our trained classifier decide, which one is better and pick the candidate that *wins* most of these comparisons. There are two variants of Aqqu with respect to this classifier: one uses random forests and one uses a neural network. For details, see Bast and Haussmann [2015].

1.7 Combination with text search and federated search

There are several scenarios, where we want to extend a SPARQL query by a text-search component. We briefly explore these in Sections 1.7.1 (text search in literals) and 1.7.2 (text search in an an external text corpus). Another frequent scenario is that we want to query multiple RDF datasets simultaneously with a single query. We briefly explore this in Section 1.7.3.

1.7.1 Keyword search in literals

As we have seen in Section 1.2, the object of a triple can also be a string containing arbitrary text. A typical use is for the name, alias or description of an entity; see Section 1.2.3. The SPARQL query language supports a function *REGEX* that determines whether a given string matches a given regular expression. For example, the following query finds all people on Wikidata, whose name ends with *Einstein*:

```
SELECT ?person_name WHERE {
  ?person    wdt:P31      wd:Q5 .                # instance of human
  ?person    rdfs:label   ?person_name .         # name
  FILTER (LANG(?person_name) = "en")             # English
  FILTER REGEX(?person_name, " Einstein$")       # ends with " Einstein"
}
```

Click to run query on QLever

Literals can contain arbitrarily long text. For example, Wikidata contains an entity for each Wikipedia article, and we might use the predicate *schema:description* to associate each article with its abstract.[34] For a text search in such literals, keyword search is often the method of choice. That is, we specify a number of keywords, and we expect the engine to match all literals containing these keywords (in any order). In principle, keyword search can be simulated in SPARQL by introducing one *FILTER REGEX* clause per keyword. For example, the following

[34] The abstract of a Wikipedia article is the part before the table of contents. We might also take the whole text, but *schema:description* is commonly used for short descriptions.

query finds all astronauts, where the Wikipedia abstract mentions the word *moon* and a word starting with *walk*.[35]

```
SELECT ?astronaut_label ?abstract WHERE {
  ?astronaut   wdt:P106            wd:Q11631 .         # occupation astronaut
  ?astronaut   rdfs:label          ?astronaut_label .  # name
  ?wikipedia   schema:about        ?astronaut .        # Wikipedia article
  ?wikipedia   schema:description  ?abstract .         # Wikipedia abstract
  FILTER (LANG(?astronaut_label) = "en")               # English name
  FILTER REGEX(?abstract, "\\bmoon\\b", "i")           # contains word "moon"
  FILTER REGEX(?abstract, "\\bwalk", "i")              # contains word starting with "walk"
}
```

Click to run query on QLever

The query above is not only cumbersome to write down, but also not processed efficiently in most SPARQL engines. The reason is that the two REGEXes will be checked for every *?abstract* that matches the rest of the query. In particular, this requires the materialization of the respective strings. A more efficient approach would be to build a dedicated data structure that supports keyword search in literals, like an inverted index. Engines like QLever and Virtuoso indeed provide this functionality. For example, to process the query above more efficiently (with exactly the same result) in QLever, one can replace the last two FILTER lines as follows.[36]

```
SELECT ?astronaut_label ?abstract WHERE {
  ...                                          # first five lines like above
  ?abstract ql:match-keywords "moon walk*"     # match given keywords
}
```

Click to run query on QLever

1.7.2 Search in an external text corpus linked to a knowledge graph

The example query above constrained a certain set of entities (astronauts) by information contained in text (the Wikipedia abstracts). This worked because the relevant texts were connected to the relevant entities via a predicate (*schema:description*).

But this is the exception: entity descriptions are typically very short and contain only the most notable information. In a more general scenario, we are given an abitrary text corpus that mentions entities from the knowledge graph. To identify these entity mentions, we need

[35] The \b in a regular expression matches a word boundary. For example, \bmoon\b matches a string that contains the word *moon*, but does not match a string that contains the words *honeymoon* or *moons*, but not *moon*. We would therefore expect the query to match all astronauts who actually walked on the moon. Backslashes have to be escaped in SPARQL, so we need to use \\b in the query. The argument *"i"* stands for case-insensitive search.

[36] Virtuoso achieves the same functionality using the special *bif:contains* predicate.

to perform a task called *entity linking* (which we have already used for question answering in Section 1.6.2).[37] For example, assume that our text corpus contains the following sentence from Section 1.2.5.

