CEUR-WS.org/Vol-3262/paper12.pdf

Hybrid Structured and Similarity Queries over Wikidata plus Embeddings with Kypher-V

Hans Chalupsky[1], Pedro Szekely[1]

[1] *Information Sciences Institute, University of Southern California*

Abstract

Augmenting large knowledge graphs such as Wikidata with unstructured sources represented by embedding vectors is important for a number of applications, such as similarity-based search, recommendation systems, question answering, entity linking, etc. However, building such hybrid systems today requires ad hoc combinations of disparate technologies such as structured query languages with approximate nearest neighbor search, since there are no truly hybrid query systems available aimed at simultaneously querying large knowledge graphs and vector data. Moreover, similarity search over large vector data is very resource and main-memory intensive requiring high-end compute servers with large amounts of RAM. This paper introduces `Kypher-V`, an extension to KGTK's Kypher query language and processor, that allows users to augment Wikidata and other KGs with large-scale vector data and efficiently query it through a hybrid query system using a structured query language based on Cypher. We show that Kypher-V can efficiently store, index and query 40M 1024-d vectors and effectively filter and join results with Wikidata-size datasets on a laptop.

Keywords

Wikidata, Knowledge Graphs, Similarity, Embeddings, KGTK, Kypher, Cypher

1. Introduction

Many important tasks and applications such as search, recommendation systems, question answering, entity linking, link-prediction, etc. require the effective combination of large amounts of structured and unstructured knowledge. Wikidata [1] is a popular, broad-coverage knowledge graph (KG) often used for such applications, which contains circa 100 million entities described with over 1.4 billion statements.[1] Despite its impressive size, however, Wikidata is still incomplete and additional unstructured sources such as text or images might need to be added to fill gaps and support a particular task.

A common approach for representing unstructured data is the use of high-dimensional embedding vectors that cluster objects by their latent semantic dimensions, and that provide a uniform representation to support operations such as semantic similarity or neural net-based machine learning. Recent work on knowledge graph embeddings such as, for example, TransE [2] or ComplEx [3] allows similar vector representations and

Wikidata'22: Wikidata workshop at ISWC 2022

EMAIL: hans@isi.edu (H. Chalupsky); pszekely@isi.edu (P. Szekely)
URL: https://www.isi.edu/~hans/ (H. Chalupsky); https://usc-isi-i2.github.io/szekely/ (P. Szekely)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

[1] https://grafana.wikimedia.org/d/000000175/wikidata-datamodel-statements

operations to also be applied directly to structured data such as knowledge graphs.

Hybrid systems that combine structured and unstructured data have the promise to amplify the best features of both while compensating for their weaknesses. For example, structured data is richly described and provides useful constraints for high precision results, but it lacks a notion of similarity and is generally incomplete. Unstructured data is often very large-scale, high-coverage, captures most relevant semantic dimensions automatically and can be used to query by example, but it lacks constraints and becomes less precise with increasing number of objects.

Effective combination of structured and unstructured sources is still an active research area and generally requires extensive experimentation. Unfortunately, building such hybrid systems today is difficult and requires ad hoc combinations of disparate technologies such as SPARQL or SQL with approximate nearest neighbor search (ANNS) systems such as Faiss [4] and others, since there are no truly hybrid query systems available aimed at simultaneously querying large knowledge graphs and vector data (the KGTK similarity service [5] is an example). Similarity search over large vector data is also very resource and main-memory intensive requiring high-end compute servers with large amounts of RAM, providing a high barrier to entry for researchers.

To support this emerging new area of large KG + vector-based systems, this paper introduces `Kypher-V`, which is an extension to Kypher, the query language and processor of the KGTK Knowledge Graph Toolkit [6], which allows creating personalized variants of Wikidata on a laptop, and enables running analytic queries faster than a Wikidata SPARQL endpoint [7]. The new Kypher-V allows users to augment Wikidata and other KGs with large-scale vector data and efficiently query it through a hybrid query system using a familiar structured query language based on Cypher. Both Kypher and Kypher-V use the KGTK representation and can therefore be used not just on Wikidata but also RDF KGs, for example, DBpedia [8], and others. We show that Kypher-V can efficiently store, index and query 40M 1024-d vectors and effectively filter and join results with Wikidata-size datasets on a standard laptop.

