Expert Systems With Applications 279 (2025) 127384

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

RDF-TDAA: Optimizing RDF indexing and querying with a trie based on Directly Addressable Arrays and a path-based strategy

Yipeng Liu [a], Yuming Lin [a,*], Xinyong Peng [b], You Li [a], Jingwei Zhang [a]

[a] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Jinji Road, Guilin, 541004, Guangxi, China
[b] Guangxi Key Laboratory of Digital Infrastructure, Guangxi Zhuang Autonomous Region Information Center, Nanning, 530000, Guangxi, China

ARTICLE INFO

Keywords:
RDF data
Knowledge graphs
Indexing
Worst-case optimal joins

ABSTRACT

The rapid expansion of RDF knowledge graphs in scale and complexity poses significant challenges for optimizing storage efficiency and query performance, with existing solutions often limited by high storage costs or slow retrieval speeds. This study introduces RDF-TDAA, a novel RDF data management engine built on an advanced trie-based index that integrates Directly Addressable Arrays, Characteristic Sets, and integer sequence compression to achieve exceptional data compactness while maintaining high-speed query processing. RDF-TDAA also employs a unique path-based query planning approach, which constructs efficient execution plans based on the paths in query graphs, and integrates a worst-case optimal join algorithm to further streamline query processing. To validate our approach, we conducted extensive experiments using both synthetic and real-world datasets. The results demonstrate that RDF-TDAA surpasses leading RDF management systems in both storage efficiency and query speed. These findings underscore RDF-TDAA's scalability and effectiveness as a robust solution for managing large-scale RDF knowledge graphs, with valuable implications for improving RDF data handling in both academic and practical applications. The code for RDF-TDAA is available at https://github.com/MKMaS-GUET/RDF-TDAA.

1. Introduction

Knowledge graphs have become a fundamental infrastructure for numerous intelligent applications, and RDF has emerged as one of the most important representations for knowledge graphs. The Resource Description Framework (RDF)[1] and SPARQL Protocol and RDF Query Language (SPARQL)[2] are foundational standards established by the World Wide Web Consortium (W3C) for representing and querying structured data on the Semantic Web.

RDF enables data to be organized as interconnected triples (subject, predicate, object), creating rich Knowledge Graphs (KGs) that span various domains. Several KGs, expressed in RDF, integrate data from multiple sources, ranging from general-purpose KGs such as DBpedia (Lehmann et al., 2015) and Wikidata (Vrandecic & Krötzsch, 2014) to domain-specific semantic repositories like ORKG (Jaradeh et al., 2019) for scholarly works, and Tweetscov19 (Dimitrov et al., 2020) for COVID-19-related datasets. In recent years, KGs have also been employed for validating and enriching the responses of Large Language Models, such as those demonstrated in Mountantonakis and Tzitzikas (2023). Furthermore, RDF has found practical applications in the integration of Internet of Things data, where RDF engines serve as gateways for semantic integration (Guo, Le-Tuan, & Phuoc, 2023). As these KGs continue to expand in both size and complexity, there is a growing demand for efficient solutions that optimize storage and enhance query performance. RDF-TDAA addresses this need by providing a scalable, high-performance indexing mechanism and an effective query planning strategy, making it well-suited for real-world applications in knowledge management, AI-driven semantic reasoning, and large-scale data integration.

In querying RDF data with SPARQL, joins, particularly multi-way joins, are fundamental operations when querying RDF data with SPARQL. Traditional pairwise join strategies often generate substantial intermediate results, resulting in inefficiencies that far exceed the size of the final output. To address these limitations, worst-case optimal join algorithms, such as Leapfrog TrieJoin (LTJ) (Veldhuizen, 2014), have been developed. These algorithms guarantee query performance proportional to the Atserias–Grohe–Marx bound, which establishes a theoretical upper limit on the maximum possible number of solutions for natural join queries, calculated based on the table sizes and how these tables are linked together through shared columns.

The worst-case optimal join algorithms process joins by traversing query variables in a specific order, incrementally eliminating variables

* Corresponding author.
E-mail addresses: 23032303001@mails.guet.edu.cn (Y. Liu), ymlin@guet.edu.cn (Y. Lin), pengxy@gxi.gov.cn (X. Peng), liyou@guet.edu.cn (Y. Li), gtzjw@guet.edu.cn (J. Zhang).

https://doi.org/10.1016/j.eswa.2025.127384
Received 22 December 2024; Received in revised form 9 March 2025; Accepted 20 March 2025

by substituting them with candidate values. Although these algorithms maintain its worst-case optimality guarantees regardless of the elimination order, the choice of variable order significantly impacts performance, akin to traditional query plans (Veldhuizen, 2014). To facilitate faster join processing, we propose a path-based query planning strategy that constructs optimized execution plans by analyzing paths in the query graph. This approach effectively reduces the query search space, leading to improved query performance in most cases.

For worst-case optimal join algorithms to achieve worst-case optimality, it is crucial to efficiently enumerate the candidate values that eliminate a variable. Existing indexing approaches supporting worst-case optimal join algorithms, however, face notable limitations. Some methods utilize redundant indexes that consume substantial storage space to effectively support worst-case optimal join algorithms (Bigerl et al., 2020; Hogan, Riveros, Rojas, & Soto, 2019). Others adopt compact index structures but rely heavily on expensive random access operations, necessitating that the indexes remain entirely in memory to mitigate the performance impact of such accesses (Arroyuelo et al., 2021).

Trie-based indexing offers an elegant solution for supporting worst-case optimal joins, as it enables efficient prefix-based searches while minimizing redundant prefixes in RDF data. However, applying trie structures to large RDF datasets poses challenges due to the inherent sparsity of RDF data, which results in storage inefficiencies. Methods like Header-Dictionary-Triples (Fernández, Martínez-Prieto, & Gutierrez, 2010) attempt to address these challenges by replacing pointers with bitmaps, though this approach can lead to slower retrieval operations. Similarly, the Permuted Trie Index (Perego, Pibiri, & Venturini, 2020) employs advanced compression techniques and cross-compression to reduce storage requirements but suffers from inefficiencies when decoding values from the trie.

To better support worst-case optimal joins, we introduce a novel trie-based index for RDF data. This index combines Characteristic Sets (CS) with Directly Addressable Arrays (DAA). A characteristic set of a subject consists of the predicates associated with it, and multiple subjects can share the same characteristic set. The use of characteristic sets minimizes duplicate values, while the DAA, as a variant of the Directly Addressable Codes (DAC), employs only two extra bitmaps to enable efficient random access to variable-length arrays. This reduces the need for numerous pointers in the trie structure. This combined approach addresses the inefficiencies inherent in existing trie-based structures, offering a compact and effective solution for RDF indexing.

The main contributions of this work can be summarized as follows:

- **Optimized Trie with CSs and DAAs**: We design a trie-based index structure that integrates characteristic sets and DAAs to improve storage efficiency and retrieval performance. By eliminating duplicate values and reducing the number of pointers in the trie structure, RDF-TDAA achieves a highly compact RDF index while maintaining fast query processing speeds.
- **Path-Based Query Planning**: We propose a plan generation strategy for worst-case optimal join algorithms by analyzing paths in the SPARQL query graph. By ordering variables based on their positions in paths, frequency of occurrence, and estimated cardinality, the engine effectively reduces the search space and enhances query performance.s
- **Experimental Verification**: We conduct extensive experiments on synthetic and real-world datasets, demonstrating that RDF-TDAA outperforms existing RDF indexing solutions in both storage efficiency and query execution speed. The results validate RDF-TDAA's scalability and effectiveness in managing large RDF datasets, establishing it as a robust solution for executing SPARQL queries.

The remainder of this paper is structured as follows: Section 2 provides a review of related work in RDF management engines. Section 3 introduces key concepts and techniques used in RDF-TDAA, including DAC and characteristic set. Section 4 details the RDF-TDAA indexing architecture, and Section 5 describes the query processing mechanisms. Section 6 outlines the implementation details, while Section 7 presents experimental results comparing RDF-TDAA with existing RDF management engines and indexing approaches. Finally, Section 8 concludes the paper and discusses future research directions.

2. Related work

In recent years, various query engines for RDF data have emerged. These engines vary not only in their indexing strategies but also in the query execution approach they deploy to efficiently handle SPARQL queries.

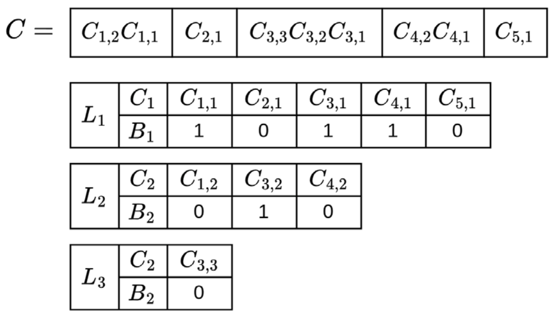
A common approach is to index various permutations of the RDF triple components, namely Subject (S), Predicate (P), and Object (O), such as SPO, SOP, PSO, POS, OSP, and OPS. This method is adopted by RDF-3X (Neumann & Weikum, 2010), a widely used RDF store, which employs compressed indexes based on clustered B+ trees to index all six permutations. Similarly, Hexastore (Weiss, Karras, & Bernstein, 2008) uses adjacency lists to store RDF triples and indexes all six permutations. Blazegraph (Thompson, Personick, & Cutcher, 2014) supports indexing triples and quads. For triples, Blazegraph indexes three permutations, while for quads, it indexes six permutations. In contrast, Virtuoso (Erling & Mikhailov, 2009) uses a combination of full and partial indexes. It stores RDF data in a quad table and employs five indexes (PSOG, POGS, SP, OP, and GS) to facilitate efficient querying.

Vertical partitioning is an effective way to materialize permutations, especially the PSO permutation. For example, in the PSO permutation, a two-column table is created for each predicate, listing all (subject, object) pairs. The k^2-triples (Álvarez-García, Brisaboa, Fernández, Martínez-Prieto, & Navarro, 2015) system uses this approach, employing a k^2-tree structure to index the subjects and objects associated with each predicate.