As Armstrong [Q1615], whose great-grandfather Friedrich Wilhelm Kötter emigrated from Ladbergen [Q181877] to the United States [Q30] in 1864, was the first man to walk on the moon [Q405] in 1969, many citizens of Ladbergen [Q181877] became interested in their American [Q846570] relatives.

In such a scenario, we want to formulate queries that specify the co-occurrence of a certain kind of entity with certain words. For example, the following query asks the same questions as in the previous subsection (astronauts who walked on the moon), but this time without assuming that the information is contained in the description of the respective entities. As an additional asset, the astronauts can now be ranked by how often they occur with the specified words.[38]

```
SELECT ?astronaut_name (COUNT(?text) AS ?count) WHERE {
  ?astronaut   wdt:P106            wd:Q11631 .          # occupation astronaut
  ?astronaut   rdfs:label          ?astronaut_name .    # name
  FILTER (LANG(?astronaut_name) = ”en”)                 # English name
  ?astronaut   ql:mentioned-in     ?text .              # entity mention in text ...
  ?text        ql:match-keywords   ”moon walk*” .       # ... that matches keywords
}
GROUP BY ?astronaut_name ORDER BY DESC(?count)
```

Click to run query on QLever

The predicate *ql:mentioned-in* is a so-called *magic predicate*. It links each entity to all texts mentioning that entity. Note that this predicate represents a many-to-many relation: a single entity may be (any typically is) mentioned in multiple texts and a single text may contain multiple entities. A simple realization of a magic predicate is to explictly add it to the RDF dataset. The performance of this approach depends on the ability of the query engine to perform join operations with this (typically huge) predicate. For example, the English Wikipedia (which is a text corpus of moderate size compared to, say, a web corpus), contains around one billion mentions of entities from Wikidata. QLever provides a more efficient realization that avoids this explicit materialization but requires that the usage of *ql:mentioned-in* is always combined with a keyword filter.[39] For more information on this, see Bast and Buchhold [2017].

Note that in the query above, as well as in the query from the previous subsection, it is implicitly assumed that entities or words that occur together in a text, also “belong together semantically”. But that is not necessarily the case, especially for longer texts or complex

[37] For a recent overview paper with a list of the top-performing tools, see [Bast et al. 2022a].

[38] We could also specify a more complex ranking function if we wanted to.

[39] Especially when the texts are sentences, a single text typically mentions only few (often only one) entity. This information can then be stored in the inverted index lists.

sentences. For example, in the sentence above, it is Neil Armstrong who walked on the moon and his great-grandfather who emigrated from Germany, and not vice versa. This problem can be addressed by splitting the text into its semantic constituents; see [Bast and Haussmann 2013].

1.7.3 Federated search

Interoperability is at the heart of the RDF framework. Because each dataset is simply a set of triples and entities have globally unique identifiers, we can simply combine two datasets by combining the sets of triples. Alternatively, SPARQL allows the querying of multiple datasets via the SERVICE keyword (for a discussion of scenarios where this approach is useful see the following section). For example, the following query asks for all movies directed by the Coen brothers (information contained in Wikidata), ranked by the number of votes in IMDb (information contained in the IMDb dataset).

```
SELECT ?movie ?imdb_votes WHERE {
  ?movie    wdt:P31      wd:Q11424 .        # instance of film
  ?movie    wdt:P345     ?imdb_id .         # IMDb ID
  ?movie    wdt:P57      wd:Q13595311 .     # directed by Joel Coen
  SERVICE <https://qlever.cs.uni-freiburg.de/api/imdb> {
    ?movie_imdb  imdb:id  ?imdb_id .        # IMDb entity with ?imdb_id
    ?movie_imdb  imdb:numVotes  ?imdb_votes .  # number of votes
  }
}
ORDER BY DESC(?imdb_votes)
```

Click to run query on QLever

The semantics of the SERVICE clause is easy to understand and fits very naturally into the RDF framework. In the query above, the body of the SERVICE clause is turned to a SPARQL query by adding a *SELECT * WHERE { … }* around it. That query is then sent to the SPARQL endpoint defined right after the SERVICE keyword. The result from that query is a table (or binding), just like the result of any graph pattern in a SPARQL query is a table (or binding).