The rest of the paper is structured as follows. Section 2 describes the KGTK toolkit and gives a brief introduction to the Kypher query language. Section 3 describes vector representation and storage used by Kypher-V, Section 4 introduces the new types of query patterns supported by Kypher-V, Section 5 presents related work, and Section 6 presents conclusions, discussion of the results and directions for future work.

2. Background

The Knowledge Graph Toolkit (KGTK) [6] is a comprehensive framework for the creation and exploitation of large hyper-relational KGs, designed for ease of use, scalability, and speed. KGTK represents KGs in tab-separated (TSV) files with four columns: edge-identifier, head, edge-label, and tail. All KGTK commands consume and produce KGs represented in this format, so they can be composed into pipelines to perform complex transformations on KGs. KGTK provides a suite of import commands to import Wikidata, RDF and popular graph representations into the KGTK format. A rich collection of transformation commands make it easy to clean, union, filter, and sort KGs; graph combination commands support efficient intersection, subtraction, and joining of large KGs; graph analytics commands support scalable computation of centrality metrics such as PageRank, degrees, connected components and shortest paths; advanced commands support lexicalization of graph nodes, and computation of multiple variants of text and graph embeddings over the whole graph. In addition, a suite of export commands supports the transformation of KGTK KGs into commonly used formats, including the Wikidata JSON format, RDF triples, JSON documents for ElasticSearch indexing and graph-tool.[2] Finally, KGTK allows browsing KGs in a UI using a variant of SQID;[3] and includes a development environment using Jupyter notebooks that provides seamless integration with Python Pandas. KGTK can process Wikidata-sized KGs, with billions of edges, on a laptop.

Kypher (`kgtk query`) is one of 55 commands available in KGTK. Kypher stands for *KGTK Cypher* [7]. Cypher [9] is a declarative graph query language originally developed at Neo4j. OpenCypher[4] is a corresponding open-source development effort for Cypher which forms the basis of the new Graph Query Language (GCL).[5] We chose Cypher since its ASCII-art pattern language makes it easy even for novices to express complex queries over graph data.

Kypher adopts many aspects of Cypher's query language, but has some important differences. Most importantly, KGTK and therefore Kypher use a hyper-relational quad-based data model that explicitly represents edges as first-class objects which is more general than the labeled property graph (LPG) model used by Cypher. For this reason Cypher and Neo4j cannot be used to express queries over all aspects of KGTK data (see [10] for a discussion of a unifying data model similar to KGTK's that can encompass LPG's, RDF and RDF*). Other differences are that Kypher only implements a subset of the Cypher commands (for example, no update commands) and has some differences in syntax, for example, to support naming and querying over multiple graphs.

To implement Kypher queries, we translate them into SQL and execute them on SQLite, a lightweight file-based SQL database. Kypher queries are designed to look and feel very similar to other file-based KGTK commands. They take tabular file data as input and produce tabular data as output. There are no servers and accounts to set up, and the user does not need to know that there is in fact a database used underneath to implement the queries. A cache mechanism makes multiple queries over the same KGTK files very efficient. Kypher has been successfully tested on Wikidata-scale graphs with 1.5B edges where queries executing on a standard laptop run in milliseconds to minutes depending on selectivity and result sizes [7]. Additional information about Kypher and its capabilities can be found online. [6]

[2]https://graph-tool.skewed.de
[3]https://sqid.toolforge.org/
[4]https://opencypher.org/
[5]https://www.gqlstandards.org/home
[6]https://kgtk.readthedocs.io/en/latest/transform/query/#overview

