

# Efficient Updates for Worst-Case Optimal Join Triple Stores

Alexander Bigerl<sup>1</sup><sup>[0000-0002-9617-1466]</sup> (✉), Nikolaos Karalis<sup>1</sup><sup>[0000-0002-0710-7180]</sup>, Liss Heidrich<sup>1</sup><sup>[0009-0006-2031-2548]</sup>, and Axel-Cyrille Ngonga Ngomo<sup>1</sup><sup>[0000-0001-7112-3516]</sup> (✉)

Data Science Group (DICE), Heinz Nixdorf Institute, Paderborn University  
{alexander.bigerl, nikolaos.karalis, liss.heidrich, axel.ngonga}@uni-paderborn.de

**Abstract.** It has been recently shown that worst-case optimal joins can significantly speed up query processing in RDF triple stores, especially in analytical workloads. However, this increase in query speed comes at the expense of updates being slow or not supported at all. We see this limited compatibility with updates as a key reason for the slow adoption of worst-case optimal joins in triple stores. In this paper, we address this challenge by presenting a fast, incremental insertion and deletion algorithm for the hypertrie, a worst-case optimal join data structure. This update algorithm can be used for offline bulk updates as well as online updates. Our evaluation on realistic update loads from DBpedia and scaling update sizes on Wikidata shows that the online performance of our algorithm is comparable to or better than that of traditional triple stores.

**Keywords:** SPARQL update · worst-case optimal join · graph database · data structure · indexing

## 1 Introduction

Approaches based on worst-case optimal joins (WCOJs) for SPARQL query processing [2, 5, 6, 12, 17] have recently been shown to significantly outperform traditional methods using binary joins. For example, triangle queries—which are common in SPARQL—can be answered asymptotically faster with WCOJs compared to binary joins [3]. WCOJs rely on extensive indexing to efficiently evaluate SPARQL queries [2, 5, 6, 12]. However, a key limitation is the inefficiency of updates in these underlying indices [2].

In this paper, we address the challenge of supporting efficient updates in WCOJ-enabled RDF triple stores by proposing a fast incremental update algorithm for the hypertrie, a state-of-the-art index for WCOJs. The algorithm supports offline bulk updates at rates of up to 150k triples per second and online updates at up to 33k triples per second.<sup>1</sup> To evaluate the efficiency of the

---

<sup>1</sup> We refer to online updates when a HTTP SPARQL endpoint is live, and to offline updates otherwise.

proposed update mechanisms, we conduct a thorough comparison of our approach’s update performance with other triple stores. To achieve this goal, we use two benchmarks on real-world datasets: (i) DBpedia ( $680 \times 10^6$  triples) with updates derived from actual change logs and (ii) Wikidata ( $5.5 \times 10^9$  triples) with synthetic updates sampled from the dataset. Our results demonstrate that the proposed algorithm achieves update performance comparable to or better than that of conventional systems based on binary joins both in terms of update runtime and triples processed per second, thus solving the update bottleneck problem associated with WCOJs.

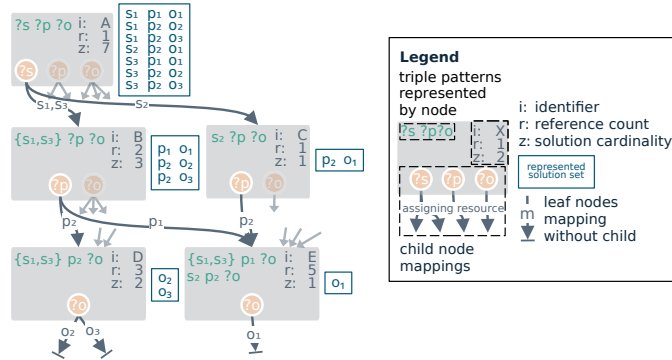
The rest of the paper is structured as follows. The related work in update support for WCOJ-enabled triple stores is reviewed in Section 2. Section 3 introduced RDF related notation. In Section 4, we provide an overview of the hypertrie data structure. The proposed update algorithm for efficient insertion and deletion operations on the hypertrie is detailed in Section 5 and its evaluation is discussed in Section 6. Finally, we conclude and discuss future directions in Section 7.

## 2 Related Work

WCOJ algorithms [16] are a class of multi-way join algorithms that eliminate a variable at a time instead of a triple pattern at a time like binary joins when applied in SPARQL [6]. WCOJs eliminate the influence of intermediate results from binary joins on the runtime complexity and are bound only by worst-case result size. For example, using binary joins to evaluate triangle query<sup>2</sup> where the triple patterns have  $n$  solutions is bound by  $\mathcal{O}(n^2)$  whereas a WCOJs reduce the complexity to  $\mathcal{O}(n^{1.5})$ . For more details about WCOJs in triplestores we refer the reader to [1, 2, 11, 14, 15].

The Ring [2] is an indexing data structure designed to store graphs with minimal memory overhead. It is a monolithic datastructure that supports WCOJs without relying on additional indices. However, the Ring does not support incremental updates, as any modification to the data requires a complete rebuild of the structure. To avoid rebuilding the Ring on every update the authors suggest to collect changes in a classical index and rebuild periodically. The hypertrie [5, 6] is another indexing data structure with native support for WCOJs, described in more detail in Section 4. While the versions presented in [5, 6] only provide basic loading capabilities similar to the Ring, this paper extends the hypertrie with full support for incremental updates. In contrast to the monolithic solutions Ring and hypertrie, triple stores like MillenniumDB [17] and Jena-LFJ [12] use multi-indexing approaches to enable WCOJs. MillenniumDB pre-materializes only four collation orders, falling back to binary joins when a WCOJ would require a different index. Jena-LFJ, on the other hand, materializes all possible collation orders to maximize query performance. Both systems leverage B-trees for indexing. Although these approaches with multiple indices enable updates,

<sup>2</sup> E.g., `SELECT * WHERE {?a :knows ?b . ?b :knows ?c . ?c :knows ?a}`.



**Fig. 1.** Example hypertrie. For better readability, all figures show only nodes for collation order SPO. The represented graph is equivalent to solution of the triple pattern in the root node.

they require a considerable amount of storage for the additional indices and the updates take longer as all indices have to be updated.

