Querying Wikidata: Comparing SPARQL, Relational and Graph Databases

Daniel Hernández[1], Aidan Hogan[1],
Cristian Riveros[2], Carlos Rojas[2], and Enzo Zerega[2]

Center for Semantic Web Research
[1] Department of Computer Science, Universidad de Chile
[2] Department of Computer Science, Pontificia Universidad Católica de Chile

Resource type: Benchmark and Empirical Study
Permanent URL: https://dx.doi.org/10.6084/m9.figshare.3219217

Abstract: In this paper, we experimentally compare the efficiency of various database engines for the purposes of querying the Wikidata knowledge-base, which can be conceptualised as a directed edge-labelled graph where edges can be annotated with meta-information called qualifiers. We take two popular SPARQL databases (Virtuoso, Blazegraph), a popular relational database (PostgreSQL), and a popular graph database (Neo4J) for comparison and discuss various options as to how Wikidata can be represented in the models of each engine. We design a set of experiments to test the relative query performance of these representations in the context of their respective engines. We first execute a large set of atomic lookups to establish a baseline performance for each test setting, and subsequently perform experiments on instances of more complex graph patterns based on real-world examples. We conclude with a summary of the strengths and limitations of the engines observed.

1 Introduction

Wikidata is a new knowledge-base overseen by the Wikimedia foundation and collaboratively edited by a community of thousands of users [21]. The goal of Wikidata is to provide a common interoperable source of factual information for Wikimedia projects, foremost of which is Wikipedia. Currently on Wikipedia, articles that list entities – such as top scoring football players – and the infoboxes that appear on the top-right-hand side of articles – such as to state the number of goals scored by a football player – must be manually maintained. As a result, for example, factual information in different locations will often be inconsistent. The aim of Wikidata is to instead keep such factual knowledge in one place: facts need be edited only once and can be drawn upon from multiple locations. Since the launch of Wikidata in October 2012, more than 80 thousand editors have contributed over 86 million statements about 17 million entities.

To allow users issue bespoke queries over the knowledge-base, Wikimedia has begun hosting an official query service[3] which according to internal statistics[4] receives in the order of hundreds of thousands of queries per day. The query

[3] https://query.wikidata.org/
[4] https://grafana.wikimedia.org/dashboard/db/wikidata-query-service

service runs over an RDF representation of Wikidata that is indexed in the Blazegraph SPARQL store (formerly known as BigData) [20].

In Wikidata, items are connected either to related items or datatype values by directed, named relations. There is thus a close correspondence between Wikidata and RDF. However, relations in Wikidata can be annotated with attribute–value pairs, such as qualifiers and references, to specify a further context for the relation (e.g., validity, cause, degree, etc.). This complicates the representation of Wikidata in RDF somewhat, requiring some form of *reification* [11] to capture the full knowledge-base in RDF. In previous work [12], we thus investigated various ways in which Wikidata can be represented in RDF and how that affects the query performance in various SPARQL engines; more specifically we tested n-ary relation, standard reification, singleton property and named graph representations of Wikidata against five SPARQL stores – 4store [7], Blazegraph (formerly BigData) [20], GraphDB (formerly (Big)OWLIM) [2], Jena TDB [22], and Virtuoso [4] – with respect to answering an initial selection of 14 real-world queries. Our results suggested that while engines struggled with the singleton property representation, no other representation was an outright winner. In terms of engines, we found that Virtuoso exhibited the best and most reliable performance, with GraphDB and Blazegraph following behind.

A follow-up question arising from our initial previous work was how the performance of these SPARQL engines would compare with that of other technologies. In this paper, we thus extend on our previous work by comparing a selection of SPARQL, relational and graph databases for the purposes of querying Wikidata. We select these families of databases since they correspond with the inherent structure of Wikidata *and* they offer support for comprehensively-featured declarative query languages as needed for the public query service (which rules out, for example, key–value stores, document stores and column-family stores). To conduct these comparisons, we design and apply a range of novel experiments.

However, there are still too many databases engines within these three families for all to be considered experimentally in the current scope; hence we must be selective in what we test. For SPARQL, we select Virtuoso [4], which performed best overall in our previous experiments, and Blazegraph [20], which is currently deployed in Wikimedia's official query service. For graph databases, we select Neo4J [19], which is based on *property graphs*: not only is it arguably the most popular graph database,[5] it is the only (non-SPARQL) graph database we know of that supports a mature query language (Cypher) in the declarative style of the example queries listed for the official query service. For relational databases, we select PostgreSQL [18] as a mature open-source solution supporting the SQL standard. Although we could consider other databases for testing – such as various relational databases or other SPARQL engines such as Allegrograph [14] or Stardog [13] – our selection allows us to draw initial conclusions with respect to how well prominent databases from each family of technologies perform relative to each other, and indeed how the solution selected by Wikimedia (Blazegraph) compares with these other alternatives.

[5] We can refer (informally) to, e.g., `http://db-engines.com/en/ranking/graph+dbms`

Towards testing these diverse engines, we first discuss some representations for encoding the Wikidata knowledge-base such that it can be loaded into each type of engine. We then introduce various novel experiments. Our first set of experiments that are based on the idea of performing sets of lookups for "atomic patterns" with exhaustive combinations of constants and variables; these results give an initial idea of the low-level performance of each configuration. Next we perform experiments on sets of instances of basic graph patterns that generalise the graph motifs we found to commonly occur in the use-case queries listed at the public query service. We also discuss how well the various engines support the query features commonly used for these use-case queries.

Before we continue, however, we first introduce Wikidata in more detail.

