New Compressed Indices for Multijoins on Graph Databases

Diego Arroyuelo[a,d], Fabrizio Barisione[b,d], Antonio Fariña[c,e], Adrián Gómez-Brandón[c,d], Gonzalo Navarro[b,d]

[a] *Escuela de Ingeniería, Pontificia Universidad Católica de Chile, Av. Vicuña Mackenna, Santiago, Chile*
[b] *Universidad de Chile, Beauchef, Santiago, Chile*
[c] *Universidade da Coruña, Campus de Elviña, A Coruña, Spain*
[d] *IMFD, Av. Vicuña Mackenna, Santiago, Chile*
[e] *CITIC, Campus de Elviña, A Coruña, Spain*

Abstract

A recent surprising result in the implementation of worst-case-optimal (WCO) multijoins in graph databases (specifically, basic graph patterns) is that they can be supported on graph representations that take even less space than a plain representation, and orders of magnitude less space than classical indices, while offering comparable performance. In this paper we uncover a wide set of new WCO space-time tradeoffs: we (1) introduce new compact indices that handle multijoins in WCO time, and (2) combine them with new query resolution strategies that offer better times in practice. As a result, we improve the average query times of current compact representations by a factor of up to 13 to produce the first 1000 results, and using twice their space, reduce their total average query time by a factor of 2. Our experiments suggest that there is more room for improvement in terms of generating better query plans for multijoins.

Keywords: Worst-case-optimal, multijoins, graph databases, compact data structures

Email address: adrian.gbrandon@udc.es (Adrián Gómez-Brandón)

1. Introduction

Natural joins are fundamental in the relational algebra, and generally the most costly operations. A bad implementation choice can lead to unaffordable query times, so they have been a concern since the beginnings of the relational model. Apart from efficient algorithms to join two tables (i.e., solve pair-wise joins), database management systems sought optimized strategies (e.g., Selinger et al. (1979)) to solve joins between several tables (i.e., multijoins), since performance differences between good and bad plans could be huge. A multijoin *query plan* was a binary tree where the leaves were the tables to join and the internal nodes were the pair-wise joins to perform.

After half a century of revolving around this pairwise-join-based strategy, it was found that it had no chance to be optimal (Atserias et al., 2013), as it could generate intermediate results (at internal nodes of the expression tree) that were much larger than the final output. The concept of a *worst-case optimal (*WCO*)* algorithm (Atserias et al., 2013) was coined to define a multijoin algorithm taking time proportional to the largest possible output size of the query on any database. Several WCO join algorithms were proposed since then (Ngo et al., 2012, 2013; Khamis et al., 2016; Nguyen et al., 2015; Ngo, 2018). The simplest and most popular of those, *Leapfrog Triejoin (LTJ)* (Veldhuizen, 2014), can be regarded at a high level as reducing the multijoin by one *attribute* at a time, instead of by one *relation*.

A serious problem of LTJ and all other existing WCO algorithms is their high space usage, however. This has become an obstacle to their full adoption in database systems. In this paper we are interested in the use of WCO algorithms on *graph databases*, which can be regarded as labeled graphs, or as a single relational table with three attributes: source node, label, and target node. Standard query languages for graph databases like SPARQL (Harris et al., 2013) feature most prominently *basic graph patterns (BGPs)*, which essentially are a combination of multijoins and simple selections. The concept of WCO algorithms can be translated into solving BGPs on graph databases (Hogan et al., 2019). WCO algorithms are very relevant because typical BGPs correspond to large and complex multijoins (Nguyen et al., 2015; Aberger et al., 2017; Kalinsky et al., 2017; Hogan et al., 2019), where non-WCO algorithms can be orders of magnitude slower than WCO ones (Aberger et al., 2017). The implementation of various WCO indices for graph databases seems to confirm that large space usage will be the price to offer WCO query times.

Surprisingly, recent research debunks this impression. In particular, the

ring (Arroyuelo et al., 2021, 2024) is a novel compact index that represents graph databases (the data *and* the index structures) within *less* space than that used by the raw data in plain form, while still supporting BGPs within competitive times, often even lower than much larger indices.

1.1. Our contribution

The unexpected result achieved by small indexes like the *ring* has opened numerous opportunities for new space-time tradeoffs in index data structures for WCO multijoins on graph databases. The *ring* was aimed at minimum space usage, to demonstrate that competitive query times could be achieved using only as much space as the raw data, and even less. Since this space is much lower than that of traditional indices, there is sufficient slack to introduce larger data structures that, still using a fraction of the space of those traditional indices, are much faster than the *ring*. Additionally, despite occupying minimal space, the data structures supporting the *ring* enable efficient computation of information—which would otherwise need to be explicitly stored by conventional indices—that helps compute efficient attribute elimination orders for LTJ (Arroyuelo et al., 2024). Motivated by this, we contribute with new compact indices that support solving BGPs in WCO time, and their combination with new query resolution techniques, thereby uncovering a wide set of new space-time tradeoffs in WCO indices for solving BGPs. Concretely:

1. We design an alternative to the *ring* that, using twice its space, is four times faster in the median and twice as fast on the average. This new index, which we call the *rdfcsa*, builds on an existing compact index representation based on text indexing concepts that only supported single joins (Brisaboa et al., 2023), so that now it supports full BGPs in WCO time. To achieve this, we implement LTJ on top of the *rdfcsa*, which requires significantly extending its basic functionality.
2. We combine the *ring* and the *rdfcsa* with an *adaptive* variable elimination order, which recomputes the best elimination order as the join proceeds and more information is available. We use new estimators for the next variable to bind that are more accurate than traditional ones. Those are computed efficiently by exploiting and extending the functionality of our compact indices.
3. We perform exhaustive experiments comparing our new variants with the original indices and with other state-of-the-art ones. These show,

for example, that our new adaptive variable elimination orders can speed up standard ones by a factor of 5, outperforming *every possible nonadaptive order*. Overall, the best variants of our new indices use from 0.6 to 2 times the plain data size (this space contains the data in compact form) and outperform every other index we tested from the state of the art, being outperformed only by an index that uses 12 times more space than the data. For example, we outperform the original *ring* by a factor up to 13 to produce the first 1000 results.

We focus on static indices that fit in main memory. Unlike most uses of relational databases, there are many applications where graph databases are static, or updated in bulk only periodically. One of the most widespread applications of graph databases are the Knowledge Graphs, which are typically costly to build and regarded as static at query time (Ji et al., 2022, Sec. VII.F). A case that has received much attention recently, for example, is the (costly) construction of Knowledge Graphs from text using LLMs (Pusch and Conrad, 2024; Sun et al., 2024). Another area where graphs are regarded as static is graph analytics, where the graph is created once in order to query it intensively (Hogan et al., 2020). As a third example, the SPARQL primitive `CONSTRUCT` allows extracting a graph from a main graph, which can be transmitted to the client's side and be queried locally; this is also a case where a succinct and functional representation of that (static) derived graph is beneficial. With respect to being in-memory, the main purpose of using compact indices is precisely to allow indexing larger graph databases in main memory, or to use the aggregated main memories of fewer machines.