### 3 RDF Notation

We assume a basic familiarity with RDF and SPARQL. This section introduces the core terms and notation used throughout the paper [8, 10]. Let  $I$  denote the set of IRIs,  $L$  the set of literals,  $B$  the set of blank nodes, and  $V$  the set of variables. The set of RDF resources is  $R = I \cup L \cup B$ . An RDF triple is a tuple  $(s, p, o) \in (I \cup B) \times I \times R$ , where  $s$  is the subject,  $p$  the predicate, and  $o$  the object. A triple pattern is a tuple  $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (R \cup V)$ . A triple pattern's solution, or simply solution, is a partial function  $\mu : V \rightarrow R$  that assigns resources to variables. We write solutions as sets of pairs, e.g.,  $\{(?s, \text{ex:Alice}), (?p, \text{ex:knows})\}$ . A single assignment pair in a solution is referred to as a binding, e.g.,  $(?s, \text{ex:Alice})$ . A triple  $(s, p, o)$  matches a triple pattern  $(s', p', o')$  if each position in the pattern is either a variable or equal to the corresponding component of the triple. The resulting solution  $\mu$  binds each variable in the pattern to the matching RDF term from the triple. Given a triple pattern  $tp$  and an RDF graph  $G$ , matching  $tp$  to  $G$  yields the set of solutions that is derived from triples in  $G$  that match  $tp$ . For simplicity, we consistently use  $?s$ ,  $?p$ , and  $?o$  as variable names for subject, predicate, and object, respectively. This does not limit generality, as the paper does not consider joins that rely on shared variable names.

### 4 Hypertrie

The *hypertrie* [5, 6] is a data structure designed to support WCOJs over RDF knowledge graphs. To this end, the hypertrie organizes triple patterns and their

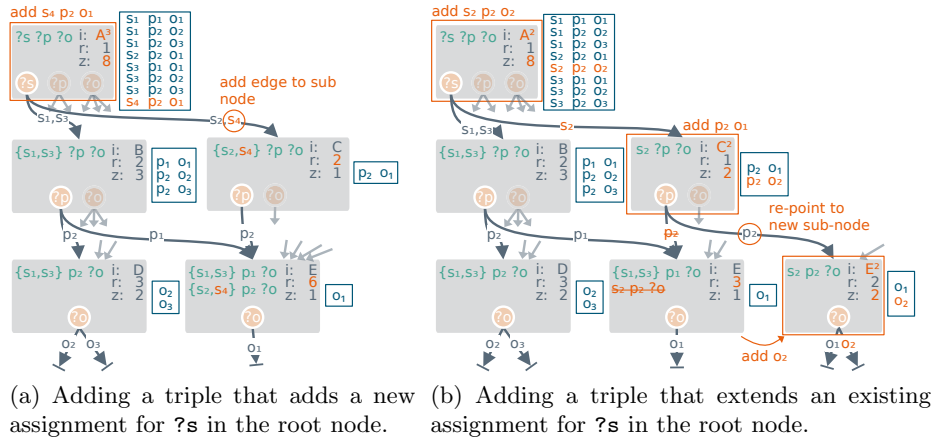
solutions hierarchically in a directed acyclic graph.<sup>3</sup> Each hypertrie node represents one or more triple patterns that share the same solution set. The cardinality of a hypertrie node refers to the cardinality of the solution set it represents. An example is given in Figure 1.

A hypertrie consists of three levels of nodes, organized by the number of unbound components (i.e., the number of variables) in the corresponding triple pattern. The root (depth-3) node represents the fully unbound pattern  $?s ?p ?o$ . Its children (depth-2 nodes) correspond to patterns with two unbound components, such as  $s_1 ?p ?o$ ,  $?s p_1 ?o$ , or  $?s ?p o_1$ . The third level (depth-1 nodes) contains patterns with one unbound component, e.g.,  $s_1 p_2 ?o$ . Each hypertrie node maintains a mapping for each unbound variable occurring in its associated triple pattern. For a given unbound variable  $v$  in a triple pattern  $tp$ , this mapping assigns each valid binding  $b$  of  $v$  a child node whose pattern is obtained by instantiating  $v$  in  $tp$  with  $b$ , while keeping the other positions unchanged. For example in Figure 1, in the root node’s (associated with  $?s ?p ?o$ ) variable mapping  $?s$  we assign  $s_1$  to a depth-2 node that represents the triple pattern  $s_1 ?p ?o$ . In depth-1 nodes, where only one variable remains unbound, the mapping reduces to the set of valid bindings for that variable. Following SPARQL semantics, the solution set of a child node consists of those solutions of the parent node in which the mapping variable  $v$  is bound to  $b$ , with  $v$  removed from the solution. Thus, each child node represents a projection of a filtered subset of its parent’s solutions.

To reduce redundancies and storage overhead in the hypertrie, [5] introduced three optimizations. (i) *Node deduplication via homomorphic hashing*: Nodes representing triple patterns with identical solution sets are deduplicated reducing the total number of nodes in the structure. For example in Figure 1 that uses this optimization, for variable mapping  $?s$  both  $s_1$  and  $s_3$  are assigned to the same node because the triple patterns  $s_1 ?p ?o$  and  $s_3 ?p ?o$  have the same solution set. For the deduplication each node is assigned an *identifier* ( $i$  in Figure 1) that is a hash of its solution set. Homomorphic hashing enables incremental updates to the identifier as the solution set changes, avoiding the need to sort and rehash the entire set on each update of a node. It is also used to look up if the node resulting from a update exists already. Since a single node may now be reachable from multiple parents, reference counting ( $r$  in Figure 1) is used to manage node sharing and deletion. The reference count of the root node defaults to 1, as it spans the entire hypertrie. (ii) *Singleton depth-2 nodes*: If a depth-2 node represents only a single solution, it is stored as a lightweight single entry node without any children. (iii) *Singleton depth-1 nodes*: If a depth-1 node would contain only a single solution, that solution is stored in-place in the parent node’s variable mapping.<sup>4</sup> Unless stated otherwise, figures in this paper apply only the

<sup>3</sup> Note that this applies only to triple patterns that do not contain a particular variable multiple times. Such triple patterns are handled in query processing. We are not aware of any triplestore that maintains an index for such patterns.

<sup>4</sup> This relies on the fact that the memory slot used for the reference to a child node is large enough to store the payload of a singleton depth-1 node.