2 The Wikidata Model

In Figure 1, we see an example Wikidata statement describing the U.S. presidency of Abraham Lincoln [12]. Internal identifiers are in grey: those beginning with Q refer to entities, and those referring to P refer to properties. These identifiers map to IRIs, where information about that entity or relationship can be found on Wikidata. Entities and relationships are also associated with labels, where the example shows labels in English. The statement contains a *primary relation* with Abraham Lincoln as *subject*, position held as *predicate*, and President of the United States of America as *object*; this relation is associated with pairs of *qualifier predicates* (e.g., start time, follows) and their *qualifier values* (e.g., "4 March 1861", James Buchanan); we call each such pair a *qualifier*. Statements are also often associated with one or more *references* that support the claims and with a *rank* that marks the most important statements for a given property, but herein we can treat references and ranks as special types of qualifiers. As such, we can define a Wikidata statement as a primary relation and a set of qualifiers.

At its core, Wikidata then consists of a set of such statements and a mapping from identifiers to labels in various languages. However, some statements may not contain any qualifiers (though in theory all statements should contain a reference, the knowledge-base is, by its nature, incomplete). Datatype values can themselves be associated with meta-data, where for example dates or times can be associated with a specific calendar. Likewise, objects can occasionally be existential (for example, when someone is known to have been murdered but their killer is unknown) or non-existential (to explicitly state that an entity has no such relation); however, these types of values are quite rare and hence for brevity we do not consider them directly.[6] Finally, it is important to note that Wikidata can contain multiple distinct statements with the same binary relation: for example, Clover Cleveland was the U.S. President for two non-consecutive terms (i.e., with different start and end times, different predecessors and successors). In Wikidata, this is represented as two separate statements whose primary relations are both identical, but where the qualifiers (start time, end time, follows, followed by) differ.

[6] See `https://tools.wmflabs.org/wikidata-todo/stats.php`

Abraham Lincoln [Q91]

position held [P39]	President of the United States of America [Q11696]
start time [P580]	"4 March 1861"
end time [P582]	"15 April 1865"
follows [P155]	James Buchanan [Q12325]
followed by [P156]	Andrew Johnson [Q8612]

Fig. 1: A Wikidata statement about Abraham Lincoln (reused from [12])

3 Wikidata Representations

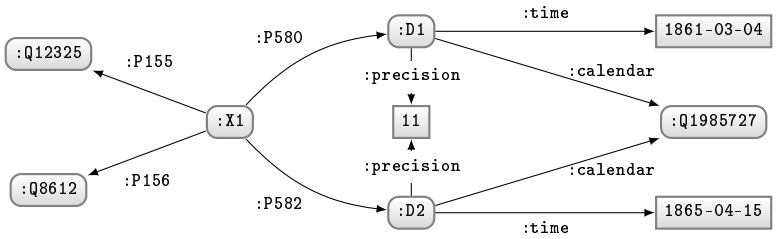
In the context of querying Wikidata, our goal is to compare the performance of a selection of database *engines* from three different *families*: SPARQL, relational and graph. Each family is associated with a different *data model*: named (RDF) graphs, relations, and property graphs, respectively. Each data model may permit multiple possible *representations* of Wikidata, for example, different RDF reification schemes, different relational schema, etc. Although the representations we present (aside from corner cases [12]) encode the same data and do not change the computational complexity of query evaluation – since, loosely speaking, translating between the representations is cheaper than query evaluation – as we will see, different representations carry different performance costs, particularly for more low-level atomic lookups. We now discuss different representations of Wikidata suitable for each of the three data models we consider.

3.1 RDF/Named Graph Representations

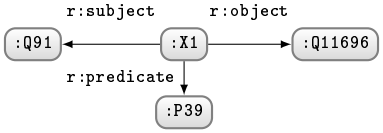
With respect to representing Wikidata for querying in SPARQL, in our previous work, we considered four well-known reification schemes [12].[7] Each such scheme associates the primary relation of a statement with a statement identifier onto which qualifiers can be associated. In Figure 2a, we give an example RDF graph where `:X1` is the statement identifier and its four outgoing edges represent the four qualifiers from Figure 1; we also show how complex datatypes can be represented. For each of the following representations, all we are left to do is associate the primary relation of Figure 1 – namely $(\mathsf{Q91}, \mathsf{P39}, \mathsf{Q11696})$ – with the statement identifier `:X1` already associated with the qualifier information; in other words, we wish to encode a quad (s, p, o, i) where (s, p, o) is the primary relation of the statement and i is an identifier for the statement. We again recap these four schemes, where more details are available from our previous paper [12].

Standard Reification (SR) [11]: an RDF resource is used to denote a triple. Using this scheme, we can use a reified triple to encode a Wikidata statement, as depicted in Figure 2b. We can encode n quadruples with $3n$ triples.

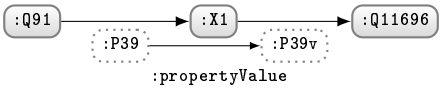
[7] We refer here to "reification" in the general sense of describing triples, whereas we refer to the specific proposal in the RDF specifications as "standard reification" [11].



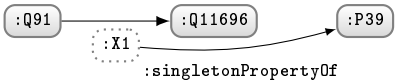
(a) Qualifier information common to all formats



(b) Standard reification



(c) n-ary relations



(d) Singleton properties

(e) Named graphs

Fig. 2: RDF representations encoding data from Figure 1 (adapted from [12])

n-ary Relation (NR) [5]: an RDF resource is used to identify a relation. Figure 2c shows such a scheme analogous to that proposed by Erxleben et al. [5] for representing Wikidata (modified slightly to reflect current data). The `:propertyValue` edge is important to explicitly link the original property with its twin. To represent n quads with m unique properties, we require $2n + m$ triples.