3. Vector representation and storage

Kypher-V represents vectors as special kinds of literals that are linked to KG objects via edges in KGTK format. For example, the following edge in KGTK's TSV format with edge ID `e4925` associates node `Q40` (or Austria) with a text embedding vector as `node2` of the edge (there is nothing special about this representation pattern, any scheme can be used that associates a node or edge ID with a vector):

```
id      node1  label      node2
e4925   Q40    textemb    0.1642,-0.5445,-0.3673,...,0.3203,0.6090
```

When vector data is queried, it is imported and cached if necessary just like any other KGTK data processed by Kypher. The main difference is that for faster query processing, vector literals get translated into binary format first before they are stored in the Kypher graph cache based on an indexing directive. Various up-front transformations such as data type conversions, normalization, storage type, etc. can also be controlled here. Finally, vector files are treated as large contiguous and homogeneous arrays, so that vectors can be referenced and accessed efficiently by vector table row ID instead of having to use an index.

This scheme stores vectors on disk and brings them into main memory on demand during a query just as any other KG data. Since vector literals can be quite large, it is important that these accesses are fast and minimized to maximize query speed. For more limited applications, it is possible to precompute similarities and only store the results instead of the full vector data (see, for example, the RDFSim browser [11]). However, we chose to explicitly store vectors to allow more open-ended queries, comparison against dynamic data (e.g., vectors from user queries), as well as vector management in general which can facilitate downstream applications such as machine learning.

Kypher-V can store and simultaneously query any number of different types of embedding vectors. The only requirement is that when two vectors are compared in a query, they must have the same dimensionality and should come from the same vector set, since it does not make sense, for example, to compute a similarity between a ComplEx and TransE vector. Just as the core Kypher system it is based on, Kypher-V is optimized for read-only processing. Nevertheless, updates and dynamic inputs can be handled but should be limited to small-batch data for best performance.

4. Similarity queries

The following types of vector queries are supported by Kypher-V:

1. Vector computation: given one or more vectors accessed in a query, compute a function based on those vectors. Currently supported vector functions are inner product, cosine similarity and Euclidean distance.
2. Vector search: traditional similarity search that finds the top-k similar vectors and associated nodes for a given input. Only cosine similarity is supported at the moment.

3. Vector join: a generalized form of vector search that filters input vectors and output vectors based on arbitrary Kypher query patterns.
4. Multi-vector computations: a planned extension is to be able to aggregate similarities from multiple vector types, as needed, for example, by the `topsim` computation of KGTK's similarity service.

Below we will illustrate the different types of queries through examples and describe any pertinent system aspects needed to support them.

4.1. Experimental setup

All queries shown below use one or more of the embedding sets listed in Table 1 as an input, together with a custom KGTK version of Wikidata that has all scholarly articles removed, which leaves about 55M entities and 500M statements. Provided query and command execution times are intended to be rough indicators of performance, but they were not determined in a rigorously controlled environment. Queries were measured on the command line, which includes overhead from loading KGTK and needed ANNS index files. These overheads can be eliminated by using Kypher-V through its Python API. All queries and commands were executed on a Lenovo Thinkpad W541 laptop with 8 cores, 30GB of RAM and a 2TB SSD disk drive.

4.2. Vector computation queries

Figure 1 shows the most basic vector computation available in Kypher-V. It accesses embedding vectors for two given QNodes `Q868` and `Q913` and computes the similarity between them. This query and the ones shown below all reference previously imported and indexed Wikidata graphs and vector data. The particular embedding vectors used here are 768-d text embeddings derived from the first sentences of Wikipedia short abstracts using KGTK's `text-embedding` command, referenced here as `short_abstracts_textemb` or simply `emb`:

```
!$kypher -i short_abstracts_textemb -i labels
  --match 'emb:    (x:Q868)-[]->(xv), (y:Q913)-[]->(yv),
           labels: (x)-[]->(xl), (y)-[]->(yl)'
  --return 'xl as xlabel, yl as ylabel, kvec_cos_sim(xv, yv) as sim'

xlabel          ylabel          sim
'Aristotle'@en  'Socrates'@en   0.84781
```