**Fig. 2.** Inserting single triples into the hypertrie of Figure 1. All insertions and deletions in subsequent figures operate on the same hypertrie instance. Executed changes are highlighted in all figures in orange.

first optimization (node deduplication) to simplify the representation but our update algorithm takes all optimization into account.

## 5 Hypertrie Updates

This section presents our approach for performing insertions and deletions on the hypertrie. We begin by introducing the update mechanism for individual triples, followed by the creation and deletion of nodes, which together form the conceptual basis for bulk updates. We then describe how bulk updates are scheduled and executed efficiently. This includes reference count management and the deletion of unreferenced nodes. Finally, we discuss how singleton node optimizations are integrated into the update process. Each update operation consists exclusively of either insertions or removals.<sup>5</sup> Updates that include both must be split into two separate operations and executed consecutively.<sup>6</sup> All examples focus on the collation order SPO; the remaining collation orders are handled analogously. The required changes for all collation orders are collected in the same structures. This eliminates overhead for equivalent operations required multiple times for the same or different collation orders and ensures that each unique operation is executed only once.

### 5.1 Updates Using a Single Triple

<sup>5</sup> We assume only new triples are inserted and only existing ones are removed.

<sup>6</sup> This is in accordance with the SPARQL 1.1 Update specification [10].

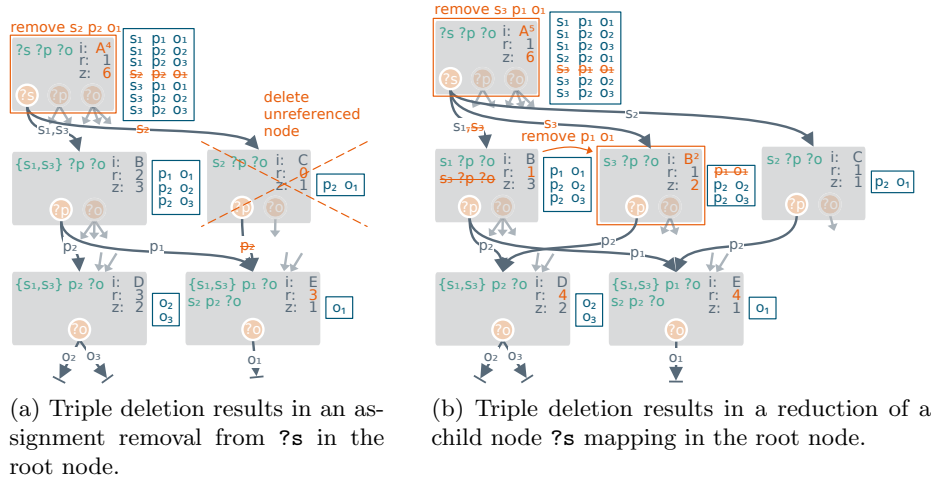


Fig. 3. Deleting single triples

*Insertion.* We begin by considering the insertion of a single triple. The process starts at the root node, where the inserted triple contributes a new solution to the fully unbound triple pattern  $?s ?p ?o$ . Consequently, each variable mapping of the root node needs to be updated by either adding a new assignment between a resource and a child node (see Figure 2a) or update the child node of an existing assignment (see Figure 2b).

The first scenario is illustrated in Figure 2a. Here, inserting the triple  $s_4 p_2 o_1$  adds the solution  $\{(?s, s_4), (?p, p_2), (?o, o_1)\}$ . This requires to add the assignment for  $s_4$  in the root node's mapping of  $?s$ , since it was not previously a solution for  $?s$ .  $s_4$  is assigned a hypertrie node representing the remaining part of the solution, i.e.,  $\{(?p, p_2), (?o, o_1)\}$ . Such a node already exists in the hypertrie, since the triple pattern  $s_2 ?p ?o$  has the same solution set. Consequently, we increment the reference count of the now shared node. Since the the node's variable mappings are not modified, no further propagation is necessary.

In Figure 2b, the subject resource  $s_2$  of the inserted triple  $s_2 p_2 o_2$  is assigned in the root node's mapping of  $?s$ . Since the assigned child node is referenced nowhere else, we can update that node directly by adding the remaining part of the solution  $\{(?p, p_2), (?o, o_2)\}$ . Again, updating the node for  $s_2 ?p ?o$  requires updating its variable mappings. For variable  $?p$ , the node assigned to  $p_2$  must be updated. However, since that node for the triple pattern  $s_2 p_2 ?o$  is still referenced elsewhere, this time we create and reference a copy instead of modifying it in place. Since the node is a leaf node, the update is completed afterward.

*Deletion.* The deletion of a single triple also requires distinguishing two cases, depending on the effect it has on the hypertrie's structure. In particular, to

reflect the removed solution an assignment for a variable mapping is either fully removed (see Figure 3a) or the assigned child is reduced (see Figure 3b).

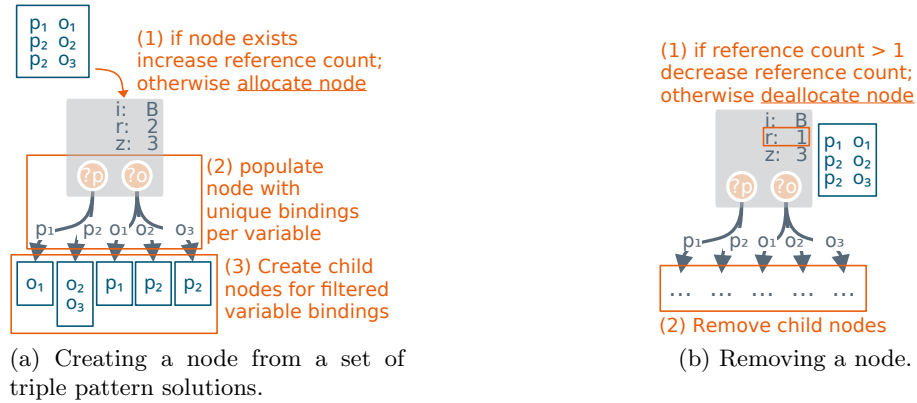
In the first case, the triple to be removed contains a resource that will no longer be a valid binding for one of the variables in the affected triple pattern. This is illustrated in Figure 3a for the triple  $s_2 p_2 o_1$ . In the root node, the subject  $s_2$  was assigned the hypertrie node representing  $s_2 ?p ?o$ . Since the triple being removed is the only solution with subject  $s_2$ , this child node would now represent an empty solution set. As a result, the assignment for  $s_2$  is removed from the root node’s mapping for variable  $?s$ , and the now-unreferenced child node is deleted. Deleting the node for  $s_2 ?p ?o$ , propagates reference count decreases that are applied to its child nodes. As its previous child node with identifier E is still referenced by other nodes, it is not deleted.