Graph-based engines leverage the structure of RDF graphs in various ways. gStore (Shen et al., 2015) uses adjacency lists and an extended signature tree, called a VS*-tree, to store RDF graphs and solves queries through efficient subgraph matching algorithms. In contrast, AMBER (Ingalalli, Ienco, Poncelet, & Villata, 2016) represents RDF graphs as multigraphs, where IRIs are nodes and (predicate, literal) pairs serve as attributes for those nodes. It then creates three indexes: the first stores the set of nodes for each attribute, the second stores vertex signatures that encode metadata about the triples where a given node is subject or object, and the third stores adjacency lists.

The engines mentioned above rely on traditional pairwise or multiway join algorithms. More recent approaches have explored worst-case optimal join algorithms to enhance query processing efficiency. However, classical indexing techniques face limitations when all six permutations of RDF triples need to be indexed. One such system, Jena-LTJ (Hogan et al., 2019), implements a worst-case optimal join algorithm within a triple store that indexes all permutations,resulting in high storage space requirements. To address these challenges, a novel index structure called the Ring (Arroyuelo et al., 2021) has been proposed, leveraging a variant of the Burrows-Wheeler Transform to index RDF data efficiently.

Additionally, recent research has explored matrix- and tensor-based storage methods for RDF data, such as Qdags (Navarro, Reutter, & Rojas-Ledesma, 2020) and Tentris (Bigerl, Conrads, Behning, Saleem and Ngomo, 2022; Bigerl et al., 2020), both of which leverage worst-case optimal join algorithms. Qdags extend quadtree structures to store RDF data in matrix form, while Tentris employs sparse order-3 tensors to represent RDF graphs and utilizes a trie-like data structure called hypertries to support tensor slice operations. Other engines have adopted matrix algebra for querying RDF data, including MAGiQ (Jamour, Abdelaziz, Chen, & Kalnis, 2019) and gSmart (Chen, Özsu, Xiao, Tang, & Li, 2021). MAGiQ stores RDF data as sparse matrices and translates



$C =$

$C_{1,2}C_{1,1}$	$C_{2,1}$	$C_{3,3}C_{3,2}C_{3,1}$	$C_{4,2}C_{4,1}$	$C_{5,1}$

L_1	C_1	$C_{1,1}$	$C_{2,1}$	$C_{3,1}$	$C_{4,1}$	$C_{5,1}$
	B_1	1	0	1	1	0

L_2	C_2	$C_{1,2}$	$C_{3,2}$	$C_{4,2}$
	B_2	0	1	0

L_3	C_2	$C_{3,3}$
	B_2	0

Fig. 1. An example of directly addressable codes.

queries into equivalent matrix algebra operations, which are then processed using existing matrix algebra libraries such as MATLAB and GraphBLAS. Similarly, gSmart stores RDF data in sparse matrices and converts join operations into sparse matrix multiplications, implementing these operations on both CPU and GPU platforms using CUDA. These approaches highlight the growing interest in leveraging matrix and tensor-based techniques to enhance the efficiency of RDF data storage and querying.

As knowledge graphs continue to grow in size and complexity, centralized RDF engines struggle to meet the demands of modern applications, prompting a shift toward distributed solutions. For instance, Wukong (Shi, Yao, Chen, Chen, & Li, 2016) leverages distributed key–value storage and Remote Direct Memory Access (RDMA) to optimize RDF query processing, while its successor, Wukong+G (Yao, Chen, Zang, & Chen, 2022), extends this framework by integrating GPU acceleration and memory optimizations for enhanced performance in distributed environments. Another notable approach is DIAERESIS (Troullinou, Agathangelos, Kondylakis, Stefanidis, & Plexousakis, 2024), which improves query efficiency through a data partitioning strategy that identifies schema node centroids, hierarchically assigns dependent nodes, and applies vertical sub-partitioning to minimize query overhead. Additionally, the P2P-based RDF Engine (Guo et al., 2023) adapts Peer-to-Peer (P2P) principles for edge computing, addressing challenges such as node heterogeneity and resource constraints. By integrating P-Grid indexing and RDF4Led storage, it enables distributed RDF processing and load balancing across low-cost edge devices. These advancements underscore the ongoing efforts to develop scalable and efficient solutions for managing large-scale RDF data in distributed settings.

3. Preliminaries

3.1. Directly addressable codes

Variable-length coding is fundamental to data compression, enabling shorter codewords for frequent symbols and efficiently encoding integers by using fewer bits for smaller values. A major challenge, however, is the lack of direct access to the ith encoded element, as its location is depends on the cumulative lengths of prior codewords.

To address this, Nieves et al. propose Directly Addressable Codes (DAC) (Brisaboa, Ladra, & Navarro, 2013), a variable-length encoding scheme that allows for fast, direct element access with minimal space overhead while preserving compression benefits.

For a sequence of variable-length codewords C, DACs divide each codeword into multiple b-bit chunks, distributed across distinct levels. The nth level, denoted C_n, contains the nth chunk of each codeword. A corresponding bitmap B_n for each level marks, with a 0-bit, the end of each codeword chunk sequence at that level. An illustrative example of directly addressable codes is provided in Fig. 1.

To retrieve the ith value in the sequence, the process begins with $i_1 = i$. The first chunk, $C_{i,1} = C_1[i_1]$, is accessed. If $B_1[i_1] = 0$, $C_1[i_1]$ represents the entire value, terminating the retrieval. Otherwise, the index advances to $i_2 = \text{rank}(B_1, i_1)$, where $\text{rank}(B, i)$ calculates the count of 1-bits in $B[1, i]$, providing the correct position for the subsequent chunk in C_2. The next chunk $C_{i,2} = C_2[i_2]$ is then accessed. If $B_2[i_2] = 0$, the completed value is calculated as $C_1[i_1] + C_2[i_2] \times 2^b$. If not, the process iterates with each $i_n = \text{rank}(B_{n-1}, i_{n-1})$, continuing until the last chunk is retrieved.

3.2. Characteristic sets

The concept of Characteristic Set (CS) was introduced by Neumann and Moerkotte (Neumann & Moerkotte, 2011) as a method to capture the structural essence of an RDF dataset. A characteristic set identifies node types by the properties they emit. Formally, given a set of triples T and a node s, the characteristic set $S_c(s)$ of s is defined as:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in T\} \tag{1}$$

The collection of all characteristic sets for triples in T is expressed as:

$$S_c(T) = \{S_c(s) \mid \exists p, o : (s, p, o) \in T\} \tag{2}$$

Characteristic sets provide a node-centric partitioning of RDF data sets grounded in the structural properties associated with each node. This concept has proven valuable in optimizing join operations and estimating cardinality, thereby improving the efficiency of RDF data management and query optimization.

3.3. Worst-case optimal joins

The Atserias–Grohe–Marx (AGM) bound (Atserias, Grohe, & Marx, 2013) establishes an upper limit on the number of solutions for natural join queries $Q = r_1 \bowtie \cdots \bowtie r_m$, where $r_1, \ldots, r_m$ are distinct relation names. Given a natural join query Q and a relational instance D, the AGM bound for Q over D specifies the maximum number of tuples that can be generated by evaluating Q on any instance D' in which the size of each relation does not exceed that of the corresponding relation in D. Assuming all relations have size $O(n)$, the AGM bound of Q, denoted as Q^*, can be defined as a function of n.

Algorithm 1 Leapfrog TrieJoin

```
   Input: List of variables $V$, Triple patterns $T$, an index $0 \le idx \le |V|$
   of $V$, and a variable mapping $\mu$ defined for the variables $\{v_k | k < idx\}$
   Output: A solution $\mu$ for $T$
1: if $idx = |V|$ then return $\mu$
2: else
3:     while $(c \leftarrow \text{Search}(V, T, \mu, idx)) \neq -1$ do
4:         $\mu' \leftarrow \mu \cup \{(V[idx] := c)\}$
5:         LEAPFROG_TRIEJOIN$(V, T, \mu, idx + 1)$
6:     end while
7: end if
```

An algorithm is defined as worst-case optimal if it operates within time $O(Q^*)$, where Q^* is the upper bound on the number of solutions. While some algorithms achieve $O(Q^*)$ complexity, an additional logarithmic factor of $O(Q^* \log n)$ is often permissible to enable flexible operations, such as binary search on sorted relations instead of hashing. Traditional pairwise join algorithms do not meet the worst-case optimal criteria, but algorithms like Leapfrog TrieJoin (LTJ) (Veldhuizen, 2014) approach near-optimal performance with a complexity of $O(Q^* \log n)$.

In contrast to pairwise join algorithms that evaluate joins one relation at a time, LTJ operates by joining on one variable at a time. Algorithm 1 demonstrates LTJ's approach to evaluating triple patterns T. LTJ utilizes a sorted variable list V as its execution plan and outputs

Algorithm 2 Search

```
    Input: List of variables $V$, Triple patterns $T$, an index $0 \le idx \le |V|$ of $V$, and a variable mapping $\mu$ defined for the variables $\{v_k | k < idx\}$
    Output: Next join result for variable $V[idx]$
 1: Initialize iterators $iter_0 \dots iter_n$ for triple patterns in $T$ with $V[idx]$ using $\mu$
 2: Sort iterators ascendingly by their current values
 3: $max \leftarrow iter_n$.Current(), $i \leftarrow 0$
 4: while true do
 5:     $v \leftarrow iter_i$.Current()
 6:     if $v = max$ then return $v$
 7:     else $iter_i$.Seek($max$)
 8:     end if
 9:     $max \leftarrow iter_i$.Current()
10:     $i \leftarrow (i + 1)\%(n + 1)$
11: end while
12: return -1
```

variable mappings μ as solutions. For each variable, LTJ recursively retrieves bindings through the Search operation, as outlined in Algorithm 2. The Search operation generates iterators for triple patterns containing a specific variable, where these iterators provide candidate values for the variable to identify the next join result for $V[idx]$. The Seek operation is essential in this process, advancing the iterator to the smallest value that is at least as large as a given parameter.

4. The proposed index structure

In this section, we present the index structure of RDF-TDAA. Section 4.1 provides an overview of the index architecture. Section 4.2 introduces the basic trie index for RDF data. Section 4.3 discusses how CSs are integrated into the trie to enhance indexing efficiency. Finally, Section 4.4 explains the incorporation of DAAs to optimize the trie structure for storage and retrieval performance.