Singleton Properties (SP) [16]: a unique property represents each statement. The idea is captured in Figure 2d. To encode n quads, we need $2n$ triples.

Named Graphs [8] (NG): this is a set of pairs of the form (G, i) where G is an RDF graph and i is an IRI (which can be omitted in the case of the *default graph*). We can "flatten" this representation by taking the union over $G \times \{i\}$ for each such pair, resulting directly in quads, as illustrated in Figure 2e.

3.2 Relational representations

Figure 3 exemplifies the relational representation we use for Wikidata, which involves three tables: **Statement** stores the primary relation and a statement id; **Qualifier** associates one or more qualifiers with a statement id; and **Label** associates each entity and property with one or more multilingual labels. We

Statement

id	s	p	o	o_{date}
X1	Q91	P39	Q11696	⊥

Qualifier

q	v	v_{date}	id
P155	Q12325	⊥	X1
P156	Q8612	⊥	X1
P580	⊥	1861-03-04	X1
P582	⊥	1865-04-15	X1

Label

e	label	lang
P39	position held	en
...	...	...
Q91	Abraham Lincoln	en
...	...	...

Fig. 3: Relational representation of data from Figure 1

keep the **Label** table separate from **Statement** since we wish to keep a **lang** column for lookup/filtering, where labels are not qualified in Wikidata.

One complication with the relational model is that certain values – in particular the object (**o**) and qualifier value (**v**) – can have multiple types. Aside from entities, properties and logical values (e.g., exists, not exists), Wikidata contains four data-types – numeric, time, coordinates and strings – where as previously mentioned, each datatype can contain meta-information such as a calendar, precision, etc. One option is to store all values in one column as strings (e.g., JSON objects); however, this precludes the possibility of using such datatype values in filters, doing range queries, ordering, etc. Another possibility is to create separate columns, such as exemplified in Figure 3 with $\mathbf{o_{date}}$ and $\mathbf{v_{date}}$; however, we would need at least five such columns to support all Wikidata datatypes, leading to a lot of nulls in the table, and we still cannot store the meta-information. A third option would be to create separate tables for different datatype values, but this increases the complexity of joins. We currently use JSON strings to serialise datatype values, along with their meta-information, and store different types in different columns (i.e., as per Figure 3 with, e.g., $\mathbf{o_{date}}$ and $\mathbf{v_{date}}$).

3.3 Property graph representations

Graph databases commonly model data as *property graphs* [9]: directed edge-labelled graphs where nodes and edges are associated with ids, where each node and edge id can be associated with a *type* and a set of *attributes* that provide additional meta-information.[8] In Figure 4, we show how the Wikidata statement in Figure 1 could be represented as a "direct" property graph. In this case, `Q91` and `Q11696` are *node ids*, `X1` is an *edge id*, `P39` is an *edge type*, pairs of the form `label_en=*` are *node attributes*, and pairs of the form `P*=*` are *edge attributes*. Though not used in this case, nodes can also have *node types*.

One may argue that property graphs offer a natural fit for Wikidata, where we first tried to represent Wikidata in Neo4J analogous to the "direct representation" given in Figure 4, but we encountered a number of fundamental issues. Firstly, Neo4J only allows one value per attribute property: this could be circumvented by using list values or special predicates such as `label_en`. Secondly,

[8] Types are sometimes called "labels" and attributes are often called "properties" [9]. We avoid such terms, which clash with entity labels, RDF properties, etc.

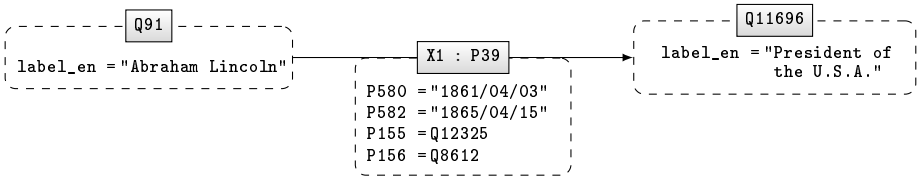


Fig. 4: A direct property graph encoding data from Figure 1

and more generally, Neo4J does not currently support queries involving joins or lookups on any information associated with edges, including edge ids, edge types or edge attributes. For example, one could not query for the president who followed (`P155`) James Buchanan (`Q12325`) without checking through all edges in the graph; this also means that we could not retrieve the labels of properties from such statements.[9] Hence, after initial experiments revealed these limitations for the direct representation, we sought another option.

Ultimately we chose to use a reified property graph representation to encode Wikidata in Neo4J, as depicted in Figure 5. Conceptually the idea is similar to standard reification for RDF, where we use a node to represent the statement and use fixed edges to connect to the subject, predicate and object of the primary relation. We can likewise connect to a qualifier object, which in turn points to a qualifier predicate and value (which can be an item or a datatype). Most importantly, this model avoids putting any domain terms (e.g, `P39`, `P580`, `Q8612`, etc.) on edges, which avoids the aforementioned indexing limitations of Neo4J. Likewise, we had to store the domain identifiers (e.g., `Q91`, `P39`) as id attributes on nodes since node ids (and edge ids) in Neo4J are internally generated.

4 Experimental Setting

We now describe the setting for our experiments. Note that further details, including scripts, queries, raw data, configuration settings, instructions, etc., are provided at `https://dx.doi.org/10.6084/m9.figshare.3219217`. All original material and material derived from Wikidata is available under CC-BY (). The raw Wikidata dump is available under CC-0 (). The use of engines is governed by the respective vendor licences.