In the second case, the removed triple reduces the number of solutions in a child node with multiple solutions, such that the resulting child retains at least one solution. This is illustrated in Figure 3b, where the triple  $s_3 p_1 o_1$  is removed. The subject  $s_3$  remains a valid binding for  $?s$ , so its assignment in the root node’s mapping for variable  $?s$  is preserved but the assigned child needs to be reduced. Note, that the  $s_3$  and  $s_1$  were originally both assigned to the same child node with identifier B. Consequently, node B is still referenced and thus must not be modified in place; instead, a copy is created and updated to reflect the removal. Specifically, its assignment for predicate  $p_1$  is removed to form the new node  $B^2$ . For all assignments of that remain unchanged between B and  $B^2$ —e.g., the assignment  $p_2$  node’s mapping for variable  $?p$ —the reference counts of its corresponding child nodes are incremented to reflect the additional reference from their newly created parent node. The update terminates once all leaf nodes have been updated accordingly.

To summarize, updates to the hypertrie for a single triple alter the root node’s mapping for variable—either by introducing a new assignment (insertion), removing an existing one (deletion), or modifying child node an assignment refers to (insertion or deletion). Updates are propagated recursively to child nodes. When modifying a child node, a copy-on-write mechanism is applied to preserve sharing: if the node is still referenced elsewhere, it is copied before modification; otherwise, it can be edited in place. Reference counts are adjusted to reflect node creation, deletion, and changes in variable mappings.

## 5.2 Creating and Deleting Nodes

As shown in Section 5.1, adding or removing variable mappings may trigger the creation or deletion of depth-2 and depth-1 nodes. This section briefly outlines the steps required for creating and deleting such nodes. These steps are used for both update operations using single triples and bulk updates (Section 5.3). The same mechanism also applies when creating or deleting the root node. Figure 4a illustrates the process of creating a node from a set of triple pattern solutions. The creation proceeds in three steps: (1) If an equivalent node already exists, it is reused by incrementing its reference count and the process terminates. (2) If such a node does not exist, a new node is allocated. For each variable of the

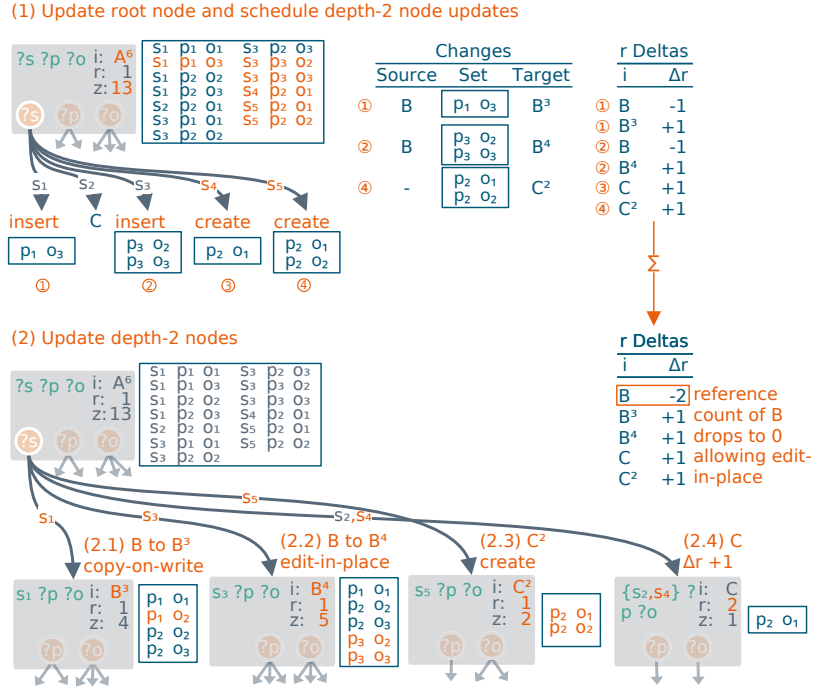


**Fig. 4.** Creating and deleting of nodes.

solutions, a variable mapping is populated assigning the unique resources for the respective variable to child nodes. (3) To ensure that all assigned child nodes exist and that they have correct reference counts, their creation is triggered recursively as described in Section 4. Figure 4b illustrates the process of deleting a node, which is carried out in two steps: (1) If the reference count of the node is greater than one, it is decremented and the process terminates. (2) Otherwise, the node is deallocated. The deletion is then recursively propagated to all child nodes referenced by the removed node.

### 5.3 Bulk Updates

Database indices often suffer from write amplification when applying numerous small updates [7]. In the hypertrie, for example, each update requires modifications to each variable mapping in the root node. By applying insertions or deletions in bulk, this overhead is incurred only once, thereby reducing write amplification. We generalize the single-triple update mechanism to handle multiple inserted or deleted triples at once. Technically, these triple modifications translate to solution modifications applied to the root node and recursively propagated changes to child nodes. The core idea is to group modified solutions in the same way that variable mappings are organized in the hypertrie, and to use these groups to update the variable mappings and child nodes. Specifically, the following steps are taken for each variable  $v$  that is found in the triple patterns of a hypertrie node. First, the solutions are grouped by their binding value  $b$  for  $v$ , forming a group  $U$  for each unique resource  $b$ . Since the group  $U$  will be used to create, delete, or update a child node, the binding for  $v$  is removed from its solutions. For example, given the variable  $?p$  and the solutions  $\{((?p, p_1), (?o, o_1)), ((?p, p_1), (?o, o_2)), ((?p, p_2), (?o, o_1))\}$ , the solutions for the grouping keys  $p_1$  and  $p_2$  are solutions  $\{((?o, o_1)), ((?o, o_2))\}$  and  $\{((?o, o_1))\}$ , respectively. Bulk insertions work similarly to insertions involving a single triple. If a resource  $b$  is not yet assigned for variable mapping  $v$ , a new