4.1. Index architecture overview

To accommodate the six distinct triple selection patterns, each featuring one or two wildcard symbols, specifically (S, ?, ?), (?, P, ?), (?, ?, O), (S, P, ?), (S, ?, O), and (?, P, O), we index two permutations of RDF: SPO and OPS. In a trie-based index, triples are hierarchically structured, enabling efficient range queries and prefix-based lookups. The SPO trie organizes triples with subjects (S) at the first level, predicates (P) at the second, and objects (O) at the third, making it well-suited for queries that specify a subject, such as (S, ?, ?) and (S, P, ?). However, due to the hierarchical nature of tries, queries that do not specify a subject – like (?, P, O) and (?, ?, O) – require scanning all subjects, leading to inefficiencies. To address this, we index the OPS permutation, which indexes objects (O) first, followed by predicates (P) and subjects (S). This structure optimizes lookup performance for object-based queries, significantly improving efficiency for patterns such as (?, P, O) and (?, ?, O). By leveraging both SPO and OPS permutations, we can efficiently support the (S, ?, O).

Additionally, although both permutations can support the (?, P, ?) pattern, their efficiency is limited in this case. To enhance performance, we introduce an additional index referred to as the predicate index.

The predicate index retains all subjects and objects associated with each predicate, as illustrated in Fig. 2. For each predicate, the subject and object arrays are sorted, and duplicates are removed. An offset array is needed to keep track of the starting offsets of each subject and object array. For each predicate, the offset array contains a pair of offsets: one for the start of the subject array and one for the start of the object array.

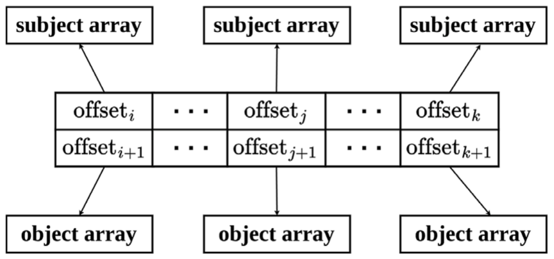


Fig. 2. Structure of predicate index.

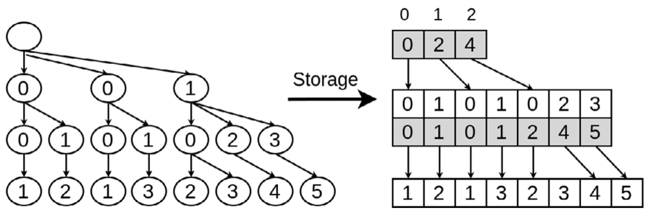


Fig. 3. Using trie to index RDF triples.

In conclusion, the index architecture of RDF-TDAA consists of three primary components: the trie index for the SPO permutation, the trie index for the OPS permutation, and the predicate index. The predicate index has been described above, while the trie indexes are detailed in the following sections, with the SPO permutation used as an example. The OPS permutation is indexed in a similar manner, with the roles of subject and object reversed.

4.2. Trie-based indexing for RDF data

The left side of Fig. 3 provides an example for the SPO permutation index for RDF data, indexing the following encoded triples: $(0, 0, 1)$, $(0, 1, 2)$, $(1, 0, 1)$, $(1, 0, 3)$, $(1, 2, 4)$, $(1, 3, 5)$. This trie structure is organized into three hierarchical levels: the first level represents subjects, the second level represents predicates, and the third level represents objects. The nodes at each level are sorted according to the respective component of the RDF triples, enabling structured traversal and efficient query processing.

The right side of Fig. 3 shows the storage layout of this trie structure. The stored trie consists of node arrays, which hold all nodes at a given level, and pointer arrays (indicated by shaded regions). Each pointer represents the offset to the corresponding node array at the next level, indicating the starting position of each node's child sequence. In Fig. 3, nodes at the first level are implicit, as each subject's ID directly serves as an offset to its respective entry in the pointer array. Therefore, first-level nodes are not explicitly stored in the data structure but are displayed in a smaller font in Fig. 3 for illustration purposes.

Traditional trie structures are not well-suited for large-scale RDF datasets due to their substantial space cost. As depicted on the right side of Fig. 3, each trie level involves storing both node arrays and pointer arrays, resulting in high storage overhead. This is particularly inefficient for large, sparsely distributed datasets, where even nodes with few children require dedicated storage for pointers.

4.3. Enhancing the trie with characteristic sets

To improve the trie structure in Fig. 3, a simple yet effective approach is to utilize Characteristic Sets (CS). As illustrated in Fig. 3, at the second level of the trie, subjects 0 and 1 share the same predicate set, $\{0, 1\}$. This redundancy can be eliminated by incorporating characteristic sets into the trie structure for the SPO permutation. For the OPS

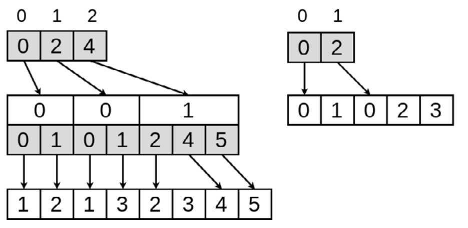


Fig. 4. An example of using characteristic sets in trie.

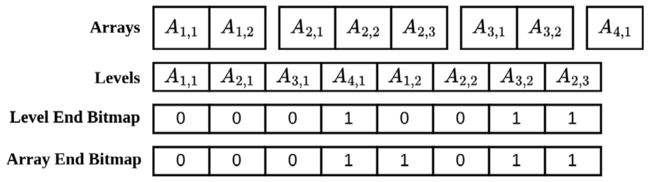

Arrays
$A_{1,1}$ $A_{1,2}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,1}$ $A_{3,2}$ $A_{4,1}$
Levels
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$ $A_{4,1}$ $A_{1,2}$ $A_{2,2}$ $A_{3,2}$ $A_{2,3}$
Level End Bitmap
0 0 0 1 0 0 1 1
Array End Bitmap
0 0 0 1 1 0 1 1

Fig. 5. An example of DAAs.

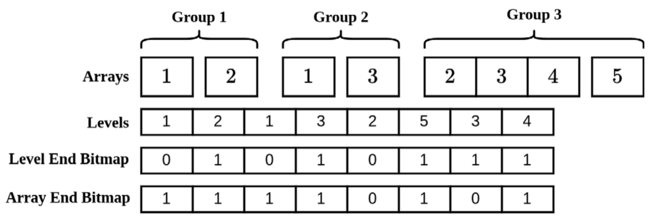


Fig. 6. Merging multiple DAA structures.

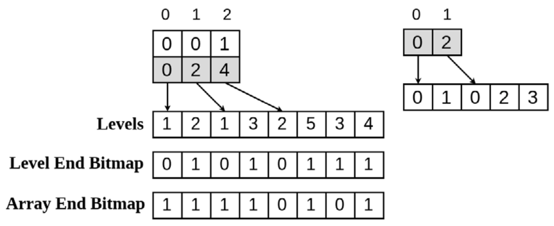


Fig. 7. Utilizing DAAs to enhance the trie with characteristic sets.

permutation, the Reverse Characteristic Set (RCS) should be employed, which are defined as follows:

$$S_{rc}(o) = \{p \mid \exists s : (s,p,o) \in T\} \tag{3}$$

A statistical evaluation demonstrating the effectiveness of integrating CSs and RCSs into the trie structure is provided in Appendix. By replacing redundant predicate sets with CSs or RCSs, the storage requirements for both the SPO and OPS permutations can be significantly reduced. This improvement is illustrated in Fig. 4.

The left side of Fig. 4 shows a trie structure enhanced with characteristic sets, where all predicate sets are replaced by their corresponding characteristic set IDs. On the right side of Fig. 4, the characteristic set index is depicted, storing all characteristic sets in a compact array. This array is accessed using an auxiliary pointer array that records the starting offsets of each characteristic set. By organizing data in this manner, the trie structure achieves improved storage efficiency while maintaining high retrieval performance.

4.4. *Optimizing the trie further with directly addressable arrays*

The use of characteristic sets effectively reduces duplicates in the node array. However, as shown in Fig. 4, a significant number of pointers still remain in the trie structure. To address this issue, we propose Directly Addressable Arrays (DAA), a variant of Directly Addressable Codes (DAC). DAAs enhance random access across multiple arrays while incurring minimal storage and retrieval overhead.

Let A denote a sequence of arrays, where $A_{i,j}$ represents the jth element of the ith array in A. DAAs store A compactly by distributing elements across levels, similar to the DACs. As illustrated in Fig. 5, each element of an array is distributed among different levels, but all levels are stored in a single array called the *Levels Array*, denoted as L. To enable efficient random access, two additional bitmaps are introduced: the *Array End Bitmap*, denoted as B_a, where a 1-bit at position i marks the ith element in L as the final element of its corresponding array, and the *Level End Bitmap*, denoted as B_l, where the ith 1-bit signifies the end position of the ith level.

The retrieval of all elements in an array can be illustrated through an example. To retrieve the elements in the third array, $\{A_{3,1}, A_{3,2}\}$, we begin by determining the starting offset $l_1 = 0$ for the current level using B_l, which has an initial offset of 0 for level 0. From the calculated element offset $i_1 = l_1 + 2 = 2$, the first element, $A_{3,1}$, is retrieved using $L[i_1]$. Since the i_1th bit in the B_a is not 1, indicating that the array is not fully retrieved, the offset is then updated. We count the number of array endings within $B_a[l_1, i_1] = B_a[0, 2]$, which yielding 0. Then the level start offset l_2 for level 1 is determined to be 4. The offset for the next element is calculated as $i_2 = l_2 + 2 - 0 = 6$, and the second element, $A_{3,2}$, is retrieved from L. Upon checking the i_2th bit in B_a, a bit value of 1 is found, indicating that the final element of the array has been retrieved. As shown, the offsets of elements across levels are calculated as follows:

$$i_n = l_n + i_{n-1} - \mathrm{rank}(B_a, l_{n-1}, i_{n-1}) \tag{4}$$

Here, the rank operation calculates the number of 1-bits in the range $[l_{n-1}, i_{n-1}]$. The start offset l_n for the nth level is defined as:

$$l_n = \begin{cases} \mathrm{select}(B_l, n) + 1 & \text{if } n \neq 0 \\ 0 & \text{if } n = 0 \end{cases} \tag{5}$$

where the select operation locates the position of the nth 1-bit in B.