2. Preliminary concepts

2.1. Graph joins

2.1.1. Edge-Labeled Graphs

Let $\mathcal{U}$ be a totally ordered, countably infinite set of *constants*, which we call the *universe*. In the RDF model (Manola and Miller, 2004), an *edge-labeled graph* is a finite set of *triples* $G \subseteq \mathcal{U}^3$, where each triple $(s, p, o) \in \mathcal{U}^3$ encodes the directed edge $s \xrightarrow{p} o$ from vertex s to vertex o, with edge label p. We call $\mathrm{dom}(G) = \{s, p, o \mid (s, p, o) \in G\}$ the subset of $\mathcal{U}$ used as constants in G. For any element $u \in \mathcal{U}$, let $u + 1$ denote the successor of u in the total order $\mathcal{U}$. We also denote $U = \max \mathrm{dom}(G)$. For simplicity, we will assume that the constants in $\mathcal{U}$ have been mapped to integers in the range $[1 \mathinner{..} U]$, and will even assume $\mathcal{U} = [1 \mathinner{..} U]$.

2.1.2. Basic Graph Patterns (BGPs)

A graph G is often queried to find patterns of interest, that is, subgraphs of G that are homomorphic to a given pattern Q. Unlike the graph G, which is formed only by constants in $\mathcal{U}$, a pattern Q can contain also *variables*, formally defined as follows. Let $\mathcal{V}$ denote an infinite set of variables, such that $\mathcal{U} \cap \mathcal{V} = \emptyset$. Then, a *triple pattern* t is a tuple $(s, p, o) \in (\mathcal{U} \cup \mathcal{V})^3$, and a *basic graph pattern* is a finite set $Q \subseteq (\mathcal{U} \cup \mathcal{V})^3$ of triple patterns. Each triple pattern in Q is an atomic query over the graph, equivalent to equality-based selections on a single ternary relation. Thus, a basic graph pattern (BGP) corresponds to a full conjunctive query (i.e., a *join query* plus simple selections) over the relational representation of the graph.

Let $\text{vars}(Q)$ denote the set of variables used in pattern Q. The *evaluation* of Q over a graph G is then defined to be the set of mappings $Q(G) := \{\mu : \text{vars}(Q) \to \text{dom}(G) \mid \mu(Q) \subseteq G\}$, called *solutions*, where $\mu(Q)$ denotes the image of Q under μ, that is, the result of replacing each variable $x \in \text{vars}(Q)$ in Q by $\mu(x)$.

2.2. Worst-case optimal joins

2.2.1. The AGM bound

A well-established bound to analyze join algorithms is the *AGM bound*, introduced by Atserias et al. (2013), which sets a limit on the maximum output size for a natural join query. Let Q denote such a query and D a relational database instance. The AGM bound of Q over D, denoted Q^*, is the maximum number of tuples generated by evaluating Q over any database instance D' containing a table R' for each table R of D, with the same attributes and $|R'| \leq |R|$ tuples. Though BGPs extend natural joins with self joins, constants in $\mathcal{U}$, and the multiple use of a variable in a triple pattern, the AGM bound can be applied to them by regarding each triple pattern as a relation formed by the triples that match its constants (Hogan et al., 2019).

Given a join query (or BGP) Q and a database instance D, a *join algorithm* enumerates $Q(D)$, the solutions for Q over D. A join algorithm is *worst-case optimal (*WCO*)* if it has a running time in $\tilde{O}(Q^*)$, which is $O(Q^*)$ multiplied by terms that do not depend, or depend only polylogarithmically, on $|D|$. Atserias et al. (2013) proved that there are queries Q for which no plan involving only pair-wise joins can be WCO.

This paper focuses on WCO algorithms, precisely on the one described next, which is the one most frequently implemented.

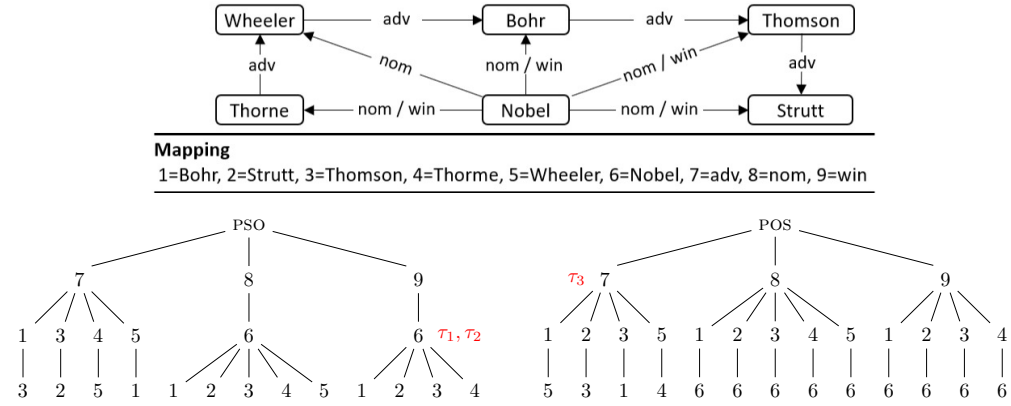

Figure 1: A labeled graph G' with its string to integer mapping and tries for orders PSO and POS. Arrows $x \xrightarrow{\text{adv}} y$ mean that y was the advisor of x, $x \xrightarrow{\text{nom}} y$ mean that y was nominated to prize x, and $x \xrightarrow{\text{win}} y$ that y won prize x.

2.2.2. Leapfrog TrieJoin (LTJ)

We describe the Leapfrog Triejoin algorithm (Veldhuizen, 2014), originally designed for natural joins in relational databases, as it is adapted for BGP matching on labeled graphs (Hogan et al., 2019).

Let $Q = \{t_1, \ldots, t_q\}$ be a BGP and $\text{vars}(Q) = \{x_1, \ldots, x_v\}$ its set of variables. LTJ uses a *variable elimination* approach, which extends the concept of attribute elimination. The algorithm carries out $v = |\text{vars}(Q)|$ iterations, handling one particular variable of $\text{vars}(Q)$ at a time. This involves defining a total order $\langle x_{i_1}, \ldots, x_{i_v} \rangle$ of $\text{vars}(Q)$, which we call a *VEO* for *variable elimination order*.

Each triple pattern t_i is interpreted as a relation that will be joined, and associated with a suitable trie τ_i. The root-to-leaf path in τ_i must start with the constants that appear in t_i, and the rest of its levels must visit the variables of t_i in an order that is consistent with the VEO chosen for Q (this is why we need the $3! = 6$ tries). Fig. 1 shows an example graph and the corresponding mapping of the constants in $\mathcal{U}$ to integers. We also show two tries representing the graph triples using the orders PSO (i.e., predicate, subject, object) and POS (ignore the marks τ_i for now). For example, we must use the trie PSO to handle a triple pattern $(x, 8, y)$ if the VEO is $\langle x, y \rangle$, and the trie POS if the VEO is $\langle y, x \rangle$. If Q has a second triple pattern $(y, 7, x)$, then we need both tries no matter the VEO we use.