Figure 1: Query to compute similarity between two Wikidata QNodes

We can also use these computations to compute top-*k* similarities for a node in a brute-force way. For example, in Figure 2 we find the top-5 similar philosophers for Socrates by first finding all nodes `y` with an occupation link (`P106`) to philosopher (`Q4964182`), then computing the similarity between the respective vectors and the vector for Socrates, and finally sorting and reporting the top-5 results.

```
!$kypher -i short_abstracts_textemb -i labels -i claims
  --match 'emb:      (x:Q913)-[]->(xv), (y)-[]->(yv),
          claims:   (y)-[:P106]->(:Q4964182),
          labels:   (x)-[]->(xl), (y)-[]->(yl)'
  --return 'xl as xlabel, yl as ylabel, kvec_cos_sim(xv, yv) as sim'
  --order  'sim desc' --limit 5

xlabel          ylabel                  sim
'Socrates'@en   'Socrates'@en           1.00000
'Socrates'@en   'Heraclitus'@en         0.93445
'Socrates'@en   'Hippocrates'@en        0.93040
'Socrates'@en   'Diogenes Laërtius'@en  0.91655
'Socrates'@en   'Ammonius Hermiae'@en   0.90550
```

Figure 2: Brute-force top-k similarity query with text embeddings

In this example enumerating a full set and then computing similarity for each of its members was feasible and still fast (about 1 second), since the set was small (only about 10,000 elements in our main Wikidata claims table). On a larger set, for example all humans in Wikidata with about 9 million elements, this becomes very inefficient (4 minutes) and we will use nearest neighbor indexing described below to speed things up. Note also that different embeddings will yield different results with similarity values that are not directly comparable across embedding types. For example, here we show the top-5 similars based on 100-d ComplEx graph embeddings:

```
!$kypher -i complexemb -i labels -i claims
  --match 'emb:      (x:Q913)-[]->(xv), (y)-[]->(yv),
          claims:   (y)-[:P106]->(:Q4964182),
          labels:   (x)-[]->(xl), (y)-[]->(yl)'
  --return 'xl as xlabel, yl as ylabel, kvec_cos_sim(xv, yv) as sim'
  --order  'sim desc' --limit 5

xlabel          ylabel                          sim
'Socrates'@en   'Socrates'@en                   1.00000
'Socrates'@en   'Menexenus'@en                  0.82710
'Socrates'@en   'Antisthenes'@en                0.82499
'Socrates'@en   'Zeno of Citium'@en             0.82058
'Socrates'@en   'Critobulus son of Crito'@en    0.81729
```

Figure 3: Brute-force top-k similarity query with ComplEx embeddings

4.3. Vector search queries

Figure 4 shows a vector search operation expressed in Kypher-V. It finds the top-k similar Wikidata nodes for Socrates (`Q913`) according to Wikipedia short abstract text embeddings. Vector search is expressed via a *virtual* graph edge `kvec_topk_cos_sim` which is implemented via an SQLite virtual table (a custom computation that can generate multiple rows and behaves like a regular table). Virtual edges are a Kypher-V extension to Cypher that take a number of user-specified input parameters expressed in Cypher property syntax. The parameters here are k controlling how many results to return, and *nprobe* which specifies how many quantizer cells should be searched (more on that below). Note that the results here are slightly different than on a similar query above, since we do not require that the results have philosopher as their occupation which accounts for the differences.

```
!$kypher -i short_abstracts_textemb -i labels
  --match 'emb:    (x:Q913)-[]->(xv),
                   (xv)-[r:kvec_topk_cos_sim {k: 5, nprobe: 4}]->(y),
          labels: (x)-[]->(xl), (y)-[]->(yl)'
  --return 'xl as xlabel, yl as ylabel, r.similarity as sim'

xlabel          ylabel               sim
'Socrates'@en   'Socrates'@en        1.00000
'Socrates'@en   'Stratos'@en         0.94111
'Socrates'@en   'Amphictyonis'@en    0.93867
'Socrates'@en   'Anytus'@en          0.93465
'Socrates'@en   'Heraclitus'@en      0.93445
```