Data: We use the Wikidata dump in JSON format from 2016/01/04, which we converted to each representation using custom scripts. The dump contains 67 million claims describing 18 million entities, using 1.6 thousand properties, where 493 thousand claims are qualified with 1.5 million qualifiers (not including references or ranks), and 24 million claims have a datatype value.

[9] One option would be to store property meta-data such as labels locally using attributes, but this would require duplicating all such meta-data every time the property was used; we thus ruled this out as a possibility.

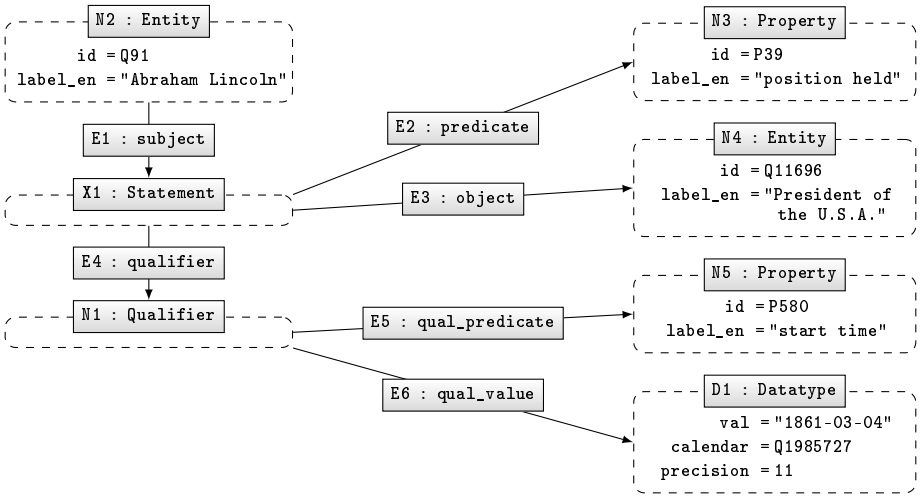


Fig. 5: A reified property graph encoding a subset of the data from Figure 1

Machine: All experiments were run on a single machine with 2× Intel Xeon Quad Core E5-2609 V3 CPUs, 32GB of RAM, and 2× 2TB Seagate 7200 RPM 32MB Cache SATA hard-disks in a RAID-1 configuration.

Engines: We used Blazegraph 2.1.0 (with Java SE "1.7.0_80"), and set 6GB of RAM (per the vendor's recommendations to not use larger heaps, but rather to leave memory for OS-level caching).[10] Although Blazegraph supports a setting called RDF* to support reification [10], this model is non-standard and cannot *directly* encode multiple Wikidata statements with the same primary relation: in RDF*, each RDF triple must be unique. Though RDF* could be used to emulate named graphs, we instead use "Quad Mode" with standard SPARQL settings.

We used Virtuoso 7.2.3-dev.3215-pthreads, where we set `NumberOfBuffers = 2720000` and `MaxDirtyBuffers = 2000000` per vendor recommendations. We used the default indexes labelled `PSOG`, `POGS`, `SP`, `OP` and `GS`, where `S`, `P` and `O` correspond to the positions of a triple and `G` to the name of a graph. Each such index is sorted and allows for prefix lookups, where for example for the index `PSOG`, given a predicate p and a subject s as input, we can find all o and g in a single lookup such that (s, p, o) is the graph named g in the data.

We used PostgreSQL 9.1.20 set with `maintenance_work_mem = 1920MB` and `shared_buffers = 7680MB`. A secondary index (i.e. B-tree) was set for each attribute that stores either entities, properties or data values (e.g. dates) from Wikidata. Specifically, in the statement table (Figure 3), each attribute **s**, **p**, **o**, and $\mathbf{o_{date}}$ has an index as well as the attributes **q**, **v**, and $\mathbf{v_{date}}$ in the qualifier table. In all other tables, only foreign keys attributes were indexed like, for example, attribute **e** in the **Label** table (Figure 3).

[10] `https://wiki.blazegraph.com/wiki/index.php/IOOptimization`

We used Neo4J-community-2.3.1 setting a 20GB heap. We used indexes to map from entity ids (e.g., `Q42`) and property ids (e.g., `P1432`) to their respective nodes. By default, Neo4J indexes nodes in the graph and their adjacent nodes in a linked-list style structure, such that it can navigate from a node to its neighbour(s) along a specific edge without having to refer back to the index.

Aside from this, we used default vendor-recommended settings. Given that the main use-case scenario we consider is to offer a public query service for Wikidata, we use REST APIs in the case of Blazegraph, Virtuoso and Neo4J. Unfortunately, however, we could not find a first-party REST API for PostgreSQL and thus resorted to using a direct command-line client.

We verified the completeness of (non-timeout) results by comparing result-sizes for the various representations. While we found minor differences (e.g., Blazegraph rejects dates like `2016-02-30`), these were $\ll 1\%$ of all results.

5 Experimental Results

Atomic lookups: In our first experiments, we wish to test the performance of atomic lookups over Wikidata statements, where we focus in particular on qualified statements that require more complex representations to encode. Recall that qualified statements consist of five types of terms: subject (s), predicate (p), object (o), qualifier predicate (q), and qualifier value (v). Hence we can consider abstract query patterns based on quins of the form $(\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{q}, \mathsf{v})$, where any term can be a variable or a constant.[11] For example, a pattern $(?, \mathsf{p}, \mathsf{o}, \mathsf{q}, ?)$ may have an instance $(\texttt{?u1}, \texttt{P39}, \texttt{Q4164871}, \texttt{P580}, \texttt{?u5})$, asking for all US presidents and the date when their presidency started. We can consider $2^5 = 32$ abstract patterns of this form, where each position is either a constant or a unique variable.