The algorithm starts at the root of every τ_i and descends by the children that correspond to the constants in t_i, and then proceeds to the variable elimination phase. Let $Q_j \subseteq Q$ be the triple patterns that contain variable x_{i_j}. Starting with the first variable, x_{i_1}, LTJ finds each $c \in \mathrm{dom}(G)$ such that for every $t \in Q_1$, if x_{i_1} is replaced by c in t, the evaluation of the modified triple pattern t over G is non-empty (i.e., there may be answers to Q where x_{i_1} is equal to c). If the trie τ of t is consistent with the VEO, the children of its current node contain precisely those suitable values c for variable x_{i_1}.

During the execution, we keep a mapping μ with the solutions of Q. As we find each constant c suitable for x_{i_1}, we *bind* x_1 to c, that is, we set $\mu = \{(x_1 := c)\}$ and branch on this value c. In this branch, we go down by c in all the virtual tries τ such that $t \in Q_1$. We now repeat the same process with Q_2, finding suitable constants d for x_{i_2} and increasing the mapping to $\mu = \{(x_1 := c), (x_2 := d)\}$, and so on. Once we have bound all variables in this way, μ is a solution for Q (this happens many times because we branch on every binding to c, d, etc.). When it has considered all the bindings c for some variable x_{i_j}, LTJ backtracks and continues with the next binding for Q_{j-1}. When this process ends, all solutions for Q have been reported.

As an example, consider solving the query $Q = \{(6, 9, x), (6, 9, y), (x, 7, y)\}$ on the graph of Fig. 1, which looks for Nobel winners y and x such that y was the advisor of x. We decide to use the VEO $\langle y, x\rangle$. The tries τ_1 and τ_2 for the first two triple patterns are PSO, whereas the trie τ_3 for the triple $(x, 7, y)$ is POS. We first descend by the constants 9 and 6 in τ_1 and τ_2, and by the constant 7 in τ_3; the reached nodes are marked in the figure. Now the variable elimination phase starts with y, which involves the tries τ_2 and τ_3. We intersect the children of the current (i.e., marked) nodes of those tries, obtaining $\{1, 2, 3\}$, and proceed to bind y with each of those values.

1. With the binding $\mu = \{(y := 1)\}$, we descend by 1 from the marked nodes of τ_2 and τ_3. The next variable to bind, x, involves tries τ_1 and τ_3. We intersect the children of their current nodes and obtain the empty set, so we abandon this branch and return.
2. With the binding $\mu = \{(y := 2)\}$, we descend by 2 from the marked nodes of τ_2 and τ_3. We intersect the children of the current nodes of τ_1 and τ_3 and obtain $\{3\}$. We then extend the current binding to $\mu = \{(y := 2), (x := 3)\}$ and, since all the variables are bound, report the solution (i.e., Strutt and Thomson) and return.
3. With the binding $\mu = \{(y := 3)\}$, we descend by 3 from the marked

nodes of τ_2 and τ_3. We intersect the children of the current nodes of τ_1 and τ_3, obtaining $\{1\}$. We then extend the current binding to $\mu = \{(y := 3), (x := 1)\}$ and, since all the variables are bound, report the solution (i.e., Thomson and Bohr) and return.

Operationally, the values c, d, etc. are found by *intersecting* the children of the current nodes in all the tries τ_i for $t_i \in Q_j$. LTJ carries out the intersection using the primitive $\mathsf{leap}(\tau_i, c)$, which finds the next smallest constant $c_i \geq c$ within the children of the current node in trie τ_i; if there is no such value c_i, $\mathsf{leap}(\tau_i, c)$ returns a special value $\perp$.

2.3. Variable Elimination Orders (VEOs)

Veldhuizen (2014) showed that if $\mathsf{leap}()$ runs in polylogarithmic time, then LTJ is WCO no matter the VEO chosen, as long as the tries used have the right attribute order. In practice, however, the VEO plays a fundamental role in the efficiency of the algorithm (Veldhuizen, 2014; Hogan et al., 2019). A VEO yielding a large number of intermediate solutions that are later discarded during LTJ execution, will be worse than one that avoids exploring many such alternatives. One would prefer, in general, to first eliminate selective variables (i.e., the ones that yield a smaller candidate set when intersecting).

A heuristic to generate a good VEO in practice (Hogan et al., 2019; Arroyuelo et al., 2021; Vrgoc et al., 2023) computes, for each variable x_j, its minimum weight

$$w_j = \min\{w_{ij} \mid x_j \text{ appears in triple } t_i\}, \tag{1}$$

where w_{ij} is the *weight* of x_j in t_i. The VEO sorts the variables in increasing order of w_j, with a couple of restrictions: (i) each new variable should share some triple pattern with a previous variable, if possible; (ii) variables appearing only once in Q (called *lonely*) must be processed at the end.

To compute w_{ij}, we (temporarily) choose a trie τ_j where x_j appears right after the constants of t_i, and descend in τ_j by the constants. The number of children of the trie node v we have reached is the desired weight w_{ij}. This is the size of the list in τ_i to intersect when eliminating x_j.

In this paper we explore the use of *adaptive* VEOs, which are defined progressively as the query processing advances, and may differ for each different binding of the preceding variables. ADOPT (Wang et al., 2023) is the first system combining LTJ with adaptive VEOs. The next variables to bind

are chosen using reinforcement learning, by partially exploring possibly upcoming orders, and balancing the cost of exploring with that of the obtained improvements. Our adaptive VEOs will be computed, instead, simply as a variant of the formula presented above for global VEOs (Hogan et al., 2019).

We will also explore more refined estimations of w_j in Eq. (1), beyond the use of simply the minimum of the set sizes w_{ij} to estimate the size of their intersection.

3. The Ring: wco joins in compact space

The *ring* (Arroyuelo et al., 2021, 2024) is an index that supports the 6 orders needed by LTJ using a single data structure that uses space close to the raw data representation (and possibly less), while supporting the $\mathsf{leap}()$ operation on the tries in logarithmic time.

3.1. Bitvectors and wavelet trees

We start surveying the compact data structures used by the *ring*. First, a *bitvector* $B[1\,..\,n]$ is an array of n bits supporting the following queries:

- $\mathsf{access}(B, i)$: the bit stored at $B[i]$.
- $\mathsf{rank}_b(B, i)$: the number of bits $b \in \{0, 1\}$ in $B[1\,..\,i]$.
- $\mathsf{select}_b(B, j)$: the position of the jth occurrence of bit $b \in \{0, 1\}$ in B.
- $\mathsf{selectnext}_b(B, j)$: the position of the leftmost occurrence of b in $B[j\,..\,n]$.

These operations can be supported in $O(1)$ time using $n + o(n)$ bits of space (Clark, 1996; Munro, 1996) or, alternatively, $nH_0(B) + o(n)$ bits (Raman et al., 2007), where $H_0(B) \leq 1$ denotes the zero-order entropy of B.