Figure 4: Vector search query with text embeddings

4.4. ANNS index creation and use in Kypher-V

In a naive brute force implementation as the one from Figure 2, the query in Figure 4 would have to scan all Wikidata nodes to find those with highest similarity (about 52 million for the version we are using). Instead, we are using a custom approximate nearest neighbor search (ANNS) index based on Faiss [4] which executes this query in less than a second.

Generic Faiss assumes ANNS indexes and their vectors are created and loaded into RAM for best search performance, and building and using of disk-based indexes has relatively limited support. Unfortunately, this requires very large amounts of RAM and leads to high startup cost due to index loading time, which would require Kypher-V to use a server architecture to amortize the cost. Instead, we are using a different approach that leverages the underlying database to store quantization information so relevant vectors can be loaded lazily from disk as needed by a query instead of eagerly at system startup. Index creation and use proceed in the following steps:

1. K-means clustering of the vectors of a vector set using the standard Faiss API. We sample the input to limit main memory use and clustering run time based on user-provided resource limits. Our currently largest vector set that requires approximately 180GB to store can be clustered with 25GB of RAM. The resulting cluster centroids form the basis of the quantization cells (q-cells) vectors will be assigned to in the next step. Centroids are saved to a file as a standard Faiss flat file index which is generally small and can be loaded quickly if needed for a vector search query.

2. Quantization assigns vectors to the q-cell centroid they are closest to and writes the respective q-cell IDs as a column to the vector table. After creating an appropriate database index, this allows us to quickly find all vectors of a q-cell using a database query.
3. Sorting of the vector table by q-cell ID improves locality, which is important, since now vectors of the same q-cell can all be retrieved from a single or just a few disk pages instead of having to read them from all over a large file.
4. At query time we search an input vector V for the top *nprobe* q-cell centroids closest to it using the Faiss search API. Thus, *nprobe* controls the depth of the similarity search trading off recall for speed.
5. Once the set of *nprobe* relevant q-cells has been determined, we bring in all the relevant vectors of these q-cells from disk and use a dynamically constructed Faiss search index to find the top-k matches for input V.

Importing large vector sets and creating nearest neighbor (NN) indexes for them is expensive. K-Means clustering complexity is $O(ndki)$ where n d-dimensional vectors are clustered into k centroids in i iterations. Kypher-V allows the user to control some of these parameters such as the number of centroids or iterations to keep run time in check. Nevertheless, the size of the dataset and the dimensionality of the vectors are two of the biggest determiners of overall run time which cannot easily be changed. In Table 1 we show data import and indexing times for different embedding sets using different parameter settings. All imports and clusterings could be handled by a laptop, even though some of the run times were quite large.

Dataset	Type	N	Dim	DB size	Import	NN-Index	Total
Wikidata KG	ComplEx	53M	100	26GB	1.4 hours*	1.1 hours	2.5 hours
Wikidata KG	TransE	53M	100	26GB	25 min	2 hours	2.4 hours
Wikidata KG	Text	39M	1024	182GB	8 hours*	11.5 hours	19.5 hours
Short abstracts	Text	5.9M	768	24.5GB	16 min	28 min	44 min

Table 1
Comparison of import and NN-indexing times on different sets of embedding vectors; (*) took longer due to import from a network drive

4.5. Vector join queries

Vector joins generalize basic vector search by adding additional input and output constraints to restrict the results generated by the search. In general, a similarity join between sets X and Y joins two elements $x_i \in X, y_j \in Y$ if y_j is in the top-k similars of x_i, or if the similarity $sim(x_i, y_j) \geq t$ where t is some minimal threshold similarity. For example, the query in Figure 5 joins Socrates with the set of all humans to find the top-5 similar matches. This query executes in less than a second now which is a greater than 250x speedup compared to the brute-force version of this query described in Section 4.