To illustrate the use of DAAs, we take the SPO permutation trie-based index, shown in Fig. 4, as an example. In this context, DAAs represent object arrays associated with a subject. The DAA structure for a subject organizes object arrays according to the order of predicates in the subject's characteristic set. In this way, predicates and their corresponding object arrays are directly linked. To retrieve the object array for the ith predicate in the characteristic set of a subject, we can directly access the same offset i in the DAA structure, ensuring efficient retrieval.

To optimize the trie structure using DAAs, multiple DAA structures for subjects should be merged, as illustrated in Fig. 6. At the third level of trie in Fig. 4, three distinct object array groups independently form separate DAA structures, which are then merged into a single structure. This unified structure merged all three levels array L into one array. Additionally, the bitmaps B_a and B_l are each merged into a single bitmap. The retrieval operation for this unified structure follows the same process as that of a standard DAA structure, with the added requirement that each offset calculation must consider the starting offset specific to the DAA structure being accessed.

The final trie structure, using the example from Fig. 4, is depicted in Fig. 7. The node array in the third level of trie is replaced with the unified DAA structure, and the pointer array that originally pointed to third-level nodes from the second level of the trie is replaced by an array that stores the starting offsets of the DAA structures. As a result, each subject is associated with a unique characteristic set ID and a pointer to the DAA structure. This configuration allows for direct access to the second level through the subject ID, obviating the need for offsets at the first level of the trie structure and thus promoting space optimization by eliminating the first level entirely.

The final trie structure, enhanced with DAAs and characteristic sets, reduces the storage overhead of the trie index without compromising the fast retrieval performance of the original trie structure. An analysis conducted on various datasets, provided in the Appendix, demonstrates that the computational overhead of retrieving an array in the DAA structure is minimal. This optimized trie structure serves as the foundation of RDF-TDAA, offering a compact and efficient index for RDF data management.

5. Index retrieval and update

5.1. Index retrieval

Index retrieval refers to the process of efficiently locating and accessing data stored in an index structure without scanning the entire dataset. It is a fundamental technique in database systems, enabling fast lookups, range queries, and join operations. In RDF-TDAA, this process is also crucial for query execution, which follows a method similar to the leapfrog triejoin algorithm, as shown in Algorithm 1. A key step in query execution is generating multiple iterators for a variable, where each iterator is derived from a triple pattern that contains the variable, leveraging the index of RDF-TDAA to efficiently retrieve relevant data. The retrieval process for iterators is detailed below.

Algorithm 3 Retrieve an object iterator

```
   Input: SPO-index $idx$, triple $tpl = (S, P, O)$
   Output: Iterator for objects
1: if $tpl.S$ exists in $idx$ then
2:     Locate the position $i$ of $tpl.P$ within the CS of $tpl.S$
3:     if $tpl.P$ is found then
4:         Access $i$th array in $tpl.S$'s DAA structure.
5:         return Iterator for the $i$th array
6:     end if
7: end if
8: return empty iterator
```

For one-variable triple patterns, such as (S, P, ?), (?, P, O), and (S, ?, O), distinct retrieval methods are used depending on the variable's position. When constructing the iterator for variable in (S, P, ?), as outlined in Algorithm 3, the process begins by verifying whether $tpl.S$ exists in the index idx (Line 1). If $tpl.S$ is present, its corresponding characteristic set is retrieved from the characteristic set index, and a binary search is performed to identify the position i of the predicate $tpl.P$ within the set (Line 2). If $tpl.P$ is found, the ith array in the DAA structure of S is accessed, enabling the construction of an iterator for the object variable (Lines 4–5). A similar procedure applies to (?, P, O), differing only in index permutation.

For the triple pattern (S, ?, O), we first retrieve the characteristic set for subject S and the reverse characteristic set for object O, and then calculate their intersection to determine the predicates shared by both S and O. Subsequently, we verify the presence of object O within the object array corresponding to subject S. Each valid predicate is then used to construct the predicate iterator for the predicate variable.

For two-variable triple patterns, such as (S, ?, ?), (?, ?, O), and (?, P, ?), retrievals vary based on the targeted variable. When obtaining iterators for the predicate variable in (S, ?, ?) or (?, ?, O), we directly retrieve the characteristic set for subject S or reverse characteristic set for object O. To obtain iterators for the object variable in (S, ?, ?) or the subject variable in (?, ?, O), the corresponding levels array L in the DAA structure is accessed directly, rather than retrieving each array individually. The values in this levels array L are then sorted and deduplicated to form the iterator. For cases involving (?, P, ?), where the iterators for subject variable or object variable related to a predicate P are required, the values are retrieved from the predicate index directly, thereby constructing the iterator.

5.2. Index update

Updating the RDF-TDAA index comprises two fundamental operations: the insertion and deletion of RDF triples. Both operations necessitate modifications to the trie index for the SPO and OPS permutations, as well as updates to the predicate index. The insertion process for the predicate index is relatively straightforward, as the subject and object arrays associated with each predicate are maintained in a sorted and deduplicated form. In contrast, the trie index requires a more intricate procedure. To illustrate, we consider the insertion of a triple (S, P, O) into the trie index corresponding to the SPO permutation. The deletion procedure is similar to the Algorithm 3, with the primary modification being the substitution of the iterator generation steps (Lines 4–5) with deletion operations.

Algorithm 4 Insert a RDF triple

```
    Input: SPO-index $idx$, triple $tpl = (S, P, O)$
 1: if $tpl.S$ exists in $idx$ then
 2:     Retrieve the characteristic set $cs$ of $tpl.S$
 3:     Insert $tpl.P$ within $cs$ and get its position $i$
 4:     if $tpl.P$ was newly inserted then
 5:         Insert $cs$ into the CS index
 6:         Update characteristic set ID of $tpl.S$
 7:         Insert array $\{tpl.P\}$ at position $i$ in the DAA
 8:     else
 9:         Insert $tpl.O$ into the $i$th array in the DAA
10:     end if
11:     Update DAA offsets in trie index
12: else
13:     Create a new characteristic set $\{tpl.P\}$
14:     Create a new DAA with $\{\{tpl.O\}\}$
15:     Insert $(id_{cs}, pos_{daa})$ into the trie
16: end if
```

The process of inserting a triple $tpl = (S, P, O)$ into the trie index is outlined in Algorithm 4. The process also begins by checking whether $tpl.S$ exists in the index idx (Line 1). If $tpl.S$ is found, its corresponding characteristic set is retrieved from the characteristic set index, then a binary search is performed to insert $tpl.P$ into the characteristic set (Lines 2–3) and get the position i of $tpl.P$ within the characteristic set.

If $tpl.P$ is newly inserted in the characteristic set, the updated characteristic set is inserted into the characteristic set index, and a new characteristic set ID is assigned to $tpl.S$ (Lines 5–6). A new array containing only $tpl.O$ is then added into the DAA structure at position i (Line 12). Conversely, if $tpl.P$ is already exists in the characteristic set, the object $tpl.O$ is inserted into the ith array within the DAA structure, provided that $tpl.O$ is not already present in the array (Line 9). To ensure consistency, the DAA structure offsets for subjects with IDs greater than $tpl.S$ are updated (Line 11).

If $tpl.S$ is not found in the trie, indicating it is a new subject, a new characteristic set $\{tpl.P\}$ is created and stored in the characteristic set index, while a new DAA structure is initialized with $tpl.O$ as its first element. The newly created characteristic set ID and DAA offset are then inserted into the trie, linking $tpl.S$ to its corresponding characteristic set and DAA structure (Lines 13–15).

The operation of inserting a value into a DAA structure constitutes the most computationally intensive component of this process. When a value is to be inserted into an array within the DAA structure, the algorithm iterates through the levels of the DAA structure to locate the first value in the array that exceeds the value being inserted. Upon identifying such a value, it is replaced with the value being inserted, and the displaced value is subsequently treated as the new value to be inserted. This procedure is repeated until the end of the array is reached. Finally, the displaced value at the last position is inserted into the next level of the DAA structure, and the two associated bitmaps are updated accordingly.

The time complexity of iterating through the levels is expressed as $O(\frac{m}{n})$, where n denotes the number of arrays in the DAA structure, and m represents the total number of values distributed across all arrays. The complexity associated with inserting the final value into the next level depends on the organization of the levels arrays within the different DAA structures. In the proposed design, all levels arrays are stored within a single array for simplicity. If the length of this array is l, the time complexity of inserting the final value is $O(l)$. This design could be easily optimized. For instance, the single array could be partitioned into several fixed-size chunks. Consequently, the overall time complexity for inserting a value into a DAA structure is $O(\frac{m}{n} + l)$.

6. Path-based query plan generation

We outline here the process of generating a query execution plan based on the paths of the query graph. The query graph, which is an undirected graph, contains only two-variable triple patterns, where two variables in each triple pattern are represented as connected nodes, and the sole constant serves as connecting edge.

The worst-case optimal join algorithms, such as leapfrog triejoin, process join queries by following a specific sequence of variables. For a given variable, LTJ leverages a variable mapping to replace variables with constants, retrieving a set of iterators for the triple patterns involving the variable and subsequently identifying a candidate result. The search space for this variable is closely influenced by the size of each iterator and the total number of iterators. To enhance the efficiency of join queries, two principles should be observed:

(1) The iterator size for a variable in a triple pattern can be reduced by increasing the number of constants in the triple pattern, which can be achieved by tracing paths in the query graph.
(2) Variables associated with a larger number of triple patterns should be prioritized earlier in the variable sequence, effectively reducing the search space in subsequent stages of the join query process.

Before identifying paths in the query graph, we first determine the starting node for these paths. This involves calculating the cardinality (i.e., the number of candidate values) for each variable within its respective triple patterns, as well as evaluating the frequency of each variable across all triple patterns. By combining these two factors, we estimate the candidate value count for each variable and arrange them in ascending order to establish the initial variable sequence.

This initial variable order is then used to construct a priority map, denoted as M, wherein each variable's position in the sequence is associated with a priority score. Lower values in M correspond to higher priority, enabling efficient retrieval of each variable's rank. The variable with the highest priority is then chosen as the starting node. Algorithm 5 provides the process for sorting variables based on their calculated priorities.

To begin variable sorting, we construct a query graph G from the two-variable triple patterns in all triple patterns T in a query. We then perform DFS on G, starting from the variable node v with the highest priority in M. Each path is added to the path list P once all neighbors of a node have been visited, continuing until all nodes are traversed. To facilitate traversal, an offset list I is initialized to zero for each path in P (Lines 1–5).