**Fig. 5.** Example of a bulk insertion execution. The steps shown include updating the root node, scheduling changes to depth-2 nodes, and applying those changes. Further recursive propagation is omitted from the figure.

assignment is added that points to a newly created node initialized with group  $U$ . If  $b$  is already assigned to a node  $X$ , a new node  $X^1$  is created by adding  $U$  to  $X$ , and  $b$  is reassigned to  $X^1$ . For deletions, the outcome depends on how many solutions are removed. If the number of solutions in  $U$  equals the total number of solutions in the child node  $X$  assigned to  $b$  (i.e.,  $|U| = |X|$ ), the assignment is removed and child node deleted. Otherwise, a new node  $X^2$  is created by removing  $U$  from  $X$ , and  $b$  is reassigned to  $X^2$ . Our algorithm proceeds in a level-wise manner: for each node at the current level, it updates the variable mappings and collects the resulting update requests for its child nodes. Once all nodes at a particular level have been processed, the collected updates for the level below are applied.

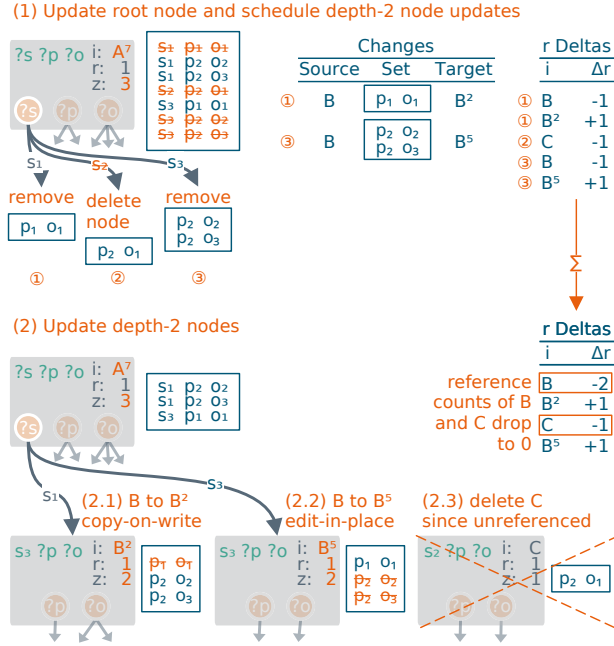
**Planning and Scheduling Changes** Planning changes targets two main objectives: (1) cache-conscious execution by applying updates originating from the same node consecutively, and (2) reduction of memory overhead by reusing nodes that become unreferenced.

To this end, we collect a set of changes. Each change consists of a source node identifier, a set of affected solutions, and a target node identifier, which

is computed from the source node identifier and the set of affected solutions. In parallel, reference count deltas are collected per change. For each change, a decrement is recorded for the source node and an increment for the target node. For insertions and removals that reduce but do not delete a child node, all three change components are recorded. If a node is newly created, the source node identifier is omitted from the change. Node deletions and removals of all solutions from a node are not recorded as changes, but only as reference count decrements. Furthermore, if the target node already exists or is scheduled for creation by another change, no change is recorded and only the corresponding reference count deltas are collected. Once the collection is completed, changes are scheduled as follows. Changes are executed grouped by their source node, and reference count increments for target nodes are applied alongside each change. The reference count decrement for a source node is applied with the final change in its group. If the reference count of a source drops to zero, the source node is modified in place since that is considered cheaper than copying it; otherwise, changes are applied using copy-on-write. Finally, any remaining reference count deltas are applied. If a reference count drops to zero as a result of this, the respective node is deleted.

Figure 5 illustrates the planning and execution of a bulk insertion. The insertion begins by adding solutions to the root node. For variable  $?s$ , this involves updating the child nodes associated with  $s_1$  and  $s_3$ , as well as creating new associations for the resources  $s_4$  and  $s_5$ . For  $s_1$ , a single solution must be inserted into node B. Since the resulting node  $B^3$  does not yet exist, a change is scheduled. For  $s_3$ , two new solutions are inserted into node B creating the node  $B^4$ . Both scheduled changes contribute a decrement to the reference count of B, and an increment to the target nodes  $B^3$  and  $B^4$ , respectively. For  $s_4$ , a node C is required. However, since C already exists, no new change is scheduled—only a reference count increment is recorded. In contrast, for  $s_5$ , the required node  $C^2$  does not yet exist. A change is recorded with no source node (indicating creation), and a reference count increment is added for  $C^2$ . In the next phase, the changes to the depth-2 nodes are applied. The first change inserts into node B, yielding node  $B^3$ . Since further operations use B as a source, the insertion is performed via copy-on-write. The next change also inserts into B, producing node  $B^4$ , and is the last one with B as its source. As applying the reference count delta drops B to zero, it is modified in place. Node  $C^2$  is created from scratch. The reference count increments for  $B^3$ ,  $B^4$ , and  $C^2$  are applied immediately during their respective executions. Finally, the remaining reference count deltas for nodes not involved in any scheduled change—here, only C—are applied. Further recursive propagation into scheduling and applying depth-1 changes takes place in the actual update process but is not shown in this example.

Figure 6 illustrates the planning and execution of a bulk removal. The process begins by deleting solutions from the root node. For the variable mapping of  $?s$ , this affects the assignments for  $s_1$ ,  $s_2$ , and  $s_3$ . For  $s_1$ , a single solution is removed from child node B. Since the resulting node  $B^2$  does not yet exist, a change is scheduled. For  $s_3$ , two solutions are removed from node B, producing the node



**Fig. 6.** Example of a bulk removal execution. The steps shown include updating the root node, scheduling changes to depth-2 nodes, and applying those changes. Further recursive propagation is omitted from the figure.

$B^5$ . Both scheduled changes contribute a decrement to the reference count of B, and an increment to the reference counts of  $B^2$  and  $B^5$ , respectively. In the case of  $s_2$ , all solutions are removed from node C and, as a result, the assignment is removed. Ultimately, for node C, a reference count decrement is recorded; however, no change is scheduled. In the next phase, the changes to the depth-2 nodes are applied. The first change removes a solution from B and creates node  $B^2$  via copy-on-write, as B is still needed for the next operation. The following change also removes solutions from B, producing node  $B^5$ , and is the last one with B as its source. Since the reference count of B drops to zero after this change, it is edited in place to become  $B_5$ . The reference count increments for  $B^2$  and  $B^5$  are applied immediately during their respective executions. Finally, the remaining reference count deltas for nodes not involved in any scheduled change—here, only C—are applied. As node C becomes unreferenced, it is deleted. Again, the further recursive propagation is omitted from the example.