The wavelet tree (Grossi et al., 2003; Navarro, 2014) is a binary tree that represents a string $S[1\,..\,n]$ of symbols from an alphabet $\Sigma = \{1, \ldots, \sigma\}$. Each node v represents a range $[a, b]$ of the alphabet, $a, b \in \Sigma$, with the root representing the whole alphabet $[1, \sigma]$ and each leaf representing a single symbol a, or range $[a, a]$. The range $[a, b]$ of internal nodes is divided into two, $[a, \lfloor (a + b)/2 \rfloor]$ and $[\lfloor (a + b)/2 \rfloor + 1, b]$, which are those of their left and right children.

Each internal node v representing a range $[a, b]$ is associated with the subsequence $S_{a,b}$ of S formed by the symbols in $[a, b]$ ($S_{a,b} = S$ if the node is the root). Instead of storing $S_{a,b}$, the node stores a bitvector $B_{a,b}[1\,..\,|S_{a,b}|]$,

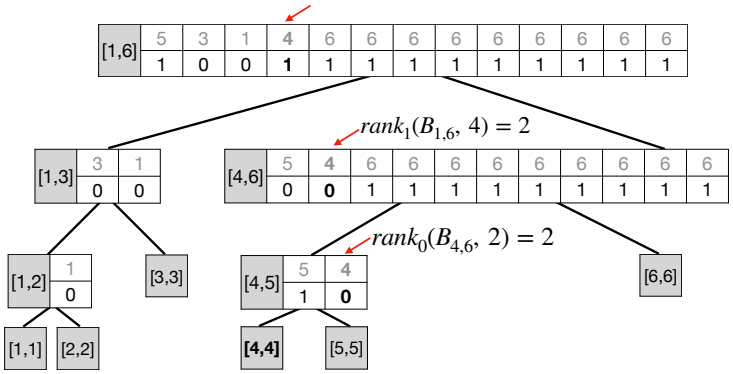

Figure 2: Example of the wavelet tree for the sequence $\{5, 3, 1, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6\}$. The ranges at the left depict the alphabet range of each bitmap. The arrows show the procedure to obtain the 4th value of the sequence.

where $B_{a,b}[i] = 0$ iff $S_{a,b}[i] \in [1, \lfloor (a+b)/2 \rfloor]$ (i.e., belongs to the first half of the alphabet range); else $B_{a,b}[i] = 1$.

Note that the bitvector lengths at any level of the tree sum up to n and we need to support binary rank/select operations on them. Therefore, the wavelet tree represents S using $n \lg \sigma + o(n \lg \sigma)$ bits (a plain representation uses almost the same, $n \lg \sigma$ bits), and even within zero-order entropy, $nH_0(S) \leq n \lg \sigma$ bits. For large alphabets (as occurs in this paper), the additional space for the $O(\sigma)$ tree pointers are eliminated in a pointerless version called wavelet matrix (Claude et al., 2015).

The wavelet trees support the functionality of access, rank, and select on general alphabets in time $O(\lg \sigma)$ by traversing the tree from the root to a leaf. For instance, to access $S[i]$ we start at position i in the bitmap of the root $B_{1,\sigma}$. Depending on $B_{1,\sigma}[i]$ we know that $S[i]$ is represented in the left (0) or right child (1). Hence, we continue by the left (resp. right) child at position $rank_0(B_{1,\sigma}, i)$ (resp., $rank_1(B_{1,\sigma}, i)$) when $B_{1,\sigma}[i] = 0$ (resp., $B_{1,\sigma}[i] = 1$). Those steps are repeated recursively within the corresponding bitmaps up to reaching a leaf. The symbol of that leaf is the solution to $S[i]$. Fig. 2 shows an example of access operation at position 4. Operation rank is solved analogously, and select involves a further bottom-up traversal using *select* on the bitmaps.

In addition, the wavelet trees support the following advanced operations that are useful for the *ring* (Gagie et al., 2012; Barbay et al., 2013):

- range_next_value(S, r_s, r_e, c): for $c \in \Sigma$, finds in time $O(\lg \sigma)$ the small-

est symbol $c' \geq c$ that occurs within $S[r_s \,..\, r_e]$. This is used to simulate the primitive leap() of LTJ on a compact representation of G.

- range_intersect($S_1\langle[l_1, r_1], \ldots, S_k[l_k, r_k]\rangle$): computes the intersection of the ranges $S_1[l_1 \,..\, r_1], \ldots, S_k[l_k \,..\, r_k]$, reporting the symbols that occur in all the k ranges. It is assumed that all the sequences S_i share the same alphabet. This intersection is typically faster than the one performed via leap().

- range_count($S, x_s, x_e, [r_s, r_e]$)): counts how many symbols in $S[r_s \,..\, r_e]$ belong to the range $[x_s, x_e]$ in $O(\lg \sigma)$ time. This will be used to estimate the costs of VEOs on compressed representations of G.

3.2. Indexing the data

To represent a labeled graph G, let us define the table $T_{\text{SPO}}[1 \,..\, n][1 \,..\, 3]$ storing the n graph triples sorted according to the SPO order. Column 1 of T_{SPO} corresponds to S, column 2 to P, and column 3 to O. We denote C_{O} the last column of T_{SPO}. Indeed, column C_{O} reads in left-to-right order the last level (i.e., the one corresponding to O) of the trie for SPO. Next, the process moves column C_{O} to the front in T_{SPO}, making it the first column. The table is then sorted to obtain table T_{OSP}, which conceptually represents the trie for the order OSP. Let C_{P} denote the last column of this table. Finally, column C_{P} is moved to the front of T_{OSP} and the table is sorted again, obtaining table T_{POS} and column C_{S}. See Fig. 3.

The ring index is then formed by the sequences C_*, which are stored using wavelet trees (Section 3.1), with a total space requirement of $3n \lg U + o(n \lg U)$ bits. We also build arrays A_j, for each C_j with $j \in \{\text{S}, \text{P}, \text{O}\}$, defined as $A_j[k] = |\{i \in [1 \,..\, n],\ C_j[i] < k\}|$, for $k = 1, \ldots, U + 1$. These arrays store the cumulative number of occurrences of the symbols of $\mathcal{U}$ in C_j. This adds $O(U \lg n)$ extra bits, which are $o(n \lg U)$ if $U \in o(n)$. In practice, these arrays are represented using bitvectors (Section 3.1), with a total space usage of $3(n+U)+o(n+U)$ bits. The total space is then close to the $3n \lg U$ bits needed to represent G in plain form, and it can be even less if we use compressed wavelet trees to represent the columns.