We then iterate over the offset list I, where each entry in I represents the current traversal position within its respective path in P. For each valid offset (i.e., an offset that has not yet reached the last variable in the path), the priority of the variable at the current offset in P is assessed. By comparing these priorities across paths, the variable x with the highest priority among the currently accessible nodes is identified (Line 6). This variable is appended to the sorted variable list V, and

Algorithm 5 Path-based Variable Sorting

```
    Input: Priority map $M$, Triple patterns $T$
    Output: Sorted variable list $V$
 1: Construct query graph $G$ from $T$
 2: Identify variable $v$ with the highest priority in $M$
 3: Perform DFS on $G$ from $v$ to generate paths $P$
 4: Initialize offset list $I$ for each path in $P$ with 0
 5: while offsets in $I$ are valid do
 6:     Select variable $x$ with the highest priority among current offsets
 7:     Append $x$ to $V$
 8:     Update offsets in $I$ for paths where the current variable is $x$
 9: end while
10: return $V$
```

the corresponding offsets in I are incremented for those paths where the current variable matches the one appended to V, advancing to the next variable node in that path (Lines 7–8).

This process proceeds iteratively, with each iteration updating the offsets and comparing variable priorities. The iteration continues until all entries in I become invalid (Line 5), signaling that all paths in P have been fully traversed except for the last variable nodes in each path. At this point, the sorted list V is finalized and returned.

After the path-based variable sorting, for one-variable and three-variable triple patterns, any pattern in which all variables are already present in the set V is excluded. Next, additional variables not yet included in V are appended. These comprise variables located at last nodes of paths in P as well as those in any remaining one-variable and three-variable triple patterns. The frequencies of these remaining variables are calculated, and they are subsequently added to the end of V in descending order of frequency.

The time complexity of Algorithm 5 can be analyzed as follows. Constructing the query graph G from the two-variable triple patterns in T (Line 4) requires $O(m)$ time, where m is the number of such triple patterns. Identifying the variable v with the highest priority in the priority map M (Line 5) takes $O(n)$ time, where n is the number of variables within these triple patterns. Performing a depth-first search (DFS) from v to traverse all nodes and edges in G (Line 6) has a time complexity of $O(n + m)$. Initializing the offset list I for each path in P (Line 7) is a straightforward operation, requiring $O(p)$ time, where p is the number of paths generated during DFS.

The primary computational burden of the algorithm lies in the iterative selection of variables based on their priority scores across multiple paths generated by the depth-first search (DFS) (Lines 5–9). In each iteration, the algorithm scans the offset list I, which has a size of p, to identify the variable with the highest priority among the current positions in all paths. Subsequently, it increments the offsets for those paths where the current variable matches the one appended to the sorted list V. This scanning process incurs a time complexity of $O(p)$ per iteration. Since each iteration adds exactly one variable to V, and all variables in the paths (except for the last variable node of each path) are eventually included, the number of iterations in the worst case is bounded by n, the total number of variables. Consequently, the overall time complexity of the main loop is $O(p \cdot n)$.

Combining these components, the total time complexity of the algorithm is: $O(m + n + p \cdot n)$.

7. System implementation

In this section, we describe the main engineering and implementation details of our RDF-TDAA engine. We implemented the RDF-TDAA engine in C++20, using g++ 12.3.0 with the -O3 optimization flag.

7.1. Compression

In the RDF-TDAA indexing architecture, there are many sorted integer sequences, such as those found in the predicate index and the characteristic set index. To handle these sequences efficiently, we apply integer sequence compression techniques.

One simple approach is to minimize the number of bits required to encode each integer by using $\lceil \log_2 mv \rceil$ bits per value, where mv is the maximum integer value in the sequence. This method, called "Compact", is effective due to its simplicity and efficiency, relying on basic bitwise operations for fast random access.

A more advanced compression technique is Variable Byte (VB) (Williams & Zobel, 1999) encoding, commonly used in information retrieval. This method divides a value into several fixed-size chunks. The highest bit of each chunk is used as a flag to indicate whether it is the last chunk. The remaining bits represent the number itself.

Both Compact and VB encoding methods are affected by the maximum value in the sequence. Since the sorted arrays in DAA structures, the characteristic set index, and the predicate index are typically accessed sequentially, we preprocess the integer arrays by calculating delta gaps. Instead of storing the actual values, we store the differences between consecutive values. This step reduces the maximum value in the array, allowing for more efficient encoding. When retrieving values from the compressed array, for example, with compact encoding, we decode the difference value, add it to the previous value, and reconstruct the actual value. This method is fast and efficient.

In RDF-TDAA, we apply gap and compact encoding to the sorted value arrays in the trie structure because they are frequently accessed sequentially during query processing. For the characteristic set index and predicate index, we use both gap and VB encoding. For offset arrays in the index, we only use compact encoding to improve retrieval speed, as offsets are often randomly accessed during query processing.

7.2. Implementation details

The components of RDF-TDAA's index utilize memory-mapped files to enable efficient data access and retrieval. In C++, memory mapping directly associates file contents with memory addresses, optimizing large-file I/O by allowing fast, random-access operations without intermediate buffering. Each component of the index, including the trie structure, characteristic set index, and predicate index, is stored in a separate file.

To reduce the storage overhead of the trie structure, we optimize the storage of DAA structures. When a DAA structure contains only a single array of length one, we store this single value directly within the pointer array, which typically holds the starting offsets of DAA structures. The highest bit of each value in the pointer array acts as a flag, indicating whether the value is an offset pointer or an actual array element.

In our leapfrog triejoin implementation, we employ an iterative rather than recursive approach. This iterative method enhances efficiency by avoiding function call overhead and stack management. Organized as a loop, the iterative leapfrog triejoin processes a query plan table where sorted variables are assigned to rows and triple patterns to columns, placing each variable in its appropriate cell. The algorithm iterates row by row through the table, storing candidate values in variable cells, performing intersections for each variable, and using results and constants to retrieve candidate values for subsequent variables.

8. Experiments

8.1. Experimental setup

Datasets. Table 1 provides a summary of the datasets used in our experiments, which include both synthetic and real-world data. We first utilize the Wikidata Graph Pattern Benchmark (WGPB) (Hogan et al., 2019), a benchmark that contains a dataset utilizes a subgraph of Wikidata with 82,923,234 triples, 54,251,877 nodes, and 2101 predicates. This benchmark offers 17 query patterns of varying widths and shapes, encompassing both cyclic and acyclic queries. Each pattern is instantiated with 50 queries constructed through random walks, ensuring nonempty results. The benchmark enables comparison across different alternatives for various abstract patterns.

Table 1
Datasets statistics in Millions (M). P is the number of unique predicates in a dataset.

Datasets	Triples(M)	Nodes(M)	P
WGPB	82.92	54.25	2101
WatDiv100M	109.89	10.25	86
DBpedia	681.33	182.59	62 558
SWDF	0.304	0.103	158
YAGO 2.3.0	109.89	54.35	96

The WatDiv100M dataset is a synthetic dataset generated by the Waterloo SPARQL Diversity Test Suite (WatDiv) benchmark (Aluç, Hartig, Özsu, & Daudjee, 2014). This dataset contains 109,890,201 triples, 10,251,255 nodes, and 86 predicates. This benchmark includes queries in four categories: Linear (L), Star (S), Snowflake-shaped (F), and Complex (C), totaling 20 queries.

The DBpedia dataset, a widely recognized real-world dataset, is derived from the DBpedia knowledge graph. The dataset and the queries used in our experiments are identical to those in Bigerl et al. (2020), which can be accessed online.[1] This dataset is the English version of DBpedia from 2015–10, containing 681,331,702 triples, 182,591,723 nodes, and 62,558 predicates. The corresponding 554 queries were generated using FEASIBLE (Saleem, Mehmood and Ngomo, 2015) based on real-world query logs contained in LSQ (Saleem, Ali, Hogan, Mehmood and Ngomo, 2015).

The Semantic Web Dog Food (SWDF) (Möller, Heath, Handschuh, & Domingue, 2007) is a real-world dataset containing Semantic Web conference metadata about people, papers, and talks. It contains 304,583 triples, 103,349 nodes, and 185 predicates. We use 14,740 queries generated by FEASIBLE.

YAGO (Hoffart, Suchanek, Berberich, & Weikum, 2013) is a real-world dataset with general knowledge about people, cities, countries, movies, and organizations. We use YAGO 2.3.0 in the experiment, which contains 109,891,150 triples, 54,351,091 nodes, and 96 predicates. Since YAGO 2.3.0 is a real dataset with no benchmark queries, we used the same set of queries (Y1–Y4) defined in Abdelaziz, Harbi, Khayyat, and Kalnis (2017), Al-Harbi, Abdelaziz, Kalnis, Mamoulis, Ebrahim, and Sahli (2016). Y1 and Y2 are selective queries that result in a small number of results, while Y3 and Y4 are data-intensive queries that require non-selective object-object joins.

Competitors. We compare our approach against a range of state-of-the-art and well-established engines or indexes in our experiments, covering both relational- and tensor-based engines. A summary of each engine is provided below:

- **RDF-3X** (Neumann & Weikum, 2010): RDF-3X indexes a single table of triples using a compressed clustered B+ tree. The triples are sorted to allow differential encoding within each B+ tree leaf. RDF-3X processes triple patterns by scanning triple ranges and employs a query optimizer based on pairwise joins.
- **Ring** (Arroyuelo et al., 2021): Ring is an in-memory RDF store that utilizes FM-indexes to represent and index RDF graphs in a structure called a "Ring". For basic graph pattern queries, Ring employs a LTJ variant to achieve worst-case optimal joins. Additionally, Ring offers a compressed variant, called C-Ring, which

[1] https://tentris.dice-research.org/iswc2020/.

Table 2
Combined storage size (GB) of index and string dictionary.

RDF engines	WGPB	WatDiv100M	DBpedia	SWDF	YAGO 2.3.0
RDF-3X	8.168	5.098	47.795	0.223	8.178
gStore	30.722	8.174	143.472	0.778	26.899
Tentris	20.772	11.359	108.344	0.717	26.526
Jena-LTJ	14.481	15.693	107.165	0.053	18.406
MillenniumDB	5.014	4.736	34.920	0.075	5.752
RDF-TDAA (ours)	**3.275**	**1.464**	**20.745**	**0.011**	**3.34**

features a more compact index but at the cost of significantly slower retrieval speeds. In our experiments, we use the latest version of Ring (Arroyuelo et al., 2024).