The implementation described here is somewhat imperfect, since it requires a large enough k to be specified to select enough matches that satisfy the join constraints and the

```
!$kypher -i short_abstracts_textemb -i labels -i claims
  --match 'emb:      (x:Q913)-[]->(xv),
                     (xv)-[r:kvec_topk_cos_sim {k: 100, nprobe: 4}]->(y),
           claims:   (y)-[:P31]->(:Q5),
           labels:   (x)-[]->(xl), (y)-[]->(yl)'
  --return 'xl as xlabel, yl as ylabel, r.similarity as sim'
  --limit 5

xlabel          ylabel              sim
'Socrates'@en   'Socrates'@en       1.00000
'Socrates'@en   'Anytus'@en         0.93465
'Socrates'@en   'Heraclitus'@en     0.93445
'Socrates'@en   'Hippocrates'@en    0.93040
'Socrates'@en   'Cleisthenes'@en    0.92928
```

Figure 5: Vector join query with text embeddings

limit clause. An experimental version of Kypher-V eliminating this limitation through *dynamic scaling* of *k* is available on the KGTK repository. For certain types of queries this can also be compensated through reformulation, such as using a similarity threshold and/or aggregation functions to get the desired result. For example, Figure 6 shows how we can find the top-1 human match for an input set of three philosophers.

```
!$kypher -i short_abstracts_textemb -i labels -i claims
  --match 'emb:      (x)-[]->(xv),
                     (xv)-[r:kvec_topk_cos_sim {k: 100, nprobe: 4}]->(y),
           claims:   (y)-[:P31]->(:Q5),
           labels:   (x)-[]->(xl), (y)-[]->(yl)'
  --where 'x in ["Q859", "Q868", "Q913"] and x != y and r.similarity >= 0.9'
  --return 'xl as xlabel, max(r.similarity) as sim, yl as ylabel'

xlabel          sim        ylabel
'Aristotle'@en  0.97378    'Bryson of Achaea'@en
'Plato'@en      0.92298    'Franciszek Kasparek'@en
'Socrates'@en   0.93465    'Anytus'@en
```

Figure 6: Generalized vector join query with multiple inputs

5. Related work

There are only very few systems available today that support hybrid querying of vector data combined with structured constraints. Alibaba AnalyticDB-V [12], Milvus [13], Vearch [14] and ElasticSearch [15] all support some form of hybrid querying, but only AnalyticDB-V (a very large scale proprietary system) integrates this capability in a standard full-function query language such as SQL. To the best of our knowledge, there is no system available today that supports this for a query language focused on knowledge graphs as we do with Kypher-V. Systems such as Wembedder [16] and KGvec2go [17] provide embedding vectors over Wikidata and other KGs through Web APIs, but do not have a general hybrid query capability. However, the embedding vectors from these systems could be used by Kypher-V. RDFSim [11] is a system that supports similarity-based browsing in KGs, but through precomputed similarity sets only.

6. Discussion and Conclusions

The main objective of KGTK, Kypher and Kypher-V is to democratize the exploitation of Wikidata so that anyone with modest computing resources can take advantage of the vast amounts of knowledge present in it. See [7] for a set of use cases that use large portions of Wikidata to distill new knowledge.

Kypher-V extends these capabilities to store, index and query large sets of embedding vectors in a fully integrated way, allowing users to efficiently combine large amounts of structured and unstructured data while still preserving the ability to use the system on modest resources such as a laptop.

The examples presented in this paper are intended to give an initial overview of the capabilities of the new system. Execution times are mostly anecdotal and a more systematic evaluation of system performance has to be reserved for a future paper. Kypher-V is still under active development and has not yet been merged with the main branch of KGTK, but early snapshots are available on the KGTK GitHub repository.[7]

We do not make any claims about the quality and usefulness of the different types of embeddings used in the examples. They are all flawed in different ways, and in fact combining multiple embedding types as supported by Kypher-V has the potential to improve overall system performance. Kypher-V is completely agnostic to the types of embeddings used, and any reasonable embeddings that can be linked to the nodes and edges of a KG can be queried by it.