A conceptually trivial implementation of SERVICE is to send the query to the remote endpoint, receive the fully materialized result (in a typical serialization format like JSON), and let the issuing SPARQL engine parse that result for further processing. This is correct, but can be inefficient when that materialized result is moderately large and becomes infeasible when the result is huge.

We briefly describe two possible optimizations. One optimization[40] is to send the remote endpoint additional information that helps it to reduce the results size. For example, in the

[40] This optimization is so obvious that it is already mentioned in the standard: https://www.w3.org/TR/sparql11-federated-query#values.

query above, the set of movies is restricted to those by a particular director. The SPARQL engine could therefore send the remote endpoint the respective bindings for *?imdb_id* (19 movies by Joel Coen), which would result in a much smaller result of the SERVICE query. Another optimization is to both send and receive bindings not in materialized form, but in binary form, for example, using the internal IDs mentioned in Section 1.5. This is challenging to implement, however, because internal IDs from different SPARQL engines are not naturally compatible so that we need an efficient translation scheme.

1.7.4 Use cases of federated search

We briefly discuss three use cases of federated search, which demonstrate the power of this paradigm.

The first and most typical use case is to query multiple RDF datasets in a single query. The query above gives an example for that. Note that a SPARQL query may issue SERVICE queries to an arbitrary number of remote endpoints, and these queries may even be nested (that is, a SERVICE query may itself contain another SERVICE query). Also note that the remote endpoint may be a variable. That way, we can select the appropriate endpoint via a SPARQL query or part of SPARQL query.

A second use case is to split a large datasets into multiple datasets. One reason for this could be that smaller datasets can be queried more efficiently. Another reason could be that we may have one large part of the dataset that is static (building an index for a large RDF dataset is expensive, see Section 1.5), and one or several smaller parts that are dynamic (so that new indexes can be built for them fast).

A third use case is that a remote endpoint must obey the SPARQL API, but depending on the service provided it must not necessarily be a full-featured SPARQL engine, but may use a special-purpose implementation. For example, the *Wikidata Query Service (WDQS)*[41] has a SERVICE that provides labels for entities from a SPARQL query. Using this SERVICE is more efficient than explicitly using the *rdfs:label* predicate on the Blazegraph query engine (which was the WDQS backend at the time of this writing), especially when many labels have to be retrieved.[42] It is, however, relatively straightforward using a special-purpose data structure.

1.8 The future of knowledge graphs

In this chapter, we have gently introduced the basics of RDF and SPARQL, we got acquainted with some of the most important publicly available knowledge graphs and their characteristics, we have learned how to index and search a knowledge graph efficiently, and we have seen some basic techniques for how to assist users in formulating their SPARQL queries.

[41] https://query.wikidata.org/

[42] At the time of this writing, the *rdfs:label* predicate contained 810M triples in Wikidata.

There are many other important questions around knowledge graphs that we haven't even touched upon: How to construct a knowledge graph automatically or semi-automatically? How to fill gaps in an existing knowledge graph or fix mistakes or resolve inconsistencies? How to resolve conflicts and fuse results when querying multiple knowledge graphs? How to reason on knowledge graphs? These aspects are discussed in the recent and very extensive survey by Hogan et al. [2021]. That survey also lists (in its Table 1) many surveys on more specialized subtopics.

Before we close, let us dare take a short glimpse into the future. The last decade has seen tremendous advances in the field of *deep learning*, with revolutionary results across a wide variety of application domains. Notably, *large language models (LLMs)* have recently achieved state-of-the-art results and better on a wide variety of tasks with little (*few-shot*) or no (*zero-shot*) task-specific learning [OpenAI 2023]. In particular, such LLMs can automatically translate natural language questions to SPARQL queries, as explained in Section 1.6.2 using an explicit approach. A central question in this context is how classical frameworks like knowledge graphs (or, more generally, databases) will co-exist with such universal models in the future. We see three possible scenarios.

1. Knowledge graphs and universal models will co-evolve, but remain separate "species". The models will learn better and better how to formulate even complex queries, but they will resort to an external query engine (like the ones described in Section 1.5) for executing these queries. The result can then be fed back into the "thinking process" of the model and this interaction may already be part of the training. As this mode of operation becomes more prevalent, query engines will adapt to suit the needs of these models better.[43] We consider this the most likely scenario in the near and mid future.

