Learned Indexes for Efficient Querying on Knowledge Graphs

Gaurav Dawra, Aanchal Gupta, Nandika Jain*, Yogender Kumar, Jishnu Raj Parashar, Bapi Chatterjee and Raghava Mutharaju

IIIT-Delhi, India

Abstract

There are several real-world Knowledge Graphs that are very large in size. Running SPARQL queries over such large graphs is time consuming. We propose improving the query performance over Knowledge Graphs by replacing traditional indices with learned indices. Learned indices exploit the underlying data distribution to enable more efficient query processing, leading to reduced storage overhead and faster runtime performance. Our evaluation on two Knowledge Graphs demonstrates promising gains, highlighting the potential of learned indexing as a direction for further research.

Keywords

Knowledge Graph stores, Learned Index, Query processing, RDF Stores, SPARQL, Knowledge Graph Management

1. Introduction

Knowledge Graphs (KGs) [1] are graph-based data models that capture the knowledge of a domain in the form of entities and the relations between them. The semantics associated with the entities and relations are generally defined in an ontology [2]. KGs are used across several different applications and domains to integrate and extract value from the data [3]. The publicly available KGs such as Wikidata [4], DBpedia [5], and YAGO [6], are quite large in size, with billions of nodes and edges. In our work, we use the Resource Description Framework (RDF)[1] representation for KGs. One of the popular query languages for KGs is a W3C standard named SPARQL. A SPARQL query typically involves several joins [7]. When such queries are applied over large KGs, it results in a huge latency. Since the inception of SPARQL, several optimization and fine-tuning techniques were developed [8, 9, 10]. However, the existing techniques do not consider the distribution of the underlying data, which has a direct impact on the performance of SPARQL query execution. In this work, we explore the use of machine learning techniques to improve the performance of queries. More specifically, we employ a learned index [11, 12, 13] – a hierarchical indexing technique that utilizes inference of shallow machine learning models at every level of hierarchy to guide the queries. A learned index is created based on the data distribution, which we use to replace a regular index in a KG store. Our evaluation of two KGs provides promising results. The code and SPARQL queries to evaluate our approach are available at https://github.com/bapichatterjee/TripleBit-SplineIndex.

2. Related Work

This section discusses relevant existing systems of learned indexes and RDF databases for efficient storage and retrieval.

ISWC 2025 Companion Volume, November 2–6, 2025, Nara, Japan

*Corresponding author.

gaurav19039@iiitd.ac.in (G. Dawra); aanchal21224@iiitd.ac.in (A. Gupta); nandika19064@iiitd.ac.in (N. Jain); yogender21505@iiitd.ac.in (Y. Kumar); jishnu19048@iiitd.ac.in (J. R. Parashar); bapi@iiitd.ac.in (B. Chatterjee); raghava.mutharaju@iiitd.ac.in (R. Mutharaju)

0009-0000-8033-4235 (N. Jain); 0000-0002-2742-4028 (B. Chatterjee); 0000-0003-2421-3935 (R. Mutharaju)

[1] https://www.w3.org/TR/rdf11-primer/

Learned indexes are designed to implement tasks typically performed by conventional data structures, such as search, insert, and delete, using machine learning models. Starting from the work of Kipf et al. [14], these models got extended to indexing multidimensional [15, 16] and spatial data [17]. These can be quite powerful and often more efficient in terms of query time enabled by lightweight machine learning inference, particularly with growing dataset size [18]. Additionally, memory consumption compared to classic approaches is typically an order of magnitude smaller [11]. Beyond hierarchical indexes, the technique of machine learning inference augmentation was also applied to bloom filters [19], and hash tables [20].

In this line of work, the RadixSpline (RS) [14] index is a single-pass learned index designed for sorted integer keys, capable of handling both equality and range predicate queries. It operates by selecting discrete spline points over the dataset and constructing a radix table to index these points. This combination allows the system to locate neighboring spline points relative to a lookup key, followed by a binary search to determine the exact key position using spline linear interpolation. RadixSpline has demonstrated significantly faster performance compared to traditional indexing methods.

Recently, the learned indexes were also combined with concurrent queries to take advantage of powerful multi-core machines [21, 22]. Such learned data structures provide significant performance improvements over traditional methods by fitting the exact distribution of the data or access patterns of a particular application.