To begin, we generated a set of 1.5 million (data) quins from Wikidata, shuffled them, and used them to randomly generate 300 unique instances for each of the 31 patterns (we exclude the open pattern $(?, ?, ?, ?, ?)$, which only has one instance). We select 300 since there are only 341 instances of $(?, \mathsf{p}, ?, ?, ?)$ (i.e., there are 341 unique properties used on *qualified* statements); hence 300 allows us to have the same number of instances per pattern. In total, we have 9,300 instances, which we translate into concrete queries for our six experimental representations. Since instances of patterns such as $(?, \mathsf{p}, ?, ?, ?)$ would generate millions of results, we set a limit of 10,000 results on all queries.

We then test these queries against our four engines, where we reset the engine, clear the cache, and start with the 300 instances of the first pattern, then moving to the second pattern, etc., in sequence. Given that we wish to run 9,300 queries for each representation, we set an internal server-side query timeout of 60 seconds to ensure a reasonable overall experiment time.

[11] Note that quins are not sufficient to represent Wikidata [12], where some form of statement identifier is needed to distinguish different statements with the same primary relation; however, such identifiers are not part of the domain but rather part of the representation, hence we do not consider them in our query patterns.

Figure 6 gives the mean runtimes for each set of 300 instances per each pattern, where the y-axis is presented in log-scale (each horizontal line represents an order of magnitude), and where the presented order of patterns corresponds to the order of execution. We also include the mean considering all 9,300 instances. The x-axis labels indicate the pattern and the mean number of tuples that *should* be returned over the 300 instances (9,300 instances in the case of All). For the purposes of presentation, when computing the mean, we count a query timeout as 60 seconds (rather than exclude them, which would reduce the mean). Thus, for any set of instances that encountered a timeout – indicated by (red) dots in Figure 6 – the mean values represent a lower-bound.

We see that PostgreSQL performs best for all but three patterns, and is often an order of magnitude faster than the next closest result, most often in the lower range of query times from 1–100 ms, where we believe that the choice of client may have an effect: the direct client that PostgreSQL offers does not incur the overhead of REST incurred by other engines, which may be significant in this range. In addition, PostgreSQL can answer instances of some patterns with a mean in the 1–10 ms range, suggesting that PostgreSQL is perhaps not touching the hard-disk in such cases (an SATA hard-disk takes in the order of 10 ms to seek), and is making more efficient use of RAM than other engines.

More conceptually, PostgreSQL also benefits from its explicit physical schema: every such pattern must join through the **id** columns of the **Statement** and **Qualifier** table, which are designated and indexed as primary/foreign keys. On the other hand, in other representations, no obvious keys exists. For example, in the named graphs representation, which is most similar to the relational version, a quin is encoded with a quad $(\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{i})$ (a triple in named graph i) and a triple $(\mathsf{i}, \mathsf{q}, \mathsf{v})$ (in the default graph). There is no special key position, but Virtuoso, with its default indexes – `PSOG`, `POGS`, `SP`, `OP`, `GS` – is best suited to cases where predicates are bound along with a subject or object value. The pattern $(?, \mathsf{p}, ?, ?, \mathsf{v})$ in named graphs then illustrates a nasty case for Virtuoso: if the engine starts with the much more selective v constant (starting with p would generate many more initial results), its only choice is to use the `OP` index (the only index where `O` is key) to get q from $(\mathsf{i}, \mathsf{q}, \mathsf{v})$ using v, then `POGS` to get i from the same triple $(\mathsf{i}, \mathsf{q}, \mathsf{v})$ using (q, v), then `GS` to get s from $(\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{i})$ using i, then `SP` to get p again from $(\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{i})$ using s, then `PSOG` to get o again from $(\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{i})$ using (s, p), and so we end up joining across all five indexes in Virtuoso instead of two in the case of PostgreSQL. Similar cases can be found for other patterns and representations, where the lack of explicit keys complicates performing joins.

On the other hand, we see from the mean of All queries that the gap between Virtuoso and PostgreSQL closes somewhat: this is because Virtuoso is often competitive with PostgreSQL for more expensive patterns, even outperforming it for the least selective pattern $(?, \mathsf{p}, ?, ?, ?)$. These patterns with higher runtimes have a greater overall effect on the mean over all queries.

Conversely, we see that Neo4J and Blazegraph often fare the worst. In both cases, we found that one query would timeout and cause a subsequent series of queries to timeout, often early in the experiment. This occurred in quite a non-