- **gStore** (Shen et al., 2015) is a in-memory graph-based RDF store that represents the RDF graph using adjacency lists. Each node has a bit vector as a vertex signature, encoding triples where the given node is the subject. These signatures are indexed in a vertex signature tree (VS-tree). Queries are processed using a subgraph matching algorithm.
- **Tentris** (Bigerl et al., 2020): Tentris is an RDF store representing RDF data as a one-hot encoded 3-order tensor within a hypertrie structure. Basic graph patterns are translated into tensor operations evaluated on the hypertrie via a worst-case optimal join algorithm. We use the latest version of Tentris (Bigerl, Conrads, Behning, Saleem and Ngonga Ngomo, 2022) in the experiments.
- **Jena-LTJ** (Hogan et al., 2019): Jena-LTJ implements the LTJ algorithm on top of Jena TDB.[2] It indexes triples in all six possible orders (SPO, SOP, OPS, OSP, POS, and PSO) using B+ trees.
- **MillenniumDB** (Vrgoc et al., 2024) is an open source graph database that stores RDF data using B+ trees and uniquely assigned 8-byte object IDs. It employs multiple indexing strategies, including four permutations of RDF triples, edge ID lookups, and property-based indexes, to optimize query performance. For query execution, MillenniumDB also supports leapfrog triejoin.
- **The Permuted Trie Index** (Perego et al., 2020): The Permuted Trie Index (referred to as PTI hereafter) is a trie-based index for RDF data, utilizing advanced compression techniques. It stores RDF data in three permutations, employing cross-compression to compress a permutation based on triples from another permutation. The PTI has several types of implementations, each with different trade-offs between retrieval speed and index size.

Hardware Configuration. The experiments were conducted on a machine running a Linux-based system, specifically Ubuntu Desktop 24.04 LTS with Linux kernel version 6.8.0. The machine is equipped with a 6-core Intel i5-12400F CPU operating at 2.5 GHz, alongside 32 GB of DDR4 RAM with a frequency of 3200 MHz, and a 20 GB swap file. Storage is provided by a 2 TB PCIe Gen4 SSD, offering a maximum sequential read speed of 7400 MB/s and a maximum sequential write speed of 6500 MB/s.

8.2. Storage performance

RDF-TDAA stores an RDF graph in two parts: the index and the string dictionary. The string dictionary allows each triple in the dataset to be represented by three integers and maps integers in query results back to their original URI strings. The use of a string dictionary is common in RDF engines, as URI strings in RDF triples can be lengthy and often appear across multiple RDF statements. Mapping triple components to integer IDs helps to reduce storage space.

We first compares the combined storage size (index and string dictionary) of RDF-TDAA with state-of-the-art RDF engines (RDF-3X, gStore, Tentris, Jena-LTJ and MillenniumDB) across all datasets, which

[2] https://jena.apache.org/documentation/tdb/.

Table 3
Index size (MB) comparison across datasets.

RDF engines	WGPB	WatDiv100M	DBpedia	SWDF	YAGO 2.3.0
Ring	1096.69	1315.84	7356.42	2.28	1108.98
PTI	924.92	818.18	6359.78	**2.11**	**983.94**
RDF-TDAA (ours)	**889.64**	**691.29**	**6179.15**	2.12	987.42

is shown in Table 2. RDF-TDAA consistently achieves the smallest total storage, demonstrating superior efficiency. Specifically, RDF-TDAA requires only 3.275 GB for WGPB, 1.464 GB for WatDiv100M, 20.745 GB for DBpedia, 0.011 GB for SWDF, and 3.34 GB for YAGO 2.3.0. In contrast, other engines exhibit significantly larger storage requirements. For example, RDF-3X, gStore, and Tentris require 8.168 GB, 30.722 GB, and 20.772 GB for WGPB, respectively. Notably, MillenniumDB demonstrates competitive storage efficiency but still falls short of RDF-TDAA in all datasets. These results highlight the superior efficiency of RDF-TDAA's index structure.

We then compare the index sizes of RDF-TDAA and various competitors, excluding the string dictionary size, as Ring and PTI lack a string dictionary. Our evaluation includes comparisons between the index of RDF-TDAA and state-of-the-art indices like Ring and PTI. Each of these indices has multiple implementations; some configurations offer slower retrieval with smaller index sizes, while others achieve faster retrieval at the cost of larger indices. For clarity in comparison, we report the average index size across these implementations. The results are presented in Table 3, which highlights the storage efficiency of RDF-TDAA relative to Ring and PTI across all datasets.

Table 3 shows that RDF-TDAA consistently outperforms Ring across all datasets. However, its performance compared to PTI varies. RDF-TDAA outperforms PTI in WatDiv100M (691.24 MB, 15.5% smaller than PTI), WGPB (893.95 MB, 3.3% smaller than PTI), and DBpedia (6178.82 MB, 2.8% smaller than PTI). In contrast, PTI has the advantage in SWDF (2.06 MB, 4.6% bigger than PTI) and YAGO 2.3.0 (893.95 MB, 6.1% smaller). This difference likely arises from PTI's use of advanced integer sequence compression algorithms, which prioritize aggressive storage optimization at the expense of increased decompression time. These findings suggest that integrating batter compression techniques into RDF-TDAA's trie-based structure could enhance its storage efficiency.

To provide a better comparison with PTI, we calculated the average bytes per triple across all datasets. RDF-TDAA achieves 8.82 bytes per triple, whereas PTI averages 9.19 bytes per triple. Indicating that RDF-TDAA is generally more space-efficient, particularly for large-scale datasets.

Overall, RDF-TDAA achieves lower storage requirements, providing efficient indexing across various RDF data-sets.

8.3. Query performance

We evaluate query performance using both the command-line and SPARQL endpoint interfaces, as different engines support different methods for handling SPARQL queries. Engines such as RDF-3X and Ring support only a command-line interface, while Tentris, Jena-LTJ, and MillenniumDB provide a SPARQL endpoint interface. Accordingly, we use the command-line interface for RDF-TDAA to compare with RDF-3X and Ring, and the SPARQL endpoint interface to compare with Tentris, Jena-LTJ, and MillenniumDB. Additionally, to ensure a fair comparison, we clear the page cache and buffer cache before each batch of tests, as Ring is the only in-memory engine. For Ring, we utilize its original implementation, which, despite its relatively larger index size compared to C-Ring, delivers faster query performance than C-Ring. As for PTI, since it lacks support for executing queries, it is excluded from the query performance evaluation.

WGPB. Tables 4 and 5 present the query performance on the WGPB dataset using the command-line and SPARQL endpoint interfaces, respectively. Across both interfaces, RDF-TDAA demonstrates consistently

Table 4
Average query time (ms) for WGPB using command-line interface.

RDF engines	J3	J4	P2	P3	P4	S1	S2	S3	S4	T2	T3	T4	TI2	TI3	TI4	Tr1	Tr2
RDF-3X	21.6	13.2	15.3	33.2	11.5	66.1	**8.5**	13.4	7.5	7.1	4.7	4.4	10.8	4.6	4.1	121.5	37.8
Ring	14.2	12.4	17.8	15.4	98.4	42.8	49.4	119.9	19.8	6.8	8.2	7.4	11.2	11.7	11.3	60.2	27.9
RDF-TDAA (ours)	**2.7**	**2.6**	**3.3**	**3.2**	**3.9**	**3.9**	29.9	**7.5**	**7.9**	**1.4**	**1.3**	**2.0**	**1.1**	**0.9**	**1.1**	**3.7**	**2.2**

Table 5
Average query time (ms) for WGPB using SPARQL endpoint interface.

RDF engines	J3	J4	P2	P3	P4	S1	S2	S3	S4	T2	T3	T4	TI2	TI3	TI4	Tr1	Tr2
gStore	29.2	22.2	20.0	19.5	34.3	36.4	49.6	42.3	36.2	14.6	8.6	7.5	24.7	15.9	20.0	33.3	27.2
Tentris	198.7	73.6	142.3	56.4	23.9	19.7	19.6	26.7	11.4	29.7	8.0	21.1	8.5	5.1	14.2	11.2	4.5
Jena-LTJ	23.9	22.3	26.1	19.3	19.1	14.8	**13.2**	26.3	10.9	16.6	12.9	11.4	31.3	18.7	11.9	15.6	10.9
MillenniumDB	11.4	12.8	**4.3**	8.8	11.3	12.5	19.2	13.6	11.4	6.6	8.2	10.0	3.4	3.8	3.9	7.5	7.6
RDF-TDAA (ours)	**5.6**	**6.6**	5.4	**6.2**	**6.1**	**5.7**	30.6	**8.3**	**9.0**	**2.5**	**1.8**	**2.2**	**1.5**	**1.4**	**1.8**	**4.8**	**3.9**

Table 6
Average query time (ms) for WatDiv100M using command-line interface.

RDF engines	L	S	F	C
RDF-3X	37.63	9.27	19.45	2005.45
Ring	21.49	19.38	10.57	2570.70
RDF-TDAA (ours)	**21.08**	**4.41**	**4.92**	**730.66**

Table 7
Average query time (ms) for WatDiv100M using SPARQL endpoint interface.

RDF engines	L	S	F	C
gStore	46.77	11.49	58.14	1926.07
Tentris	80.94	28.47	27.46	1009.24
Jena-LTJ	40.98	21.07	15.63	3668.54
MillenniumDB	**11.23**	13.56	12.06	870.41
RDF-TDAA (ours)	24.25	**10.33**	**9.78**	**434.7**

Table 8
Average query time (ms) and time out count for DBpedia using the command-line interface.

RDF engines	Query time	Time outs
RDF-3X	72.441	**0**
Ring	96.878	19
RDF-TDAA (ours)	**46.304**	**0**

Table 9
Average query time (ms) and time out count for DBpedia using the SPARQL endpoint interface.

RDF engines	Query time	Time outs
Tentris	136.238	21
Jena-LTJ	145.046	9
MillenniumDB	73.296	2
RDF-TDAA (ours)	**46.603**	**0**

Table 10
Average query time (ms) for SWDF using command-line interface.

RDF engines	Query time
RDF-3X	9.164
Ring	11.714
RDF-TDAA (ours)	**3.381**

Table 11
Average query time (ms) for SWDF using SPARQL endpoint interface.