**Handling Singleton Optimizations** The application of singleton node optimizations—namely, single entry nodes (SENs) for depth-2 and in-place storage for depth-1—requires minor adjustments in both change planning and execution. In the following we refer to regular hypertrie nodes, which have been covered above, as full nodes (FNs) in contrast to SENs or in-place stored nodes.

Unlike FNs, which can be modified to represent different sets of solutions, SENs can only be created or deleted. Accordingly, SEN creation and reference count tracking are handled separately. Whenever a change involves both a FN and a SEN, it is split into two self-contained operations: a FN change, and a SEN-specific operation consisting of a reference count delta and, if necessary, a planned SEN creation. For example, transitioning from a SEN to a FN requires adding the SEN’s single solution to the change set for the FN. The recorded change is then a creation, i.e., the source identifier is omitted. Conversely, transitioning from a FN to a SEN involves computing the remaining single solution and scheduling the creation of a new SEN from it. This is necessary since SEN changes are scheduled after FN changes so the FN might not be available any more by the time the SEN is actually created. As with FNs, a SEN is deleted when its reference count drops to zero. Handling singleton depth-1 nodes follows the same principles as SENs in relation to FNs. However, since singleton depth-1 nodes are not shared, they can be created and deleted immediately, without requiring change scheduling or reference counting.

**Runtime Complexity** A depth- $d$  the hypertrie encoding a set of  $z$  tuples with all optimizations from [5] requires  $\mathcal{O}(z \cdot 2^{d-1} \cdot d)$  space. The runtime complexity of applying (inserting or deleting) a changeset  $\Delta$  of  $d$ -tuples to a depth- $d$  hypertrie is bound by the space complexity  $\mathcal{O}(|\Delta| \cdot 2^{d-1} \cdot d)$  of a surrogate hypertrie that encodes the change set changeset  $\Delta$ . The proofs are provided in the supplementary material.

#### 5.4 Comparison to the Baseline & Implementation

Our algorithm builds on the hypertrie implementation from [5], which includes an insertion procedure not described in the paper, but lacks support for deletions. While that insertion algorithm also follows a top-down approach, it does not separate phases making it unsuitable for extension to support removals. In our algorithm, changes are planned first based on the previous state of the hypertrie context and then executed without further consulting the hypertrie context. This allows to re-use common sub-routines for insertion and deletion in the implementations and makes it easier to follow as it is dividing it into clearly separated steps.

Our implementation TETRIS-ID includes our extended hypertrie implementation and the necessary changes in TETRIS to support updates, i.e., adding support for SPARQL INSERT/DELETE DATA update queries and implementing a reader-writer lock on the hypertrie to ensure atomicity and isolation during updates. We used the latest code versions from the repositories referenced in [5]. Since the publication of [5], the code was updated, most notably by enhancing RDF support through the `rdf4cpp`<sup>7</sup> library, and by moving from in-memory to persistent disk-based indexing by leveraging the Metall allocator [13].

<sup>7</sup> <https://github.com/tentris/rdf4cpp>

## 6 Evaluation

For our evaluation, we focus on pure online update performance and hence solely use INSERT/DELETE DATA queries. We use a real-world scenario based on DBpedia changelogs and a scenario based on Wikidata that tests the benchmarked systems’ behavior with increasing update sizes. All experiments were executed on a Debian 11 Server with an AMD EPYC 7713 CPU, 1TB DDR4 octa-channel RAM at 2933MT/s and a 15.36TB KIOXIA CD6-R SSD with an ext4 filesystem. The supplementary material includes benchmarking results confirming that TENTRIS and TENTRIS-ID have equivalent bulk-loading performance and storage efficiency.

### 6.1 Systems and Experimental Setup

We compare the performance of our implementation (TENTRIS-ID) with the following triple stores that are freely available for benchmarking:

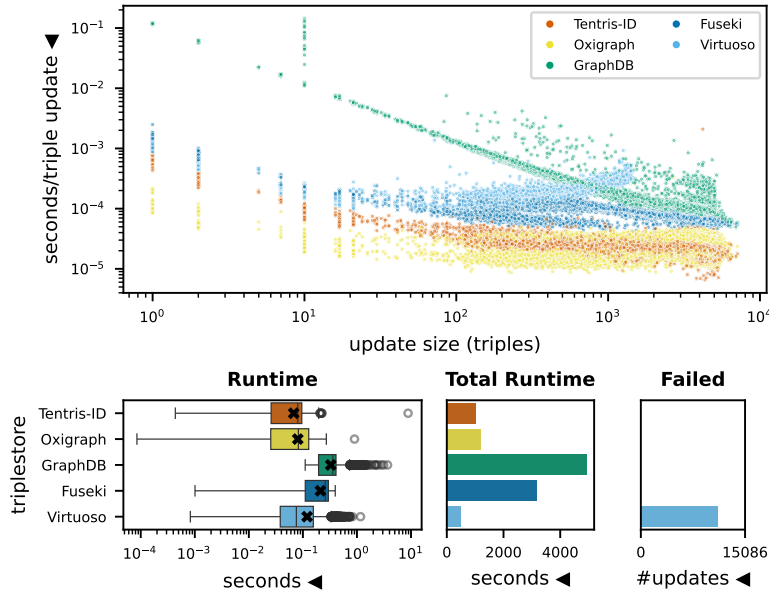
(i) Fuseki 5.2.0, (ii) GraphDB 10.6.2 (free version), (iii) Virtuoso 7.2.12, and (iv) Oxigraph 0.4.7. MilleniumDB [17] and QLever [4] were excluded since they failed or crashed on most of the updates. Jena-LFJ [12] was excluded from the evaluation because it relies on Fuseki’s TDB backend with additional indices enabled, which inevitably makes it slower than Fuseki. Our experiments were carried out over HTTP, according to SPARQL’s protocol ”update via POST directly” [9]. Before each experiment, the triple stores went through a warmup phase for which we sent the queries from the benchmarks of [5].