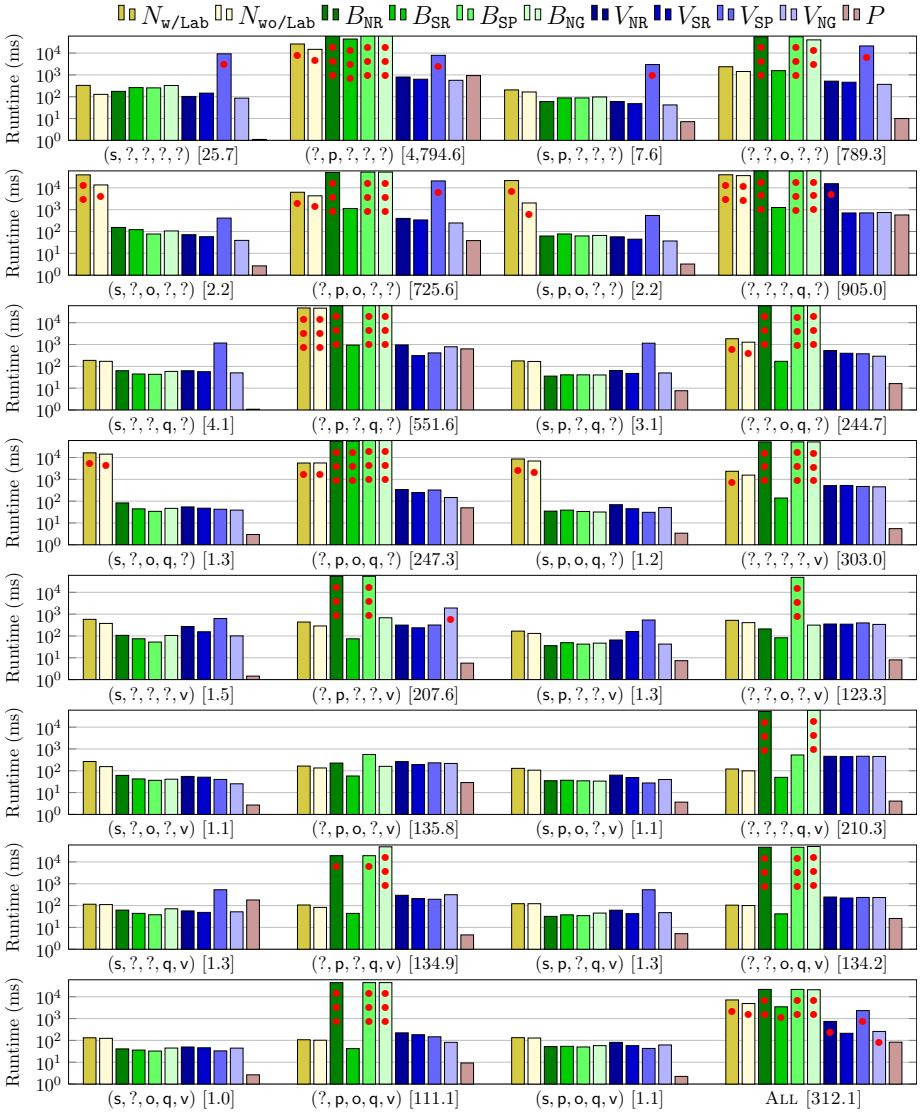


Fig. 6: Mean response times for the 31 patterns, and mean response times for All patterns, where N, B, V and P refer to Neo4J, Blazegraph, Virtuoso and Postgres, resp.; `w/Lab` and `wo/Lab` refer to Neo4J representations with and without label information, resp.; `NR`, `SR`, `SP`, and `NG` refer to n-ary relations, standard reification, singleton properties and named graphs, resp.; one, two and three (red) dots on a bar indicate that]0–33%],]33–66%] and]66–100%] of queries timed out, resp.; timeouts are counted as 60 seconds; the x-axis labels indicate the pattern and the mean number of complete results for the instances

deterministic manner when dealing with low-selectivity instances: in repeated runs, different queries would instigate the issue. Given this non-determinism and our observations of memory usage and various settings, we believe this instability is caused by poor memory management, which leads to swapping and garbage collector issues. On the other hand, for patterns and representations where we did not encounter timeouts, we found that Blazegraph was often competitive and in some cases faster than Virtuoso, with Neo4J being the slowest.

Investigating Neo4J's performance, we speculated that the manner in which Neo4J operates – traversing nodes in a breadth-first manner – could be negatively affected by storing lots of embedded labels and aliases in different languages on the nodes, adding a large I/O overhead during such traversals. To test this, we built a version of Neo4J without labels. We found that this indeed improved performance, but typically by a constant factor: i.e., it was not the sole cause of slow query times. Investigating further, we found that the engine does not seem to make use of reliable selectivity estimates. One such example of this is to compare the results of (`s`, ?, ?, ?, ?) versus (`s`, ?, `o`, ?, ?) where the performance of the latter query is much worse than the former despite the fact that the latter could be run as the former, applying a simple filter thereafter. Investigating these curious results, we found that Neo4J often naively chooses to start with `o` rather than `s`, where objects often have low selectivity; for example, they can be countries of birth, or genders, etc. In general, we found the query planning features of Neo4J to be ill-suited to these sorts of queries.

Basic graph patterns: The atomic-lookup experiments show a clear distinction between two pairs of engines: Neo4J–Blazegraph and Virtuoso–PostgreSQL. Although PostgreSQL generally outperformed Virtuoso, these results may not hold for other types of queries. Thus having analysed atomic lookups involving qualifiers, next we wished to focus on queries involving joins. To guide the design of these queries, we studied the 94 example queries present on the official Wikidata query service[12] – queries submitted by the Wikidata community as exemplary use-cases of the service – to see what are the most common types of joins that appear. From this study, we identified that queries most commonly feature "`s`–`s`" joins – joins from subject to subject – and "`s`–`o`" joins – joins from subject to object, specifically on the primary relation. These observations correspond with other analyses of real-world SPARQL queries [6,17]. We also identified that for almost all queries, `p` is bound and that patterns form trees. About 80/94 queries followed this pattern: 11/94 involved qualifier information and 3/94 involved variables in the predicate position. We thus designed a set of experiments to generate a large number of instances of basic graph patterns (bgps) that follow the same abstract structure as most commonly observed in the use-case queries.