In traditional RDF triple stores, such as Apache Jena [23], TripleBit [24], and RDF-3X [25], indexing is critical for efficient query execution. The general approach involves several components such as a node table, triple and quad indexes, and a prefix table. Apache Jena's [23] TDB architecture includes the node table, triple and quad indexes, and the prefix table. The node table maps Nodes to NodeIDs using a B+ Tree, while NodeID to Node mapping utilizes a sequential access file. Triple and Quad indices store triples and 4-tuples, and the prefix table handles presentation and serialization. Although robust, Jena's indexing mechanism can be further optimized with machine learning techniques.

TripleBit [24] is a fast and compact RDF data management system designed for developmental use. It represents RDF triples as a 2D-bit matrix, called the triple matrix, where columns represent the triples and rows represent the objects and subjects. The columns are lexicographically sorted by predicates and partitioned into buckets. The system employs variable-size encoding for subject/object IDs and uses column compression to represent each triple within the matrix. Inside each bucket, triples are further divided into fixed-size chunks, and TripleBit creates an index structure for these chunks within each predicate bucket, enhancing efficiency and compactness.

RDF-3X [25] is known for its sophisticated indexing and query optimization strategies, designed to handle specific types of queries efficiently. It employs various indexes to support rapid query execution. We noticed that none of the mainstream triple stores use machine learning for indexing. We aim to fill that gap by using a learned index in an RDF store.

3. Architecture

We chose TripleBit for its straightforward implementation and focus on specific query types, making it ideal for experimental development. RadixSpline, a single-pass learned index, was selected for its ability to efficiently model data distributions, offering promising build times and query latencies. Unlike mainstream triple stores like Apache Jena, Virtuoso, and RDF-3X, TripleBit's simpler codebase allows easier modification. RadixSpline's compatibility with C++ aligns well with TripleBit, facilitating seamless integration and leveraging machine learning for enhanced indexing performance.

3.1. Challenges

The objective is to replace the indexing mechanism of TripleBit with a learned index created using RadixSpline. TripleBit organizes triples into predicate-based buckets, which are further divided into fixed-size chunks. While its native LineHashIndex is specifically designed to operate on this structure,

RadixSpline requires sorted numeric keys. This structural mismatch gave rise to the following challenges, among others.

Key representation: To integrate RadixSpline with TripleBit, these subject–object or object–subject pairs had to be translated into a consistent, sortable numeric key format compatible with the learned index.

Chunk-level granularity: We need to adapt the spline construction process to operate on a chunk while still leveraging the global distributional characteristics as TripleBit constructs indexes at the chunk level.

3.2. Integration

To address these, we created a wrapper class, SplineIndex, to bridge the TripleBit interface and the RadixSpline interface (Figure 1).

Data transformation: We introduced a numeric encoding of subject–object or object–subject pairs to produce keys compatible with RadixSpline's requirements.

Chunk-wise spline construction: The SplineIndex::buildIndex() function first gathers and sorts the subject-object pairs within each chunk, then invokes the rs::MultiMap implementation of RadixSpline to create a spline index with error bounds tuned to the chunk size. The spline is constructed by modeling key positions within a bounded error, selecting representative data points, and interpolating between spline points. A radix table is then built by partitioning the sorted spline points according to their r-bit prefixes. During lookups, the r-bit prefix of the query key is used to identify the relevant spline region, after which binary and linear searches refine the prediction to locate the key within the specified error bound. This facilitates the creation of localized learned indices that still capture global distributional patterns.

Parameter tuning: RadixSpline requires tuning on parameters such as the number of spline points and radix table bits. We experimented with different error tolerance (ε) and radix bits (r) to balance query latency and memory usage. Larger ε reduces index size by reducing the number of spline points at the cost of longer binary searches. Radix bits (r) control the granularity of the radix table: higher r values create finer partitions, enabling faster access to the relevant spline segment but increasing memory usage.

Serialization: Additionally, we integrated custom serialization and deserialization functions for RadixSpline into TripleBit, thus effectively replacing its native index structure with RadixSpline.

3.3. Query Execution Benefits

This integration improves lookup efficiency for subject–object mappings within chunks, thereby reducing the size of intermediate results during SPARQL joins. Equality queries gain from the tighter error bounds of spline interpolation, while range queries benefit from RadixSpline's interpolation model, which allows large portions of data to be skipped.