### 6.2 DBpedia Update Logs

We start with a snapshot of DBpedia from October 2015 with 860M triples which we date stamped to October 1st, 2015 and execute the updates reported in the official DBpedia changelogs from October 2015. Based on the triples inserted and deleted in each changelog entry, one INSERT DATA and one DELETE DATA are created, resulting in a total of 15,086 updates. Figure 7 shows the update performance of the evaluated triplestores. The top row plots the update time per triple against the update size, while the bottom row summarizes the runtime distribution, total runtime, and failed updates.

Overall, TENTRIS-ID and Oxigraph are the fastest, followed by Fuseki. GraphDB shows longer update durations, particularly for smaller update sizes, but improves with larger updates. All triplestores but Virtuoso perform better with larger update sizes, likely due to the amortization of HTTP and transactional overheads. Virtuoso’s performance is comparable to that of Fuseki for update sizes up to 100 triples but slows down considerably for larger updates. Additionally, Virtuoso rejected<sup>8</sup> all updates exceeding 1,359 triples, resulting in 11,090 (73%) rejected updates. Oxigraph rejected 18 queries (0.12%). In terms of

<sup>8</sup> Its SPARQL-to-SQL mapping has a hard-coded limit (see <https://github.com/openlink/virtuoso-opensource/blob/34c367/libsrc/Wi/sparql2sql.h#L1031>).

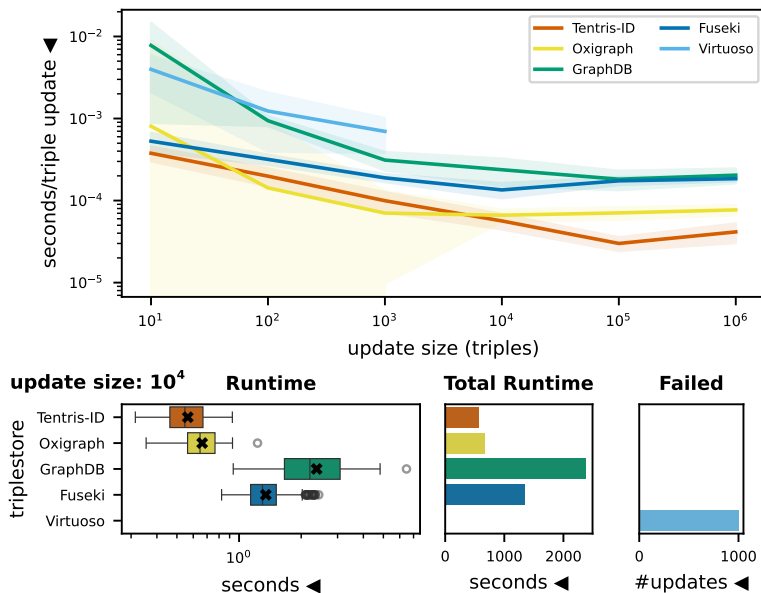


**Fig. 7.** The triple stores’ runtime performance at replaying DBpedia’s official update logs. Lower values are better on all subplots as indicated by the ◀. The top plot shows the update size versus the time spent per updated triple. The bottom row provides summary statistics: The left panel is a standard boxplot on the update runtimes with an additional cross for the mean. The middle panel compares the total cumulative runtime and the right panel shows the number of failed updates per system.

mean update time, TETRIS-ID consistently outperforms or matches the other systems with a mean ( $\pm$  standard deviation) runtime of 0.067 ( $\pm 0.081$ ) seconds (s). Oxigraph performs similarly with 0.074 ( $\pm 0.049$ )s. The other triplestores are slower, with Virtuoso at 0.12 ( $\pm 0.12$ )s (not comparable due to 73% rejected updates), Fuseki at 0.21 ( $\pm 0.11$ )s, and GraphDB at 0.33 ( $\pm 0.15$ )s. For total runtime see Figure 7.

### 6.3 Wikidata

In our second evaluation, we assess scalability with update sizes up to 10<sup>6</sup> triples using the larger Wikidata trusty dataset (November 11, 2020), which contains 5.5B triples. For this scenario, SPARQL updates were generated from randomly sampled triples. We generated batches of 500 insert and 500 delete updates of sizes 10<sup>1</sup>, 10<sup>2</sup>, 10<sup>3</sup> and 10<sup>4</sup>. For the larger update sizes, we created 100 batches of size 10<sup>5</sup> and 10 batches of size 10<sup>6</sup> for both insert and delete updates. The triples from the insert queries were removed from the dataset before it was loaded into the triple stores. Like in our first experiment, the queries were executed alternating between insert and delete.



**Fig. 8.** The triple stores’ runtime performance at applying updates to Wikidata with 5.5B triples. The top plot shows scaling update size from  $10^1$  to  $10^6$  triples, bands represent one standard deviation. For missing Virtuoso data see Footnote 8. The bottom row provides summary statistics for the batch of updates of size  $10^4$ . For a detailed descriptions of the row layout see Figure 7.

The top plot of Figure 8 shows the mean update time per triple as a function of the update size. Consistent with the findings from the DBpedia scenario, all systems benefit from update sizes. However, this improvement plateaus, for update sizes exceeding  $10^3$  in Oxigraph,  $10^4$  in Fuseki and  $10^5$  in GraphDB and TETRIS-ID. Virtuoso<sup>8</sup> is again unable to process updates larger than  $10^3$ . TETRIS-ID achieves the overall best performance of all systems for update sizes of  $10^4$  or more. In particular, it is more than 5.6 times faster than Fuseki and GraphDB on updates of size  $10^5$  and  $10^6$ . For small and large updates (sizes 10,  $10^5$  and  $10^6$ ) TETRIS-ID performs about 2 times faster than Oxigraph. For middle-sized updates (sizes  $10^3$ ,  $10^4$  and  $10^5$ ), both systems perform similar within a 30% range. We further compare TETRIS-ID with the second-best performing systems Oxigraph by measuring the performance of both systems during the warmup phase, which executed 495 SPARQL SELECT queries. In our experiments, this took Oxigraph over 9 hours where as TETRIS-ID finished in 43 minutes. Notably, Oxigraph’s storage footprint is also 25–54% larger than that of TETRIS-ID. Hence, we conclude that our extension of the hypertrie with updates leads to the first WCOJ-enabled data structure that outperforms the state of the art both on SELECT (as shown in [5, 6]) and INSERT/DELETE queries.