The first query pattern, which we refer to as DEPTH-1 "snowflake", starts with a subject variable and adds between 1 and 6 edges to it. The predicate of each edge is bound, and each object is made a constant or a variable with equal probability. The subject variable is projected; object variables are projected or not with

[12] `https://query.wikidata.org/`, see "Examples".

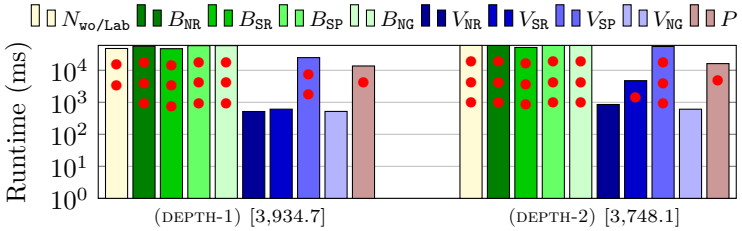


Fig. 7: Mean response times and result sizes for the snowflake queries of DEPTH-1 and DEPTH-2 (please see caption of Figure 6 for explanation of legend, etc.)

equal probability. An example DEPTH-1 bgp is as follows, with projected variables underlined: {(?s, P19, Q2280), (?s, P106, ?v1), (?s, P31, ?v2), (?s, P27, ?v3)}, which asks for entities born in Minsk (P19,Q2280), their occupation(s) (P106), their citizenship(s) (P27), and checks that they are an instance (P31) of something. Each such instance thus tests s–s joins. We run an SQL query to generate 10,000 clusters of six unique primary relations from the data (joined by subject). We then randomly select 300 clusters to generate the DEPTH-1 bgps we use in the experiments. These are then converted into concrete queries for each representation. Each such query is guaranteed to return at least one result.

The second query pattern, which we call DEPTH-2 "snowflake", adds a second layer to the query. More specifically, we start with a DEPTH-1 query and then select an object variable at random to be used as the subject variable for a nested DEPTH-1 bgp (with out-degree of 0–4); e.g. {(?s, P140, Q9592), (?s, P27, Q30), (?s, P725, ?v1), (?s, P39, ?o), (?o, P511, Q1337757), (?o, P910, ?u1)} is a DEPTH-2 bgp that asks for entities with religion Catolicism (P140,Q9592), who are citizens of the U.S. (P27,Q30), who have a given name (P725), and who have a position (P39) with the title "His Eminence" (P511,Q1337757), additionally requesting the main category (P910) of that position. The result is a tree of depth two that tests s–s joins and s–o joins. We again use an SQL query to generate 10,000 clusters of data that can answer such patterns, randomly selecting 300 to generate instances that we convert into concrete queries in each representation.

For each setting, we run each batch of 300 queries sequentially and separately, clearing the cache and resetting the engine before each such batch. As before, each query is set to return a limit of 10,000 results and each engine is assigned an internal timeout of 60 seconds where, in Figure 7, we see the results of these experiments, using similar conventions as before. The mean result sizes show that these queries tend to have low selectivity (i.e., lots of results).

We see that this time, Virtuoso performs best overall, where in particular the n-ary relations and named graph representations experience zero timeouts and outperform PostgreSQL (with a 15–19% timeout rate) by an order of magnitude in both experiments. Regarding these two engines, the situation is almost the inverse of that for the previous experiments. In these experiments, under the standard reification and named graphs settings, the predicates are always

bound, and the queries require a lot of self-joins (joins on the same table, for example for multiple distinct tuples about a given subject entity, which are particular common in SPARQL due to the structure of RDF). Virtuoso's indexing scheme – which in a relational sense considers predicates as a form of key for complete quads – is well-tuned for this morphology of query and excels for these settings. In more depth, Virtuoso can perform prefix lookups, meaning that a pattern $(\mathsf{s}, \mathsf{p}, ?)$ or $(?, \mathsf{p}, \mathsf{o})$ can be performed in one lookup on its PSOG/POGS indexes respectively. On the other hand, PostgreSQL must join s/o with p (non-key attributes), through the statement ids (the key attribute), incurring a lot more work. In the case of singleton properties for Virtuoso, since the predicate encodes the statement identifier, these remain as a variable, which again means that Virtuoso struggles. We also observe that s–o joins cause some difficulty for Virtuoso in the n-ary relation setting for DEPTH-2 queries.

On the other hand, we see that most of the queries time-out for Neo4J and Blazegraph; more specifically, we encountered the same "domino effect" where one timeout causes a sequence of subsequent timeouts. Note that we only tested Neo4J without labels (the setting in which it is most efficient).

Query features: The aforementioned 94 use-case queries (taken from the public query service) use a mix of query features, where from SPARQL 1.0, 45/94 queries use `ORDER BY`, 38/94 use `OPTIONAL`, 21/94 use `LIMIT`, and 10/94 use `UNION`; and from SPARQL 1.1, 27/94 use `GROUP BY`, 18/94 use recursive property paths (i.e., `*` or `+`, mainly for traversing type hierarchies), 10/94 use `BIND`, and 5/94 use negation in the form of `NOT EXISTS`. Likewise, queries often use `FILTER` and occasionally sub-queries, though usually relating to handling labels. In any case, we see that a broad range of SPARQL (1.1) features are often used.

Testing all of these combinations of features in a systematic way would require extensive experiments outside the current scope but as an exercise, we translated a selection of eight such queries using a variety of features to our six representations. During that translation, we encountered a number of difficulties.