4. Evaluation

In this section, we present the results of benchmarking our approach against Apache Jena, TripleBit, and RDF3x with DBLP[2] and SP^{2}Bench [26] benchmarks. We use two types of SPARQL queries – queries with single selection triple patterns and queries with join triple patterns. We set the RadixSpline parameters, error tolerance (ε) and radix bits (r), by balancing the trade-off between index size and query latency. In our evaluations, we found ε = 32 and r = 18 to provide the best balance between memory footprint and query performance.

[2] https://dblp.org/

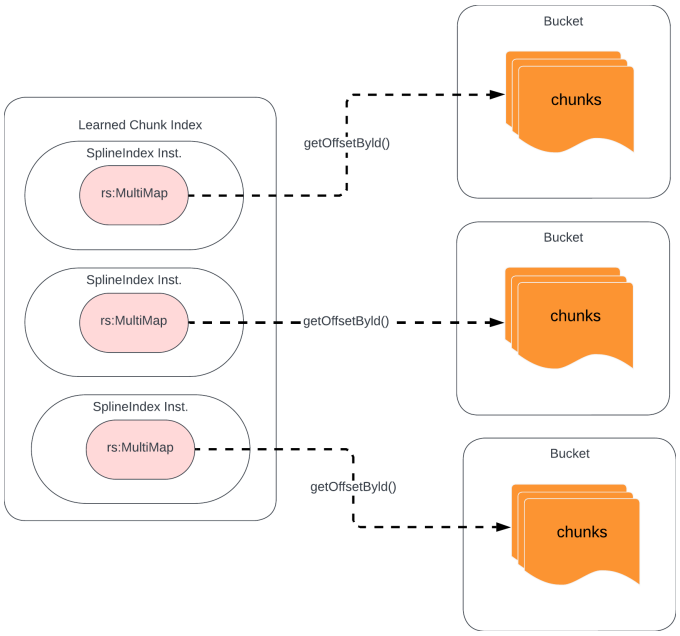

Figure 1: The storage schema of our learned triple store.

4.1. Performance on DBLP Dataset

DBLP (Digital Bibliography and Library Project) is a comprehensive online bibliography of computer science literature. It is a commonly used benchmark dataset for evaluating the performance of database management systems, particularly those that are designed to handle large amounts of data efficiently. Performance is typically measured in terms of the time it takes for the system to execute the queries and return the results.

Although the RDF schema and triples of DBLP are available, the SPARQL queries are not public. We used 350 million triples from the DBLP dataset with five queries listed at https://github.com/bapichatterjee/TripleBit-SplineIndex/blob/master/SPARQL-Queries.txt. Table 1 shows the results after benchmarking DBLP on TripleBit with SplineIndex (TBSP), Apache Jena, TripleBit, and RDF3x.

Stores	Q1	Q2	Q3	Q4	Q5
TBSP	**0.0605**	0.0057	0	**0.0001**	**4.1426**
Apache Jena	11.2515	0.1990	0	0.1130	10.9165
TripleBit	0.068	0.005784	0	0.000137	4.2223
RDF3X	1.26725	**0.00225**	0	0.00339	4.27625

Table 1
Performance of triple stores (time in seconds) on 350M triples from DBLP. TBSP stands for TripleBitWithSplineIndex. The best runtimes are in bold.

4.2. Performance on SP^{2}Bench

The SP^{2}Bench benchmark consists of a set of queries that cover a variety of query types and complexities, ranging from simple triple pattern matching to complex queries with subqueries and aggregates. The queries are designed to test different aspects of a system's performance, such as query processing time, query planning time, and data loading time.

We used 100 million triples generated using SP^{2}Bench on five queries listed at https://github.com/bapichatterjee/TripleBit-SplineIndex/blob/master/SPARQL-Queries.txt. Table 2 shows the results after benchmarking SP^{2}Bench on TripleBit with SplineIndex (TBSP), Apache Jena, TripleBit, and RDF3x.

Stores	Q1	Q2	Q3	Q4	Q5
TBSP	**0.0001**	**53.73**	**52.55**	**268.3**	**0.0041**
Apache Jena	0.12	1210.37	102.38	498.67	0.3247
TripleBit	0.00021	61.07	68.28	277.30	0.0870
RDF3X	0.0027	385.26	68.20	347.0	0.0012

Table 2
Performance of triple stores (time in seconds) on 100M triples from SP^{2}Bench. TBSP stands for TripleBitWithSplineIndex. The best runtimes are in bold.