2. In a more dim scenario, knowledge graphs will become mere food (training data) for universal models. Given enough data, the models might learn by themselves how to answer even complex queries without having to resort to an external query engine. We consider this scenario unlikely for two reasons. First, it looks hard for a learned model to answer complex queries involving large datasets with 100% accuracy (a task that for a classical query engine is merely a matter of a correct implementation). Second, resourcefulness is a major factor for large datasets. For example, consider a business application that needs to store billions or trillions of data records. Why let a model figure out how to store and query this data precisely and compactly instead of using standard compression and indexing techniques directly? We therefore don't consider this scenario very likely.

3. The third scenario is that of hybrid systems that somehow combine universal models and classical storage and indexing techniques. At the time of this writing, there is no noteworthy development in this direction. We consider such development unlikely because of the following

[43] For example, embeddings of entities and predicates might become first-class citizens in the query engine.

high-level argument: The evolution of all complex organizations we know of (including nature and human societies) has brought forth different kind of subsystems with specialized capabilities. Interaction between these subsystems is extremely important (and may range from slim interfaces to symbiotic relationships), but the subsystems remain separately identifiable species. The universe we live in does not seem to favor hybrid monsters. Then again, there is the Borg[44] and we are still at a very early stage in our evolution.

Whatever will happen, we will soon be able to discuss these fascinating topics in depth and in person with a universal model of our choice … if they are still talking to us then.

[44] ”Your biological and technological distinctiveness will be added to our own. Resistance is futile. You will be assimilated.”

Bibliography

I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. 2017. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proc. VLDB Endow.*, 10(13): 2049–2060.

K. Alaoui. 2019. A categorization of RDF triplestores. In *SCA*, pp. 66:1–66:7. ACM.

W. Ali, M. Saleem, B. Yao, A. Hogan, and A.-C. N. Ngomo. 2021. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *Computing Research Repository*, abs/2102.13027.

R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5): 68:1–68:40.

H. Bast and B. Buchhold. 2017. QLever: a query engine for efficient SPARQL+Text search. In *Proc. 26th ACM Int. Conf. on Information and Knowledge Management*, pp. 647–656. ACM.

H. Bast and E. Haussmann. 2013. Open information extraction via contextual sentence decomposition. In *ICSC*, pp. 154–159. IEEE Computer Society.

H. Bast and E. Haussmann. 2015. More accurate question answering on Freebase. In *Proc. 24th ACM Int. Conf. on Information and Knowledge Management*, pp. 1431–1440. ACM.

H. Bast, F. Bäurle, B. Buchhold, and E. Haußmann. 2014. Easy access to the freebase dataset. In *Proc. 26th Int. World Wide Web Conf. (Companion Volume)*, pp. 95–98. ACM.

H. Bast, M. Hertel, and N. Prange. 2022a. ELEVANT: A fully automatic fine-grained entity linking evaluation and analysis tool. In *EMNLP (Demos)*, pp. 72–79. Association for Computational Linguistics.

H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle. 2022b. Efficient and effective SPARQL autocompletion on very large knowledge graphs. In *CIKM*, pp. 2893–2902. ACM.

K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 1247–1250. ACM. https://doi.org/10.1145/1376616.1376746.

T. Chawla, G. Singh, E. S. Pilli, and M. C. Govil. 2020. Storage, partitioning, indexing and retrieval in big RDF frameworks: A survey. *Comput. Sci. Rev.*, 38: 100309.

A. Chebotko, S. Lu, and F. Fotouhi. 2009. Semantics preserving sparql-to-sql translation. *Data & Knowl. Eng.*, 68(10): 973–1000.

D. Diefenbach, V. López, K. D. Singh, and P. Maret. 2018. Core techniques of question answering systems over knowledge bases: a survey. *Knowl. and Information Syst.*, 55(3): 529–569.

O. Erling and I. Mikhailov. 2009. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pp. 501–519. Springer. https://doi.org/10.1007/978-3-642-04329-1_21.