3.3. Moving between tables

We can move from a table to the next one using C_j and A_j, for $j \in \{\text{S}, \text{P}, \text{O}\}$, using the function $F_j : [1 \,..\, n] \to [1 \,..\, n]$, defined as follows:

$$F_j(i) \ := \ A_j[c] + \mathsf{rank}_c(C_j, i), \tag{2}$$

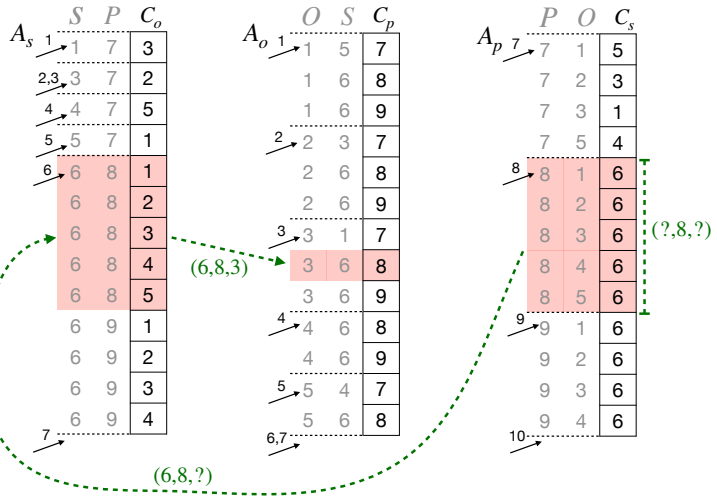

Figure 3: The *ring* representation of the graph of Fig. 1. The horizontal lines mark the values of A_{S}, A_{O}, A_{P}, left to right.

where $c = C_j[i]$. Function F_{O} maps a position in table T_{SPO}, using A_{O} and C_{O}, to the corresponding one in T_{OSP}. In Fig. 3, the straight dashed line maps from $C_{\mathrm{O}}[7] = 3$ to the position of that 3 in T_{OSP}, with $F_{\mathrm{O}}[7] = A_{\mathrm{O}}[3] + \mathsf{rank}_3(C_{\mathrm{O}}, 7) = 6 + 2 = 8$. Similarly, F_{P} maps from T_{OSP} to T_{POS}, and F_{S} maps from T_{POS} back to T_{SPO}. The function takes $O(\lg U)$ time. We can also move in the opposite direction, with the same time complexity, by computing the inverse function of F_j from Eq. (2): let c satisfy $A_j[c] < i' \leq A_j[c+1]$, then

$$F_j^{-1}(i') := \mathsf{select}_c(C_j, i' - A_j[c]). \tag{3}$$

Every node v in the trie of SPO corresponds to a range of rows in $T_{\mathrm{SPO}}[s\,..\,e]$ (i.e., a range in C_{O}): if v is the root, the range is $C_{\mathrm{O}}[s\,..\,e] = [1\,..\,n]$. If v is in the first level and corresponds to the subject $\mathrm{S} = x$, the range $C_{\mathrm{O}}[s\,..\,e]$ is that of all triples starting with x. If v is in the second level and corresponds to $(\mathrm{S}, \mathrm{P}) = (x, y)$, then $C_{\mathrm{O}}[s\,..\,e]$ corresponds to the triples starting with xy. A leaf trie node denoting the triple $(\mathrm{S}, \mathrm{P}, \mathrm{O}) = (x, y, z)$ corresponds to a single position in T_{SPO} containing xyz (i.e., a cell in C_{O}). The same holds, analogously, for tables T_{OSP} (column C_{P}) and T_{POS} (column C_{S}). The other three tries are also implicitly represented by the tables. Consider the trie for PSO. A first-level node for $\mathrm{P} = x$ corresponds to a range of rows in T_{POS} (i.e., in C_{S}), a second-level node representing $(\mathrm{P}, \mathrm{S}) = (x, y)$ corresponds to a range of rows in T_{SPO} (i.e., in C_{O}), and so on.

Along the search, each triple pattern will have a bound subset of attributes $\{\text{S}, \text{P}, \text{O}\}$, which always matches a prefix X of either SPO, OSP, or POS, the three tries we represent via columns C_{O}, C_{P}, and C_{S}. As explained, every concrete value for a prefix X corresponds to a range in some column. As we progress, the set X expands and we may have to switch from one column to another. For example, given the range $C_{\text{O}}[s\,..\,e]$ (i.e., $T_{\text{SPO}}[s\,..\,e]$) of the triples sharing a prefix X of SPO, we obtain the range $C_{\text{P}}[s'\,..\,e']$ (i.e., $T_{\text{OSP}}[s'\,..\,e']$) of the triples sharing prefix cX of OSP with

$$\begin{aligned} s' &:= A_{\text{O}}[c] + \mathsf{rank}_c(C_{\text{O}}, s-1) + 1, \\ e' &:= A_{\text{O}}[c] + \mathsf{rank}_c(C_{\text{O}}, e). \end{aligned} \tag{4}$$

This is called a *backward step*. Fig. 3 shows how we descend from the first-level node 8 in T_{POS} (represented by $C_{\text{S}}[5\,..\,9]$) to its child with value 6 (represented by $C_{\text{O}}[5\,..\,9]$), and from there to its child with value 3 (represented by $C_{\text{P}}[8\,..\,8]$). An analogous *forward step* extends X to Xc, in this case restricting the range $C_{\text{O}}[s\,..\,e]$ to a smaller range $C_{\text{O}}[s'\,..\,e']$ in the same column; see the original article (Arroyuelo et al., 2021, 2024) for details.

3.4. Constants in triple patterns

When LTJ starts, we find a range in some suitable column C_* for the constants of each triple pattern t_i. We choose a table T_* (T_{SPO}, T_{OSP}, or T_{POS}) whose attribute order is prefixed by the constant attributes in t_i, and find the range corresponding to the constant prefix X in the column that represents T_*. For example, if only the attribute O is the constant, we start from $T_{\text{SPO}}[1\,..\,n]$ and apply Eq. (4) to end with some $T_{\text{OSP}}[s\,..\,e]$; if P and S are the constants, we start from $T_{\text{OSP}}[1\,..\,n]$ and apply (the analogous of) Eq. (4) twice to end with some $T_{\text{SPO}}[s\,..\,e]$. The total initialization time is $O(\lg U)$ per triple pattern.

3.5. Supporting leaps

The remaining piece to support LTJ is function $\mathsf{leap}(t'_i, c)$ (Section 2.2.2), where t'_i is either a triple pattern t_i from Q, or one of its progressively bound versions $\mu(t_i)$. This finds the smallest child of the current node of t'_i with value $c_x \geq c$. In the context of the *ring*, this is done differently depending on whether or not the variable appears to the left of the current prefix matched. If it does, for example we are binding O and our range is $T_{\text{SPO}}[s\,..\,e]$, then we use $\mathsf{range\_next\_value}(T_{\text{SPO}}, s, e, c)$ (Section 3.1) to find the appropriate value

of c_x, and if we assign that value to O we use Eq. (4). A more difficult case arises when the desired variable is not to the left, as if binding P in $T_{\text{SPO}}[s\,..\,e]$. This only happens when we have bound just one position so far, so we start from the range $T_{\text{POS}}[A_{\text{P}}[c]+1\,..\,n]$, rework Eq. (4) for the current value of S, and finally use Eq. (3) to obtain the desired value c_x. In all cases, leap() takes $O(\lg U)$ time and the *ring* solves queries in WCO time $O(Q^* \lg U)$.