From Tables 1, and 2, we can observe that TripleBitWithSplineIndex (TBSP) performs better than the other three triple stores, including TripleBit with the standard indexing scheme. This shows the potential of using a learned index for triple stores.

5. Conclusion and Future Work

There are several applications of Knowledge Graphs across different domains. Most often, Knowledge Graphs are very large in size. Querying over such large Knowledge Graphs is time consuming. Although several techniques for optimization exist, to the best of our knowledge, we are the first to explore the use of learned indexes for KG stores. Our first attempt in this work to integrate learned indexes (RadixSpline) with RDF stores is encouraging. We addressed the challenges arising in replacing the classical indexes in the pipeline with learned indexes. The code for our learned index based triple store is available at https://github.com/bapichatterjee/TripleBit-SplineIndex.

As a next step, we plan to use more recent and advanced learned index structures for triple stores. The literature on learned indexes has been enriched by the benchmarks at different points. For example, see [27], [28], and [13]. The existing benchmarks provide insights to efficacy of the learned indexes for standard query tasks for real numbers. Given that RDF queries involve multiple and potentially repeated join operations, it is a natural extension of this work to benchmark the different learned indexes such as ALEX [29], FITing-tree [30], the PGM-index [31], etc. for the task of SPARQL queries by including these indices in our architecture.

Building on the success of indexes for one-dimensional real-number, subsequent works extended learned indexes to domain specific queries such as multidimensional indexes for correlated and skewed data [32], learned indexes for KNN queries [33], indexes for string queries [34], and learned index for large-scale dense passage retrieval [35]. It is imperative to explore the efficient design of learned indexes for RDF stores.

6. Acknowledgements

This work was partially supported by the Infosys Center for AI (CAI), IIIT-Delhi, India.

7. Declaration on Generative AI

The authors acknowledge the use of Generative AI tools (ChatGPT) to assist in text refinement and language editing. All generated content was critically assessed and validated by the authors. The final wording reflects the author's own judgment.

References

[1] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen,

J. Sequeda, S. Staab, A. Zimmermann, Knowledge graphs, ACM Computing Surveys 54 (2021). URL: https://doi.org/10.1145/3447772. doi:10.1145/3447772.

[2] N. Guarino, D. Oberle, S. Staab, What Is an Ontology?, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–17. URL: https://doi.org/10.1007/978-3-540-92673-3_0. doi:10.1007/978-3-540-92673-3_0.

[3] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, J. Taylor, Industry-scale knowledge graphs: lessons and challenges, Communications of the ACM 62 (2019) 36–43. URL: https://doi.org/10.1145/3331166. doi:10.1145/3331166.

[4] D. Vrandečić, L. Pintscher, M. Krötzsch, Wikidata: The making of, in: Companion Proceedings of the ACM Web Conference 2023, WWW '23 Companion, Association for Computing Machinery, New York, NY, USA, 2023, p. 615–624. URL: https://doi.org/10.1145/3543873.3585579. doi:10.1145/3543873.3585579.

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

[6] T. Rebele, F. Suchanek, J. Hoffart, J. Biega, E. Kuzey, G. Weikum, Yago: A multilingual knowledge base from wikipedia, wordnet, and geonames, in: The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part II, Springer-Verlag, Berlin, Heidelberg, 2016, p. 177–185. URL: https://doi.org/10.1007/978-3-319-46547-0_19. doi:10.1007/978-3-319-46547-0_19.

[7] A. Bonifati, W. Martens, T. Timm, An analytical study of large SPARQL query logs, VLDB J. 29 (2020) 655–679. URL: https://doi.org/10.1007/s00778-019-00558-9. doi:10.1007/S00778-019-00558-9.

[8] M. Schmidt, M. Meier, G. Lausen, Foundations of sparql query optimization, in: Proceedings of the 13th International Conference on Database Theory, ICDT '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 4–33. URL: https://doi.org/10.1145/1804669.1804675. doi:10.1145/1804669.1804675.

[9] A. Gubichev, T. Neumann, Exploiting the query structure for efficient join ordering in SPARQL queries, in: S. Amer-Yahia, V. Christophides, A. Kementsietsidis, M. N. Garofalakis, S. Idreos, V. Leroy (Eds.), Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014, OpenProceedings.org, 2014, pp. 439–450. URL: https://doi.org/10.5441/002/edbt.2014.40. doi:10.5441/002/EDBT.2014.40.