D. C. Faye, O. Curé, and G. Blin. 2012. A survey of RDF storage approaches. *ARIMA J.*, 15: 2.

G. Fu, C. Batchelor, M. Dumontier, J. Hastings, E. Willighagen, and E. Bolton. 2015. PubChemRDF: towards the semantic annotation of PubChem compound and substance databases. *J. Cheminformatics*, 7: 34. https://dx.doi.org/10.1186/s13321-015-0084-4.

A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A.-C. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. 2021. Knowledge graphs. *Computing Research Repository*, abs/2003.02320.

A. Ismayilov, D. Kontokostas, S. Auer, J. Lehmann, and S. Hellmann. 2018. Wikidata through the eyes of DBpedia. *Semantic Web*, 9(4): 493–503.

D. Janke and S. Staab. 2018. Storing and querying semantic data in the cloud. In *Reasoning Web*, volume 11078 of *Lecture Notes in Computer Science*, pp. 173–222. Springer.

Z. Kaoudi and I. Manolescu. 2015. RDF in the clouds: a survey. *VLDB J.*, 24(1): 67–91.

J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. 2015. DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2): 167–195. https://doi.org/10.3233/SW-140134.

D. B. Lenat. 1995. CYC: A large-scale investment in knowledge infrastructure. *Commun. ACM*, 38(11): 32–38.

M. Ley. 2009. DBLP - some lessons learned. *Proc. VLDB Endowment*, 2(2): 1493–1500.

Y. Luo, F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. 2012. Storing and indexing massive RDF datasets. In *Semantic Search over the Web*, Data-Centric Systems and Applications, pp. 31–60. Springer.

Z. Ma, M. A. M. Capretz, and L. Yan. 2016. Storing massive resource description framework (RDF) data: a survey. *Knowl. Eng. Rev.*, 31(4): 391–413.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

OpenStreetMap contributors, 2021. OpenStreetMap. https://www.openstreetmap.org.

M. T. Özsu. 2016. A survey of RDF data management systems. *Frontiers Comput. Sci.*, 10(3): 418–432.

Z. Pan, T. Zhu, H. Liu, and H. Ning. 2018. A survey of RDF management technologies and benchmark datasets. *J. Ambient Intell. Humaniz. Comput.*, 9(5): 1693–1704.

S. Sakr and G. Al-Naymat. 2009. Relational processing of RDF queries: a survey. *SIGMOD Rec.*, 38(4): 23–28.

L. H. Z. Santana and R. dos Santos Mello. 2020. An analysis of mapping strategies for storing RDF data into nosql databases. In *SAC*, pp. 386–392. ACM.

A. Singhal, 2012. Introducing the knowledge graph: things, not strings. https://www.blog.google/products/search/introducing-knowledge-graph-things-not/. Accessed: 2021-02-15.

F. M. Suchanek, G. Kasneci, and G. Weikum. 2008. YAGO: a large ontology from Wikipedia and WordNet. *J. Web Semantics*, 6(3): 203–217.

M. Svoboda and I. Mlýnková. 2011. Linked data indexing methods: A survey. In *OTM Workshops*, volume 7046 of *Lecture Notes in Computer Science*, pp. 474–483. Springer.

Systap, 2013. The bigdata RDF database. https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf, retrieved 27.02.2021.

T. P. Tanon, D. Vrandecic, S. Schaffert, T. Steiner, and L. Pintscher. 2016. From Freebase to Wikidata: the great migration. In *Proc. 25th Int. World Wide Web Conf.*, pp. 1419–1428. ACM. https://doi.org/10.1145/2872427.2874809.

T. P. Tanon, G. Weikum, and F. M. Suchanek. 2020. YAGO 4: a reason-able knowledge base. In *Proc. 17th Extended Semantic Web Conf.*, pp. 583–596. Springer. https://doi.org/10.1007/978-3-030-49461-2_34.

The UniProt Consortium. 2017. UniProt: the universal protein knowledgebase. *Nucleic Acids Res.*, 45(Database-Issue): D158–D169. https://doi.org/10.1093/nar/gkw1099.

D. Vrandecic and M. Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10): 78–85.

M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr. 2018. RDF data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4): 84:1–84:36.

M. Q. Yasin, X. Zhang, R. Haq, Z. Feng, and S. Yitagesu. 2018. A comprehensive study for essentiality of graph based distributed SPARQL query processing. In *DASFAA Workshops*, volume 10829 of *Lecture Notes in Computer Science*, pp. 156–170. Springer.