4. RDFCSA: LTJ on a compressed suffix array

We now present a new data structure, which roughly doubles the space of the *ring* in exchange for being potentially faster. The *rdfcsa* (Brisaboa et al., 2023) was designed as a compact representation for labeled graphs that can be queried by single triple patterns and binary joins. It predates the *ring* and shares with it the model of viewing the graph triples as cyclic strings of length 3 (in SPO order). This set of strings is indexed and compactly represented with a compressed suffix array (CSA, see next). The CSA on the cyclic strings suffices to solve the original *rdfcsa* queries, but in order to support the LTJ algorithm, the *rdfcsa* lacks bidirectionality, that is, unlike the *ring*, it cannot support leap() on variables to the left and to the right of the already bound positions.

We now extend the *rdfcsa* to support LTJ by storing two CSAs, one for the SPO order, and another for the OPS order, and adding them the support for leap(), in one direction. We expect this implementation to be faster than that of the *ring* (Section 3.5) because the CSA is in practice more efficient than the wavelet tree for this problem, even if both take logarithmic time.

4.1. Compressed Suffix Array

Given a string $S[1\,..\,n]$ of symbols drawn from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ (except the special symbol $S[n] = \$$, which is lexicographically smaller than all symbols in Σ), the suffix array A of S (Manber and Myers, 1993) lists all suffix indices $[1\,..\,n]$ of S in increasing lexicographic order; that is, $S[A[i]\,..\,n] < S[A[i+1]\,..\,n]$ for all $i \in [1\,..\,n-1]$. For example, let $S =$ *abracadabra*\$, then $A = \langle 12, 11, 8, 1, 4, 6, 9, 2, 5, 7, 10, 3\rangle$. All the occurrences of any given substring pattern $P[1\,..\,m]$ are pointed from a contiguous range $A[r_s\,..\,r_e]$ (because they are prefixes of the suffixes $S[A[i]\,..\,n], i \in A[r_s\,..\,r_e]$).

The compressed suffix array (CSA) (Sadakane, 2003) is a compact representation of the suffix array that replaces both S and A. It uses a permutation $\Psi[1\,..\,n]$ such that $\Psi[i] = j$ if $A[j] = A[i]+1$ (or $A[j] = 1$ if $A[i] = n$).

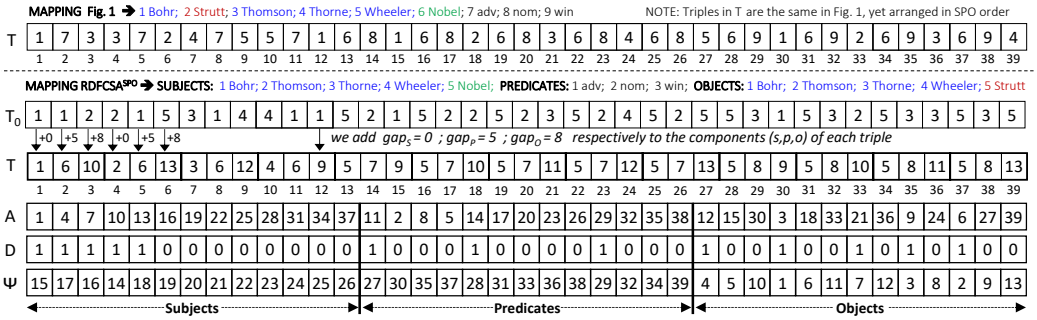

Figure 4: Structures involved in the construction of *rdfcsa*$^{\text{SPO}}$ (i.e., D and Ψ) for the graph in Fig. 1.

Therefore, given a position $p = A[i]$ in S, $j = \Psi[i]$ gives the index in A such that $A[j] = p + 1$, the next position in S. For the example above we have $\Psi[1\,..\,n] = \langle 4, 1, 7, 8, 9, 10, 11, 12, 6, 3, 2, 5\rangle$. Note that $S[A[7]\,..\,12] = bra\$$, $\Psi[7] = 3$, and so $S[A[3]\,..\,12] = ra\$$. The CSA also includes a bitvector $D[1\,..\,n]$ that sets $D[i] = \mathsf{1}$ to mark the positions i in A where the first symbol of the suffix pointed to from $A[i]$ changes, that is, $D[i] = \mathsf{1}$ iff $i = 1$ or $S[A[i-1]] < S[A[i]]$. In our example, $D = \langle \mathsf{1}, \mathsf{1}, \mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{0}, \mathsf{1}, \mathsf{0}, \mathsf{1}, \mathsf{1}, \mathsf{1}, \mathsf{0}\rangle$. The symbol $c = S[x]$, pointed from $A[i] = x$, can be obtained as $c = \mathsf{rank}_{\mathsf{1}}(D, i)$. Further, $S[x{+}1] = \mathsf{rank}_{\mathsf{1}}(D, \Psi[i])$, $S[x{+}2] = \mathsf{rank}_{\mathsf{1}}(D, \Psi[\Psi[i]])$, and in general $S[x + k] = \mathsf{rank}_{\mathsf{1}}(D, \Psi^k[i])$.

Regarding space, Ψ is composed of at most σ increasing sequences, which can be compressed by encoding differences and applying run-length encoding for runs of $+1$ differences. The required space is $nH_k(S) + O(n \lg\lg \sigma)$ bits for any $k \leq \alpha \lg_\sigma n$ and constant $\alpha < 1$ (Navarro and Mäkinen, 2007), where $H_k(S) \leq \lg \sigma$ is the k-th order entropy of S. Bitvector D adds $n + o(n)$ bits.

4.2. Indexing the data

The *rdfcsa* requires a particular mapping from $\mathrm{dom}(G)$ to integers. Different alphabets $[1\,..\,n_{\mathrm{S}}]$, $[1\,..\,n_{\mathrm{P}}]$, and $[1\,..\,n_{\mathrm{O}}]$ must be considered, respectively, for subjects, predicates, and objects. From them, the first n_{SO} symbols in the alphabets of subjects and objects are constants that could occur both as subjects and objects in a triple. It then holds $|U| = n_{\mathrm{P}} + n_{\mathrm{S}} + n_{\mathrm{O}} - n_{\mathrm{SO}}$.

Considering a sequence of n triples sorted in SPO order, the *rdfcsa* creates a unique sequence of integers $T[1\,..\,3n]$ where, for each triple $(s, p, o) \in G$, the string $\langle s', p', o'\rangle = \langle s + gap_{\mathrm{S}}, p + gap_{\mathrm{P}}, o + gap_{\mathrm{O}}\rangle$ is appended to T. The offsets

$(gap_{\text{S}}, gap_{\text{P}}, gap_{\text{O}}) = (0, n_{\text{S}}, n_{\text{S}} + n_{\text{P}})$ enforce disjoint identifiers for subjects, predicates, and objects, and ensure $s' < p' < o'$. Then, a CSA is built on T. Because of the offsets, there are three regions in the suffix array A (and D), $A[1\,..\,n]$, $A[n{+}1\,..\,2n]$, $A[2n{+}1\,..\,3n]$, with entries pointing respectively to the subjects, predicates, and objects in T. Consequently, $\Psi[1\,..\,n]$ contains only values within $[n{+}1, 2n]$, whereas the values in $\Psi[n{+}1\,..\,2n]$ and $\Psi[2n{+}1\,..\,3n]$ are within $[2n+1, 3n]$ and $[1, n]$, respectively. Finally, Ψ is modified to make it cycle on the triples, that is, we enforce $\Psi[\Psi[\Psi[i]]] = i$. This is easily done by decrementing the values in $\Psi[2n+1\,..\,3n]$, except that $\Psi[i] = 1$ is converted to $\Psi[i] = n$. To reduce space, Ψ is represented as the sequence $\Psi[i] - \Psi[i-1]$, using Huffman and run-length encoding on those gaps. Access in time $O(t_\Psi)$ to any $\Psi[i]$ value is supported by sampling values $\Psi[1 + k \cdot t_\Psi], k \geq 0$, which requires $O((n/t_\Psi) \lg n)$ additional bits on top of the compressed sequence (we will asume $t_\Psi \in O(1)$). Bitvector D takes $3n + o(n)$ further bits. Fig. 4 shows an example.