RDF engines	Query time
gStore	2.729
Tentris	0.876
Jena-LTJ	1.504
MillenniumDB	0.924
RDF-TDAA (ours)	**0.839**

strong query performance, outperforming other engines in most query types. However, RDF-TDAA exhibits slightly lower efficiency in the S2 queries, where its performance lags marginally behind other engines. This is due to the query plan generated for S2 queries, which may not be the most optimal. Additionally, RDF-TDAA is slower than MillenniumDB in P2 queries, consistent with findings from the WatDiv benchmark, which also showed that RDF-TDAA is less efficient than MillenniumDB in some linear or path queries. Despite these specific cases, RDF-TDAA remains competitive in terms of overall query performance across the WGPB dataset.

WatDiv100M. We evaluated all 20 queries from the WatDiv100M benchmark, averaging query times across the four query types (L, S, F, and C). As shown in Tables 6 and 7, RDF-TDAA consistently outperforms other engines across most query types. In Linear (L) queries using the SPARQL endpoint interface, RDF-TDAA is slower than MillenniumDB. However, RDF-TDAA achieves the lowest average query times with both interfaces for the other query types, particularly excelling in complex (C) queries.

DBpedia. Tables 8 and 9 present the query performance on the DBpedia dataset using the command-line interface and the SPARQL endpoint interface, respectively. A 5-s timeout threshold was applied across 554 queries, and the average query time was calculated excluding failed queries. Notably, gStore requires loading the index cache into memory, but its index size on DBpedia exceeds our system's memory capacity. As a result, its query time is not reported in Table 9.

Using the command-line interface, RDF-TDAA consistently demonstrates superior performance with no timeouts. RDF-3X also completes all queries within the 5-s threshold. In contrast, Ring encounters 19 timeouts. With the SPARQL endpoint interface, for the SPARQL endpoint interface, RDF-TDAA maintains its efficiency with an average query time of 46.603 ms and no timeouts, demonstrating robust real-world applicability. Meanwhile, Tentris (136.238 ms, 21 timeouts) and Jena-LTJ (145.046 ms, 9 timeouts) experience significant slowdowns and instability under the same constraints, with MillenniumDB (73.296 ms, 2 timeouts) showing moderate improvement but still lagging behind RDF-TDAA. These results underscore RDF-TDAA's reliability.

Overall, RDF-TDAA demonstrates consistent zero timeouts and lower average query times, highlighting its robustness for large-scale RDF data processing.

SWDF. Tables 10 and 11 summarize the query performance for 14,740 queries on the SWDF dataset across interfaces. In the command-line interface, RDF-TDAA achieves the fastest execution, with an average query time of 3.381 ms, outperforming RDF-3X (9.164 ms) and Ring (11.714 ms). This demonstrates RDF-TDAA's efficiency in handling large-scale query workloads.

In the SPARQL endpoint interface, RDF-TDAA continues to excel, achieving the lowest average query time of 0.839 ms, surpassing MillenniumDB (0.924 ms) and significantly outperforming Jena-LTJ (1.504 ms). Tentris, despite being competitive, exhibits higher latency.

Table 12
Average query time (ms) for YAGO 2.3.0 using command-line interface.

RDF engines	Y1	Y2	Y3	Y4
RDF-3X	**20.72**	43.76	731.34	21.81
Ring	39.54	17.07	11 658.91	180.93
RDF-TDAA (ours)	46.34	**2.03**	**725.89**	**10.9**

Table 13
Average query time (ms) for YAGO 2.3.0 using SPARQL endpoint interface.

RDF engines	Y1	Y2	Y3	Y4
gStore	**39.6**	28.98	1767.06	54.64
Tentris	115.82	27.81	2082.32	113.79
Jena-LTJ	47.47	11.12	4090.75	38.04
MillenniumDB	41.01	22.71	1179.14	95.93
RDF-TDAA (ours)	49.41	**5.03**	**590.35**	**12.35**

These results highlight RDF-TDAA's ability to efficiently process a high volume of queries, maintaining low query latency across different execution environments.

YAGO 2.3.0. As shown in Tables 12 and 13, RDF-TDAA achieves the fastest execution times for Y2, Y3, and Y4 across both interfaces, significantly outperforming competitors. However, RDF-TDAA lags slightly in Y1 query, where RDF-3X (20.72 ms) and gStore (39.6 ms) exhibit faster performance.

9. Conclusions and future work

In this study, we presented RDF-TDAA, an efficient solution for RDF indexing and querying, designed to address the challenges posed by large-scale RDF datasets. RDF-TDAA combines DAAs and CSs within a trie-based structure to enhance storage efficiency. The integration of a novel path-based query planning mechanism and a worst-case optimal join algorithm accelerates query processing, overcoming the limitations of existing RDF indexing engines.

The experimental results validate RDF-TDAA's scalability and effectiveness across a range of synthetic and real-world datasets. In comparison to state-of-the-art RDF indexing solutions, RDF-TDAA consistently delivers superior performance in terms of both reduced indexing size and faster query execution times. This makes RDF-TDAA a highly competitive choice for quering large RDF knowledge graphs using SPARQL queries.

RDF-TDAA presents a promising approach to scalable RDF data management. However, several limitations of the current implementation warrant further exploration. The predicate index is inefficient when a predicate is associated with a large number of subjects or objects, leading to suboptimal retrieval performance. Additionally, learned models can be utilized to predict the position of a given value within an array (Kraska, Beutel, Chi, Dean, & Polyzotis, 2018; Lan, Bao, Culpepper, Borovica-Gajic, & Dong, 2024), reducing the reliance on binary search operations and significantly enhancing lookup efficiency. Further improvements can also be achieved through advanced compression techniques, which would enhance space efficiency.

As a centralized RDF engine, RDF-TDAA faces inherent scalability constraints. Future work could investigate adapting its index structure for distributed environments. For instance, splitting the unified DAA structure (the largest storage component) across machines, with pointers augmented to include machine addresses for cross-node navigation. Alternatively, graph partitioning algorithms could distribute the RDF graph across machines, with RDF-TDAA's indexing applied locally to each partition.

Another key area for improvement lies in query execution strategies. While the leapfrog triejoin algorithm is efficient, experimental results reveal that it is not universally optimal. Worst-case optimal join algorithms excel at cyclic queries but are less effective for acyclic queries, where traditional join algorithms perform better (Ngo, Ré, & Rudra, 2013). Future research could explore more advanced worst-case optimal join algorithms or hybrid approaches that dynamically select between worst-case optimal and classic join algorithms based on query structure. Additionally, the path-based query planning mechanism, though effective, could benefit from dynamic optimization strategies to adapt to varying workloads.

In future work, we will optimize indexing by enhancing predicate indexing efficiency and leveraging learned indexes for faster lookups. Additionally, we will refine query execution strategies by developing adaptive join selection mechanisms and exploring graph homomorphism algorithms to improve performance.

Table 14
Number of PSs and CSs for subjects, and PSs and RCSs for objects (values in millions, M).

Dataset	Subject		Object	
	PSs (M)	CSs (M)	PSs (M)	RCSs (M)
Watdiv	5.21	0.04	9.75	1.1×10^{-3}
WGPB	19.23	1.73	39.89	0.68
DBpedia	40.43	1.84	178.28	0.98
SWDF	3.69×10^{-2}	2.49×10^{-3}	9.55×10^{-2}	1.89×10^{-3}
YAGO 2.3.0	10.12	6.95×10^{-2}	46.03	1.68×10^{-3}

CRediT authorship contribution statement

Yipeng Liu: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Writing – original draft. **Yuming Lin:** Funding acquisition, Resources, Project administration, Validation, Writing – review & editing. **Xinyong Peng:** Funding acquisition, Project administration, Supervision. **You Li:** Methodology, Writing – review & editing, Funding acquisition. **Jingwei Zhang:** Writing – review & editing, Resources, Supervision.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to improve language and readability. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Natural Science Foundation of China (Nos. 62362015 and U22A2099), Guangxi Key Research and Development Program (No. Gui Ke AB24010160), Open Project Program of Guangxi Key Laboratory of Digital Infrastructure (No. GXDIOP2024013), and Innovation Project of GUET Graduate Education (No. 2025YCXS056).

Appendix A. Effectiveness of using characteristic sets

To evaluate the effectiveness of using Characteristic Sets (CS) in the trie structure, we analyzed the number of Predicate Sets (PSs) and CSs for each subject, as well as the number of PSs and Reverse Characteristic Sets (RCS) for each object in the datasets. We analyzed both synthetic and real-world datasets, as detailed in Section 8.1, with the results summarized in Table 14.

The results in Table 14 demonstrate that the number of CSs and RCSs is significantly lower than the total number of PSs in the datasets. This reduction highlights the efficiency of using CSs and RCSs in the trie structure, as they effectively reduce duplicates of PSs, thereby optimizing storage space.

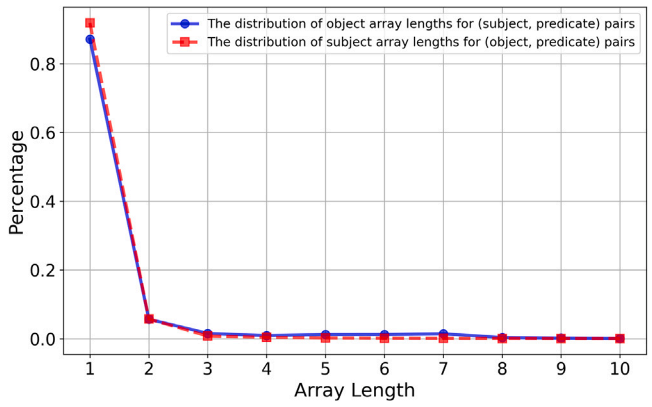


Fig. 8. The distribution of object array lengths for (subject, predicate) pairs and the distribution of subject array lengths for (object, predicate) pairs in DBpedia dataset.

Appendix B. Analysis of retrieval overhead in DAAs

In typical scenarios, a trie based on DAAs achieves reduced storage size; however, the retrieval process can become more complex compared to the original trie structure. Specifically, when accessing a subject or object array in a trie with DAAs, elements must be retrieved sequentially. To demonstrate that the retrieval overhead is minimal and that DAAs are well-suited for indexing RDF data, we analyzed the distribution of object arrays lengths for (subject, predicate) pairs and subject arrays lengths for (object, predicate) pairs across all datasets.