The first issue was with *recursive property paths*. With respect to SPARQL, these queries cannot be expressed for `SR` and `SP` due to how the RDF is structured, and are only possible in `NG` assuming the default graph contains the union (or merge) of all named graphs.[13] Although Neo4J specifically caters for these sorts of queries, the necessity to use a reified representation means that they are no longer possible. One solution in such cases would be to store the "direct" unqualified relations also. With PostgreSQL, it is in theory possible to use `WITH_RECURSIVE` to achieve paths, though being a more general operator, it is more difficult for the engine to optimise, thus incurring performance costs.

The second issue was with *datatype values*. In the case of PostgreSQL, a number of queries involve filters on dates, where currently we store datatype values as a JSON string, where SQL does not allow for parsing such objects. In order to better handle datatypes, one possibility is to create a different table for each

[13] Paths cannot be traversed across named graphs in SPARQL unless loaded into the default graph by `FROM` clauses, which would require using specific statement ids.

datatype, with columns for the meta-information it can carry (calendar, precision, etc.). Another option is to use a non-standard feature for semi-structured data, where PostgreSQL has recently released support for accessing values in JSON objects stored in a table through its JSONB functionality.

Aside from these issues, the user queries can be expressed in any of the settings. We refer the interested reader to our previous work [12], where we present some example results for a selection of fourteen such queries over five different SPARQL engines and four representations.

6 Related Work

The goal of our work was to compare SPARQL, relational and graph databases for the purposes of querying Wikidata. Other authors have performed similar experimental comparisons across database families. Abreu et al. [1] compared a SPARQL engine (RDF-3X [15]) with a number of graph databases, showing that RDF-3X outperformed existing graph databases for the task of graph pattern matching. Bizer and Schultz [3] proposed the Berlin SPARQL Benchmark for comparing SPARQL engines with SPARQL-to-SQL engines and MySQL directly, concluding that MySQL easily outperformed the best performing SPARQL engine; however, these results were published in 2009 when SPARQL had only been standardised for one year, and likewise the dataset selected was inherently relational in nature. Our results complement such studies.

7 Conclusions

Our experiments revealed a number of strength and weaknesses for the tested engines and representations in the context of Wikidata. In terms of where systems could be improved, we recommend that Blazegraph and Neo4J should better isolate queries such that one poorly performing query does not cause a domino effect. Likewise, our results show that Neo4J would benefit from better query planning statistics and algorithms, as well as the provision of more customisable indices, particular on edge information. With respect to Virtuoso, it falls behind PostgreSQL in our first experiments due to the default indexing scheme chosen, but performs better in our second experiments based on real-world queries where predicates are bound and joins follow more "typical" patterns. On the other hand, PostgreSQL could benefit from better support for semi-structured information, where JSONB is indeed a step in the right direction. Regarding representations, we found that standard reification and named graphs performed best, with n-ary relations following in third, and singleton properties not being well-supported.

In this paper, we have not covered all pertinent issues for choosing an engine and representation – issues such as licensing, standardisation, support for inference, federation, etc. – but based on our results for query performance, the best configuration for a Wikidata query service would probably (currently) use Virtuoso and named graphs: a combination that performed well in experiments and that supports the various query features needed, including property paths.

Acknowledgements This work was partially funded by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004. The second author was supported by Fondecyt Grant No. 11140900 and the third author by Fondecyt Grant No. 11150653. We also thank Markus Krötszch for his contributions to the original workshop paper.

References

1. Abreu, D.D., Flores, A., Palma, G., Pestana, V., Piñero, J., Queipo, J., Sánchez, J., Vidal, M.: Choosing between graph databases and RDF engines for consuming and mining linked data. In: COLD (2013)
2. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. SWJ 2(1), 33–42 (2011)
3. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. IJSWIS 5(2), 1–24 (2009)
4. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. 35(1), 3–8 (2012)
5. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the linked data web. In: ISWC. pp. 50–65 (2014)
6. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., Fuente, P.D.L.: An empirical study of real-world SPARQL queries. In: USEWOD (2012)
7. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: SSWS. pp. 94–109 (2009)
8. Harris, S., Seaborne, A., Prud'hommeaux, E. (eds.): SPARQL 1.1 Query Language. W3C Recommendation (21 March 2013)
9. Hartig, O.: Reconciliation of RDF* and property graphs. CoRR abs/1409.3288 (2014)
10. Hartig, O., Thompson, B.: Foundations of an alternative approach to reification in RDF. CoRR abs/1406.3399 (2014)
11. Hayes, P., Patel-Schneider, P.F. (eds.): RDF 1.1 Semantics. W3C Recommendation (25 February 2014)
12. Hernández, D., Hogan, A., Krötzsch, M.: Reifying RDF: What Works Well With Wikidata? In: SSWS. pp. 32–47 (2015)
13. Inc., C.: Stardog 4.0: The Manual. `http://docs.stardog.com/` (2016)
14. Inc., F.: AllegroGraph RDFStore: Web 3.0's Database. `http://www.franz.com/agraph/allegrograph/` (2012)
15. Neumann, T., Weikum, G.: x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. PVLDB 3(1), 256–263 (2010)
16. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification? Making statements about statements using singleton property. In: WWW. ACM (2014)
17. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: ISWC. pp. 261–269 (2015)
18. Stonebraker, M., Kemnitz, G.: The Postgres next generation database management system. CACM 34(10), 78–92 (1991)
19. The Neo4j Team: The Neo4j Manual v2.3.1. `http://neo4j.com/docs/` (2015)
20. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Linked Data Management., pp. 193–237 (2014)
21. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Comm. ACM 57, 78–85 (2014)
22. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., Ding, L.: Supporting scalable, persistent Semantic Web applications. IEEE Data Eng. Bull. 26(4), 33–39 (2003)