Analogously, we create a second *rdfcsa* considering triples sorted in OPS order, and with offsets $(gap_{\text{S}}, gap_{\text{P}}, gap_{\text{O}}) = (n_{\text{O}} + n_{\text{P}}, n_{\text{O}}, 0)$. We refer to our two *rdfcsa* structures as *rdfcsa*$^{\text{SPO}}$ and *rdfcsa*$^{\text{OPS}}$. In either of them, the triple content pointed at position i is retrieved in $O(1)$ time by extracting $\mathsf{rank}_1(D, i)$, $\mathsf{rank}_1(D, \Psi[i])$, and $\mathsf{rank}_1(D, \Psi[\Psi[i]])$, permuting them to order SPO, and subtracting the corresponding *gap* values. We now show how we carry out the critical processes of LTJ with these structures.

4.3. Constants in triple patterns

We use the text searching capabilities of the *rdfcsa* to find a suffix array interval corresponding to all the triples that match the constants of a given triple pattern. The subsequent variable intersection process then starts from those intervals, which correspond to trie nodes in LTJ, as with the *ring*. Recall that, before finding any constant in the *rdfcsa*, it must be mapped by adding the corresponding *gap*. We use two operations:

- $[l, r] := \mathsf{range}(c)$. For a given constant c we obtain the suffix array range $A[l, r]$ of the (cyclic) triples starting with c, with $l \leftarrow \mathsf{select}_1(D, c)$ and $r \leftarrow \mathsf{select}_1(D, c + 1) - 1$. This takes constant time. Note that, since c can be a subject, a predicate, or an object, and those identifiers have disjoint suffix array areas in the *rdfcsa*, this operation lets us select all the triples with a given subject, a given predicate, or a given object.

- $[l, r] := \mathsf{down}(l_c, r_c, d)$. Given a suffix array range $[l_c, r_c] \subseteq \mathsf{range}(c)$, so the triples in $A[l_c, r_c]$ start with constant c, this operation finds the subrange $[l, r] \subseteq [l_c, r_c]$ of those triples where the c is followed by constant d. This is equivalent to stating that $\forall i \in [l, r], \Psi[i] \in \mathsf{range}(d)$. Since Ψ is increasing inside $\mathsf{range}(c)$, we can binary search for the first (last) position l (r) in $[l_c, r_c]$ such that $\Psi[l]$ ($\Psi[r]$) falls into $\mathsf{range}(d)$, in $O(\lg n)$ time.

If a triple pattern t has no constants, its range is $[1 \,..\, 3n]$ in both *rdfcsa*$^{\text{SPO}}$ and *rdfcsa*$^{\text{OPS}}$. If it has a single constant c, then its range in both is $\mathsf{range}(c)$ (the mapping of c using *gap* and the resulting range differs in both *rdfcsa*s). Which *rdfcsa* will be used depends on whether the next variable to eliminate is to the left or to the right of c. Therefore, some triple patterns will have a range in *rdfcsa*$^{\text{SPO}}$ and others in *rdfcsa*$^{\text{OPS}}$.

If t has two constants, we can use either *rdfcsa* because the next variable to eliminate will be both to the left and to the right of the bound positions. Say we choose *rdfcsa*$^{\text{SPO}}$. Let cd be the two consecutive constants in SPO order (i.e., SP $= cd$, PO $= cd$, or OS $= cd$). We thus compute $[l_c, r_c] := \mathsf{range}(c)$ and then the desired range is $[l_d, r_d] := \mathsf{down}(l_c, r_c, d)$.

4.4. Supporting leaps

To support the operation $\mathsf{leap}()$ we define new primitives:

- $l' := \mathsf{findTarget}_{\Psi}(l, r, t_l, t_r)$. Given a range $[l \,..\, r]$ where Ψ is increasing, it returns the smallest $l' \in [l \,..\, r]$ such that $\Psi[l'] \in [t_l \,..\, t_r]$, and 0 if there is none. It proceeds as for down, yet it needs only one binary search in $[l, r]$.

- $l' := \mathsf{findTarget}_{\Psi\Psi}(l, r, t_l, t_r)$. Given a range $[l \,..\, r]$ where Ψ is increasing, it returns the smallest $l' \in [l \,..\, r]$ such that $\Psi[\Psi[l']] \in [t_l \,..\, t_r]$, or zero if there is none.

- $L := \mathsf{limitV}(v)$. It returns the highest offset in D for any constant of the same type as variable v. For example, if v is a subject this is n in *rdfcsa*$^{\text{SPO}}$ and $3n$ in *rdfcsa*$^{\text{OPS}}$.

Recall that $\mathsf{leap}(t'_i, c)$ returns the first constant $c_x \geq c$ where a given variable x has occurrences in t'_i, where t'_i is either a triple pattern t_i from Q, or one of its progressively bound versions $\mu(t_i)$. The way we solve $\mathsf{leap}()$ depends on where the constant(s) and the variable x appear in t'_i.

If there are no constants in t'_i, the answer is simply $c_x := c$, because our mapping makes all the symbols appear in T.

If there is only one constant d in t'_i, then we have a range $[l, r]$ for t'_i in both *rdfcsa*$^{\text{SPO}}$ and *rdfcsa*$^{\text{OPS}}$. If x follows d in SPO order, we use *rdfcsa*$^{\text{SPO}}$, otherwise we use *rdfcsa*$^{\text{OPS}}$. We first compute $l' := \mathsf{findTarget}_{\Psi}(l, r, l_c, \mathsf{limitV}(x))$ where $l_c = \mathsf{select}_1(D, c)$. Then, if $l' \neq 0$ we return $c_x := \mathsf{rank}_1(D, \Psi[l'])$, otherwise we return $c_x := \perp$.

Finally, if t'_i contains two constants d and d', they have a range $[l, r]$ for dd' or $d'd$ in either *rdfcsa*$^{\text{SPO}}$ or *rdfcsa*$^{\text{OPS}}$, so we complete the search in the corresponding structure. In this case, we first compute $l_c = \mathsf{select}_1(D, c)$ and then $l' := \mathsf{findTarget}_{\Psi\Psi}(l, r, l_c, \mathsf{limitV}(x))$. Then, if $l' \neq 0$ we return $c_x := \mathsf{rank}_1(D, \Psi[\Psi[l']])$, and $c_x := \perp$ otherwise.