The contribution of this paper is to show that KGTK, Kypher and Kypher-V are effective tools for complex analytic use cases. The paper demonstrates that Kypher-V supports a variety of hybrid queries that are difficult to implement with existing tooling. Kypher-V allows researchers and developers to investigate use cases on their own laptop, exploring extensions of Wikidata with large amounts of vector data, using minimal setup, modest resources and a simple query language.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-20-2-10002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

[7]https://github.com/usc-isi-i2/kgtk/tree/feature/kypher/embeddings

References

[1] D. Vrandečić, M. Krötzsch, Wikidata: A free collaborative knowledgebase, Communications of the ACM 57 (2014) 78–85.

[2] A. Bordes, N. Usunier, A. Garcia-Durán, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, in: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS'13, 2013, p. 2787–2795.

[3] T. Trouillon, J. Welbl, S. Riedel, E. Gaussier, G. Bouchard, Complex embeddings for simple link prediction, in: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ICML'16, 2016, p. 2071–2080.

[4] J. Johnson, M. Douze, H. Jégou, Billion-scale similarity search with GPUs, IEEE Transactions on Big Data 7 (2021) 535–547.

[5] F. Ilievski, P. A. Szekely, G. Satyukov, A. Singh, User-friendly comparison of similarity algorithms on Wikidata, ArXiv abs/2108.05410 (2021).

[6] F. Ilievski, D. Garijo, H. Chalupsky, N. T. Divvala, Y. Yao, C. Rogers, R. Li, J. Liu, A. Singh, D. Schwabe, P. Szekely, KGTK: a toolkit for large knowledge graph manipulation and analysis, in: International Semantic Web Conference, Springer, 2020, pp. 278–293.

[7] H. Chalupsky, P. Szekely, F. Ilievski, D. Garijo, K. Shenoy, Creating and querying personalized versions of Wikidata on a laptop, in: L.-A. Kaffee, S. Razniewski, A. Hogan (Eds.), Proceedings of the 2nd Wikidata Workshop (Wikidata 2021) co-located with the 20th International Semantic Web Conference (ISWC 2021), CEUR-WS.org, 2021.

[8] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, DBpedia: A nucleus for a web of open data, in: The Semantic Web, Springer, 2007, pp. 722–735.

[9] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 1433–1445.

[10] R. Angles, A. Hogan, O. Lassila, C. Rojas, D. Schwabe, P. Szekely, D. Vrgoč, Multilayer graphs: A unified data model for graph databases, in: Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), ACM, 2022.

[11] M. Chatzakis, M. Mountantonakis, Y. Tzitzikas, RDFsim: Similarity-based browsing over DBpedia using embeddings, Information 12 (2021).

[12] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, Y. Cai, AnalyticDB-V: A hybrid analytical engine towards query fusion for structured and unstructured data, Proc. VLDB Endow. 13 (2020) 3152–3165.

[13] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, C. Xie, Milvus: A purpose-built vector data management system, in: Proceedings

of the 2021 International Conference on Management of Data, SIGMOD ’21, ACM, 2021, p. 2614–2627.
[14] J. Li, H.-F. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, Y. Chen, The design and implementation of a real time visual search system on JD E-commerce platform, Proceedings of the 19th International Middleware Conference Industry (2018).
[15] ElasticSearch, Elasticsearch: Open source, distributed, restful search engine, https://github.com/elastic/elasticsearch, 2020.
[16] F. Nielsen, Wembedder: Wikidata entity embedding web service, 2017. URL: https://arxiv.org/abs/1710.04099.
[17] J. Portisch, M. Hladik, H. Paulheim, KGvec2go – knowledge graph embeddings as a service, 2020. URL: https://arxiv.org/abs/2003.05809.