[10] K. Rabbani, M. Lissandrini, K. Hose, Optimizing SPARQL queries using shape statistics, in: Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, F. Guerra (Eds.), Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021, OpenProceedings.org, 2021, pp. 505–510. URL: https://doi.org/10.5441/002/edbt.2021.59. doi:10.5441/002/EDBT.2021.59.

[11] T. Kraska, A. Beutel, E. H. Chi, J. Dean, N. Polyzotis, The case for learned index structures, in: Proceedings of the 2018 international conference on management of data, 2018, pp. 489–504.

[12] P. Ferragina, F. Lillo, G. Vinciguerra, Why are learned indexes so effective?, in: International Conference on Machine Learning, PMLR, 2020, pp. 3123–3132.

[13] Z. Sun, X. Zhou, G. Li, Learned index: A comprehensive experimental evaluation, Proceedings of the VLDB Endowment 16 (2023) 1992–2004.

[14] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, T. Neumann, Radixspline: A single-pass learned index (2020).

[15] A. Davitkova, E. Milchevski, S. Michel, The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries., in: EDBT, 2020, pp. 407–410.

[16] Q. Liu, M. Li, Y. Zeng, Y. Shen, L. Chen, How good are multi-dimensional learned indexes? an experimental survey, The VLDB Journal 34 (2025) 17.

[17] P. Li, H. Lu, Q. Zheng, L. Yang, G. Pan, Lisa: A learned index structure for spatial data, in: Proceedings of the 2020 ACM SIGMOD international conference on management of data, 2020, pp. 2119–2133.

[18] P. Ferragina, F. Lillo, G. Vinciguerra, On the performance of learned data structures, Theoretical Computer Science 871 (2021) 107–120.
[19] M. Mitzenmacher, A model for learned bloom filters, and optimizing by sandwiching, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, 2018, p. 462–471.
[20] I. Sabek, K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, T. Kraska, Can learned models replace hash functions?, Proceedings of the VLDB Endowment 16 (2022) 532–545. URL: https://doi.org/10.14778/3570690.3570702.
[21] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, H. Chen, Xindex: a scalable learned index for multicore data storage, in: Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming, 2020, pp. 308–320.
[22] G. Bhardwaj, B. Chatterjee, A. Sharma, S. Peri, S. Nayak, Kanva: A lock-free learned search data structure, in: Proceedings of the 53rd International Conference on Parallel Processing, 2024, pp. 252–261.
[23] A. S. Foundation, Apache jena (2021). URL: https://jena.apache.org/.
[24] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, Triplebit: A fast and compact system for large scale rdf data, Proceedings of the VLDB Endowment 6 (2013) 517–528. doi:10.14778/2536349.2536352.
[25] T. Neumann, G. Weikum, Rdf3x: a risc-style engine for rdf, Proceedings of The Vldb Endowment - PVLDB 1 (2008) 647–659. doi:10.14778/1453856.1453927.
[26] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, SP^2Bench: A sparql performance benchmark, in: IEEE 25th International Conference on Data Engineering, 2009, pp. 222–233. doi:10.1109/ICDE.2009.28.
[27] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, T. Neumann, Sosd: A benchmark for learned indexes, arXiv preprint arXiv:1911.13014 (2019).
[28] L. Bindschaedler, A. Kipf, T. Kraska, R. Marcus, U. F. Minhas, Towards a benchmark for learned systems, in: 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW), IEEE, 2021, pp. 127–133.
[29] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, et al., Alex: an updatable adaptive learned index, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 969–984.
[30] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, T. Kraska, Fiting-tree: A data-aware index structure, in: Proceedings of the 2019 international conference on management of data, 2019, pp. 1189–1206.
[31] P. Ferragina, G. Vinciguerra, The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds, Proceedings of the VLDB Endowment 13 (2020) 1162–1175.
[32] J. Ding, V. Nathan, M. Alizadeh, T. Kraska, Tsunami: A learned multi-dimensional index for correlated data and skewed workloads, arXiv preprint arXiv:2006.13282 (2020).
[33] Y. Peng, Lk-index: A learned index for knn queries, IEEE Access (2024).
[34] P. Ferragina, M. Frasca, G. C. Marinò, G. Vinciguerra, On nonlinear learned string indexing, IEEE Access (2023).
[35] Y. Wang, H. Ma, D. Z. Wang, Lider: an efficient high-dimensional learned index for large-scale dense passage retrieval, Proceedings of the VLDB Endowment 16 (2022) 154–166.