## 7 Conclusion and Future Work

We present the first, to the best of our knowledge, efficient online update algorithm for RDF triplestores that fully support worst-case optimal joins (WCOJs). Our solution enables both insertion and deletion of triples while preserving the performance benefits of WCOJs, achieving a crucial milestone in making WCOJs viable for production-ready triple stores. Our evaluation demonstrated that the proposed methods deliver update performance on live updates—scaling up to 1M triples—comparable to or better than traditional triple store systems on large graphs such as DBpedia and Wikidata.

While this work focused on the update operations of the hypertrie, future work should aim to optimize the update process further. The current implementation does not support concurrent reads during updates. In the future, we plan to investigate approaches that allow a hypertrie to remain accessible for reads while updates generate a new version. A possible direction that we plan to explore is deferred materialization where nodes track deltas relative to other hypertrie nodes, thus enabling incremental updates with minimal disruption to ongoing reads. Further, a systematic study of runtime under specific update patterns, e.g., different insertion orders, could be executed to identify possible performance pitfalls.

*Supplemental Material Statement:* Source code for our modified version of the hypertrie and TENTRIS-ID are available from Github<sup>9</sup>. Raw benchmarking results, bulk-loading performance and storage efficiency results, a technical report for the implemented algorithm, scripts to setup and run the benchmarks, and scripts to generate the figures are available from GitHub<sup>10</sup>.

*Acknowledgements:* This work has been supported within the project "TENTRIS" (03EFNW0356) funded under the EXIST Transfer of Research programme by the German Federal Ministry for Economic Affairs and Energy (BMWE) and co-financed by the European Social Fund. This work has been supported by the German Federal Ministry of Research, Technology and Space (BMFTR) within the project NEBULA under the grant no 13N16364. This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101070305. This work has been supported within the project "WHALE" (LFN 1-04) funded under the Lamarr Fellow Network programme by the Ministry of Culture and Science of North Rhine-Westphalia (MKW NRW).

<sup>9</sup> <https://github.com/dice-group/hypertrie> and <https://github.com/dice-group/tentris-research-project>

<sup>10</sup> <https://github.com/dice-group/tentris-wcoj-with-updates>

## References

1. Aberger, C.R., Tu, S., Olukotun, K., Ré, C.: Old techniques for new join algorithms: A case study in rdf processing. In: 2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW). pp. 97–102 (2016)
2. Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A.: Worst-case optimal graph joins in almost no space. In: Proceedings of the 2021 International Conference on Management of Data. pp. 102–114 (2021)
3. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. In: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science. p. 739–748. FOCS '08, IEEE Computer Society, USA (2008)
4. Bast, H., Buchhold, B.: Qlever: A query engine for efficient sparql+text search. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. p. 647–656. CIKM '17, Association for Computing Machinery, New York, NY, USA (2017)
5. Bigerl, A., Conrads, L., Behning, C., Saleem, M., Ngonga Ngomo, A.C.: Hashing the hypertrie: Space- and time-efficient indexing for sparql in tensors. In: Sattler, U., Hogan, A., Keet, M., Presutti, V., Almeida, J.P.A., Takeda, H., Monnin, P., Pirrò, G., d'Amato, C. (eds.) The Semantic Web – ISWC 2022. pp. 57–73. Springer International Publishing, Cham (2022)
6. Bigerl, A., Conrads, L., Behning, C., Sherif, M.A., Saleem, M., Ngonga Ngomo, A.C.: Tentrism – A Tensor-Based Triple Store. In: Pan, J.Z., Tamma, V., d'Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) The Semantic Web – ISWC 2020. pp. 56–73. Springer International Publishing, Cham (2020)
7. Brodal, G.S., Fagerberg, R.: Lower bounds for external memory dictionaries. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. p. 546–554. SODA '03, Society for Industrial and Applied Mathematics, USA (2003)
8. Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J.J.: RDF 1.1 Concepts and Abstract Syntax. <https://www.w3.org/TR/rdf11-concepts/> (Feb 2014), w3C Recommendation
9. Feigenbaum, L., Williams, G., Clark, K., Torres, E.: SPARQL 1.1 Protocol. <https://www.w3.org/TR/sparql11-protocol/> (Mar 2013), w3C Recommendation
10. Harris, S., Seaborne, A., Prud'hommeaux, E., Feigenbaum, L.: SPARQL 1.1 Query Language. <https://www.w3.org/TR/sparql11-query/> (Mar 2013), w3C Recommendation
11. Hogan, A., Riveros, C., Rojas, C., Soto, A.: 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. Lecture Notes in Computer Science, vol. 11778, pp. 258–275. Springer (2019)
12. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., Cruz, I., Hogan, A., Song, J., Lefrançois, M., Gandon, F. (eds.) The Semantic Web – ISWC 2019. pp. 258–275. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019)
13. Iwabuchi, K., Youssef, K., Velusamy, K., Gokhale, M., Pearce, R.: Metall: A persistent memory allocator for data-centric analytics. *Parallel Computing* **111**, 102905 (2022)

14. Karalis, N., Bigerl, A., Demir, C., Heidrich, L., Ngonga Ngomo, A.C.: Evaluating negation with multi-way joins accelerates class expression learning. In: Bifet, A., Davis, J., Krilavičius, T., Kull, M., Ntoutsis, E., Žliobaitė, I. (eds.) *Machine Learning and Knowledge Discovery in Databases. Research Track*. pp. 199–216. Springer Nature Switzerland, Cham (2024)
15. Karalis, N., Bigerl, A., Heidrich, L., Sherif, M.A., Ngonga Ngomo, A.C.: Efficient evaluation of conjunctive regular path queries using multi-way joins. In: Meroño Peñuela, A., Dimou, A., Troncy, R., Hartig, O., Acosta, M., Alam, M., Paulheim, H., Lisena, P. (eds.) *The Semantic Web*. pp. 218–235. Springer Nature Switzerland, Cham (2024)
16. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. *J. ACM* pp. 16:1–16:40 (2018)
17. Vrgoč, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J.: MillenniumDB: An open-source graph database system. *Data Intelligence* **5**(3), 560–610 (Aug 2023)