As shown in Fig. 8, the charts illustrate the percentage distribution of array lengths in the DBpedia dataset. Our analysis reveals that most arrays have a length of one, and the distributions in other datasets closely resemble those in the DBpedia dataset. For clarity, we present the statistics for the DBpedia dataset alone. Furthermore, DAAs provide the additional advantage of directly accessing the first element of an array without any computational overhead. Consequently, the retrieval overhead of using DAAs in a trie-based index for RDF data with SPO and OPS permutations is negligible in most cases.

Data availability

Data will be made available on request.

References

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

Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., & Sahli, M. (2016). Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, *25*(3), 355–380.

Aluç, G., Hartig, O., Özsu, M. T., & Daudjee, K. (2014). Diversified stress testing of RDF data management systems. In *The semantic web - ISWC 2014: 13th international semantic web conference, riva del garda, Italy, October 19-23, 2014. proceedings, part i* (pp. 197–212). Berlin, Heidelberg: Springer-Verlag.

Álvarez-García, S., Brisaboa, N. R., Fernández, J. D., Martínez-Prieto, M. A., & Navarro, G. (2015). Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, *44*(2), 439–474.

Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Reutter, J., Rojas-Ledesma, J., et al. (2024). The ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, *49*(2).

Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J. L., Rojas-Ledesma, J., & Soto, A. (2021). Worst-case optimal graph joins in almost no space. In *Proceedings of the 2021 international conference on management of data* (pp. 102–114). New York, NY, USA: Association for Computing Machinery.

Atserias, A., Grohe, M., & Marx, D. (2013). Size bounds and query plans for relational joins. *SIAM Journal on Computing*, *42*(4), 1737–1767.

Bigerl, A., Conrads, L., Behning, C., Saleem, M., & Ngomo, A. N. (2022). Hashing the hypertrie: Space- and time-efficient indexing for SPARQL in tensors. In *Lecture notes in computer science*: *vol. 13489*, *The semantic web - ISWC 2022 - 21st international semantic web conference, virtual event, October 23-27, 2022, proceedings* (pp. 57–73). Springer.

Bigerl, A., Conrads, L., Behning, C., Saleem, M., & Ngonga Ngomo, A.-C. (2022). Hashing the hypertrie: Space- and time-efficient indexing for SPARQL in tensors. In *The semantic web – ISWC 2022* (pp. 57–73). Cham: Springer International Publishing.

Bigerl, A., Conrads, F., Behning, C., Sherif, M. A., Saleem, M., & Ngonga Ngomo, A.-C. (2020). Tentris – a tensor-based triple store. In *The semantic web – ISWC 2020* (pp. 56–73). Cham: Springer International Publishing.

Brisaboa, N. R., Ladra, S., & Navarro, G. (2013). DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, *49*(1), 392–404.

Chen, Y., Özsu, M. T., Xiao, G., Tang, Z., & Li, K. (2021). Gsmart: An efficient SPARQL query engine using sparse matrix algebra - full version. CoRR abs/2106.14038. URL: https://arxiv.org/abs/2106.14038. arXiv:2106.14038.

Dimitrov, D., Baran, E., Fafalios, P., Yu, R., Zhu, X., Zloch, M., et al. (2020). Tweetscov19 - a knowledge base of semantically annotated tweets about the COVID-19 pandemic. In *CIKM '20: the 29th ACM international conference on information and knowledge management, virtual event, Ireland, October 19-23, 2020* (pp. 2991–2998). ACM.

Erling, O., & Mikhailov, I. (2009). Virtuoso: RDF support in a native RDBMS. In *Semantic web information management - a model-based perspective* (pp. 501–519). Springer.

Fernández, J. D., Martínez-Prieto, M. A., & Gutierrez, C. (2010). Compact representation of large RDF data sets for publishing and exchange. In *Lecture notes in computer science*: *vol. 6496*, *The semantic web - ISWC 2010 - 9th international semantic web conference, ISWC 2010, Shanghai, China, November 7-11, 2010, revised selected papers, part i* (pp. 193–208). Springer.

Guo, X., Le-Tuan, A., & Phuoc, D. L. (2023). Building a P2P RDF store for edge devices. In *Proceedings of the 13th international conference on the internet of things, ioT 2023, Nagoya, Japan, November 7-10, 2023* (pp. 33–41). ACM.

Hoffart, J., Suchanek, F. M., Berberich, K., & Weikum, G. (2013). YAGO2: a spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, *194*, 28–61.

Hogan, A., Riveros, C., Rojas, C., & Soto, A. (2019). A worst-case optimal join algorithm for SPARQL. In *The semantic web - ISWC 2019: 18th international semantic web conference, Auckland, New Zealand, October 26-30, 2019, proceedings, part i* (pp. 258–275). Berlin, Heidelberg: Springer-Verlag.

Ingalalli, V., Ienco, D., Poncelet, P., & Villata, S. (2016). Querying RDF data using a multigraph-based approach. In *Proceedings of the 19th international conference on extending database technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016* (pp. 245–256). OpenProceedings.org.

Jamour, F. T., Abdelaziz, I., Chen, Y., & Kalnis, P. (2019). Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs. In *Proceedings of the fourteenth euroSys conference 2019, Dresden, Germany, March 25-28, 2019* (pp. 27:1–27:15). ACM.

Jaradeh, M. Y., Oelen, A., Farfar, K. E., Prinz, M., D'Souza, J., Kismihók, G., et al. (2019). Open research knowledge graph: Next generation infrastructure for semantic scholarly knowledge. In *Proceedings of the 10th international conference on knowledge capture, k-CAP 2019, Marina del Rey, CA, USA, November 19-21, 2019* (pp. 243–246). ACM.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. In G. Das, C. M. Jermaine, & P. A. Bernstein (Eds.), *Proceedings of the 2018 international conference on management of data, SIGMOD conference 2018, Houston, TX, USA, June 10-15, 2018* (pp. 489–504). ACM.

Lan, H., Bao, Z., Culpepper, J. S., Borovica-Gajic, R., & Dong, Y. (2024). A fully on-disk updatable learned index. In *40th IEEE international conference on data engineering, ICDE 2024, Utrecht, the Netherlands, May 13-16, 2024* (pp. 4856–4869). IEEE.

Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., et al. (2015). DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, *6*(2), 167–195.

Möller, K., Heath, T., Handschuh, S., & Domingue, J. (2007). Recipes for semantic web dog food - the ESWC and ISWC metadata projects. In *Lecture notes in computer science*: *vol. 4825*, *The semantic web, 6th international semantic web conference, 2nd Asian semantic web conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007* (pp. 802–815). Springer.

Mountantonakis, M., & Tzitzikas, Y. (2023). Using multiple RDF knowledge graphs for enriching ChatGPT responses. In *Lecture notes in computer science*: *vol. 14175*, *Machine learning and knowledge discovery in databases: applied data science and demo track - European conference, ECML PKDD 2023, Turin, Italy, September 18-22, 2023, proceedings, part VII* (pp. 324–329). Springer.

Navarro, G., Reutter, J. L., & Rojas-Ledesma, J. (2020). Optimal joins using compact data structures. In *LIPIcs*: *vol. 155*, *23rd international conference on database theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark* (pp. 21:1–21:21). Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Neumann, T., & Moerkotte, G. (2011). Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th international conference on data engineering* (pp. 984–994).

Neumann, T., & Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, *19*, 91–113.

Ngo, H. Q., Ré, C., & Rudra, A. (2013). Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, *42*(4), 5–16.

Perego, R., Pibiri, G. E., & Venturini, R. (2020). Compressed indexes for fast search of semantic data. *IEEE Transactions on Knowledge and Data Engineering*, *33*(9), 3187–3198.

Saleem, M., Ali, M. I., Hogan, A., Mehmood, Q., & Ngomo, A. N. (2015). LSQ: the linked SPARQL queries dataset. In *Lecture notes in computer science*: *vol. 9367*, *The semantic web - ISWC 2015 - 14th international semantic web conference, Bethlehem, PA, USA, October 11-15, 2015, proceedings, part II* (pp. 261–269). Springer.

Saleem, M., Mehmood, Q., & Ngomo, A. N. (2015). FEASIBLE: a feature-based SPARQL benchmark generation framework. In *Lecture notes in computer science*: *vol. 9366*, *The semantic web - ISWC 2015 - 14th international semantic web conference, Bethlehem, PA, USA, October 11-15, 2015, proceedings, part i* (pp. 52–69). Springer.

Shen, X., Zou, L., Üzsu, M. T., Chen, L., Li, Y., Han, S., et al. (2015). A graph-based RDF triple store. In *2015 IEEE 31st international conference on data engineering* (pp. 1508–1511). http://dx.doi.org/10.1109/ICDE.2015.7113413.

Shi, J., Yao, Y., Chen, R., Chen, H., & Li, F. (2016). Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *12th USeNIX symposium on operating systems design and implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (pp. 317–332). USENIX Association.

Thompson, B. B., Personick, M., & Cutcher, M. (2014). The bigdata® RDF graph database. In *Linked data management* (pp. 193–237). Chapman and Hall/CRC.

Troullinou, G., Agathangelos, G., Kondylakis, H., Stefanidis, K., & Plexousakis, D. (2024). DIAERESIS: RDF data partitioning and query processing on SPARK. *Semantic Web*, *15*(5), 1763–1789.

Veldhuizen, T. L. (2014). Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th international conference on database theory (ICDT), Athens, Greece, March 24-28, 2014* (pp. 96–106). OpenProceedings.org.

Vrandecic, D., & Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, *57*(10), 78–85.

Vrgoc, D., Rojas, C., Angles, R., Arenas, M., Calisto, V., Farias, B., et al. (2024). MillenniumDB: A multi-modal, multi-model graph database. In *Companion of the 2024 international conference on management of data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024* (pp. 496–499). ACM.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, *1*(1), 1008–1019.

Williams, H. E., & Zobel, J. (1999). Compressing integers for fast file access. *Computer Journal*, *42*, 193–201.

Yao, Z., Chen, R., Zang, B., & Chen, H. (2022). Wukong+G: Fast and concurrent RDF query processing using RDMA-assisted GPU graph exploration. *IEEE Transactions on Parallel and Distributed Systems*, *33*(7), 1619–1635.