As an example, recall that if we have a triple pattern (x, y, z) with no constants, the initial range in *rdfcsa* is $\mathcal{R} := [1, 3n]$. If we bind $y :=$ win (i.e., the 3rd predicate, which is mapped to id $\mathbf{8} = 3 + gap_{\text{P}}$), we update $\mathcal{R} := \mathsf{range}(\mathbf{8}) = [23, 25]$ in *rdfcsa*$^{\text{SPO}}$ (and also in *rdfcsa*$^{\text{OPS}}$). Now, since Wheeler is the 4th object and maps into id $\mathbf{12} = gap_{\text{O}} + 4$, if we call $\mathsf{leap}((x, \mathbf{8}, z), \mathbf{12})$, since Wheeler follows win in SPO order, we must use *rdfcsa*$^{\text{SPO}}$. We first compute $l' := \mathsf{findTarget}_{\Psi}(23, 26, \mathsf{select}_1(D, \mathbf{12}), 39) = 26$, and then solve $\mathsf{leap}((x, \mathbf{8}, z), \mathbf{12}) = \mathsf{rank}_1(D, \Psi[26]) = \mathsf{rank}_1(D, 39) = \mathbf{13}$. Therefore, leap() returns object $5 = \mathbf{13} - gap_{\text{O}}$ which corresponds to Strutt, that is, the first object after Wheeler reached from the current range $\mathcal{R}$.

5. URing: A Unidirectional Ring

Bidirectionality is the key to using just one *ring* to index the $3! = 6$ orders required by LTJ. The *rdfcsa*, instead, requires two copies of the index, thereby roughly doubling the space. We now explore the fact that the wavelet tree representation of the ring columns C_* supports an intersection algorithm (range_intersect, Section 3.1) that is likely faster than the one implemented in LTJ, which is based on the primitive leap().

When we eliminate a new variable x, every triple pattern t_i where it appears is represented by a range in some column, $C_*[l_i \,..\, r_i]$. Assume x appears to the left of the positions already bound. The desired constants c_x for x are the values that appear in all those ranges $C_*[l_i, .. \, r_i]$. We find them by running range_intersect on all those ranges in order to obtain, one by one, the desired values c_x (the algorithm runs even if the ranges are in different

sequences C_*). We then add each such binding $(x := c_x)$ to the mapping μ and recurse on that branch.

The problem with using that intersection algorithm is that it works only if the variable to eliminate is to the left of the current ranges, and therefore, analogously to the *rdfcsa*, we have a unidirectional index. Just as for *rdfcsa*, we must then have two indices, $ring^{\text{SPO}}$ and $ring^{\text{OPS}}$ to ensure that we always have a range that can be extended to the left. The algorithm proceeds exactly as the *rdfcsa* over those two copies, except that the intersection algorithm of LTJ is replaced by the custom algorithm range_intersect. An additional benefit is that going rightwards in the binding is somewhat more expensive on the *ring* than going leftwards, and this new variant goes always leftwards.

6. Improved Variable Elimination Orders

Our second contribution is the study of improved VEOs in the context of compact indices for LTJ, which deviate from the VEO defined in Section 2.3. The first improvement is the use of *adaptive* VEOs; the second is on how to efficiently compute (or approximate) w_{ij} in our compact index representations.

6.1. Adaptive VEOs

In previous work using the VEO described in Section 2.3, the VEO is fixed before running LTJ. The selectivity of each variable x_j is estimated beforehand, by assuming it will be the *first* variable to eliminate. In this case, Eq. (1) takes the minimum of the number of children in all the trie nodes we must intersect, as an estimation of the size of the resulting intersection. The estimation is much looser on the variables that will be eliminated later, because the children to intersect can differ a lot for each value of x_j.

We then consider an *adaptive* version of the heuristic: we use the described technique to determine only the *first* variable to eliminate. Say we choose x_j. Then, for each distinct binding $x_j := c$, the corresponding branch of LTJ will run the VEO algorithm again in order to determine the second variable to eliminate, now considering that x_j has been replaced by c in all the triples t_i where it appears. This should produce a much more accurate estimation of the intersection sizes.

In the adaptive setting, we do not check anymore that the new variable shares a triple with a previously eliminated one; this aimed to capture the fact that those triples would be more selective when some of their positions

were bound, but now we know exactly the size of those progressively bound triples. The lonely variables are still processed at the end.

Observe that, in the adaptive case, there is not anymore a single VEO; each different branch can have one. While this technique does not require any extra space, it could incur in an extra cost to repeatedly recompute the VEO (in fact, only its first variable) for each binding. The cost to compute Eq. (1), on each of our compact index representations, becomes then of paramount importance for adaptive VEOs.

6.2. Computing the VEO predictors

The *ring* cannot efficiently compute w_{ij} as described in Section 2.3, because it does not know the number of children of the node v: the *ring* only has the range $C_*[l\,..\,r]$ corresponding to v, which results from resolving the constants in t_i; recall Section 3.4. The size $r - l + 1$ of the range was then used as a reasonable estimation of w_{ij} (Arroyuelo et al., 2021, 2024). In the case t_i has one constant and x_j is to its right, we can also use $r - l + 1$ to estimate w_{ij}, because the range size would be the same in both directions. We do the same when implementing the *rdfcsa*.

While the use of $r-l+1$ as a predictor of the true weight w_{ij} can be seen as an approximation, it can be argued to be a *better* predictor of the difficulty of binding x_j in t_i. Note that $r - l + 1$ is the number of *leaf descendants* of the current node v of the trie τ_i, whose children are the bindings of x_j in t_i. The number of descendants may be a more accurate estimation of the *total* work that is ahead if we bind x_j in t_i, as opposed to the children, which yield the number of distinct values x_j will take without looking further.

By using (additional) wavelet trees, we can also compute w_{ij} as the number of children of v on the *ring*, in $O(\lg n)$ time (Gagie et al., 2013). What we need is to count the number of *different* symbols in the range $C_*[l\,..\,r]$. Let M be such that $M[i]$ is the largest value $i' < i$ such that $C_*[i] = C_*[i']$, or 0 if there is no such i'. This implies that $C_*[q]$ is the *first* occurrence of a symbol in $C_*[l\,..\,r]$ iff $l \leq q \leq r$ and $M[q] < l$. We can then count the number of first occurrences of symbols in $C_*[l\,..\,r]$ by counting the number of values less than l in $M[l\,..\,r]$. This is accomplished by the range_count function (Section 3.1) if we have M represented as a wavelet tree.

Note that the use of M requires that the new variable x_j is to the left of the constants in t_i; therefore we need one sequence M per column C_* in both the (actual) *ring* for the order SPO and the (virtual) *ring* for the order OPS.

So, even if we use just one bidirectional *ring*, we must add two sequences M (thus tripling the space).

6.3. Refining VEO predictors

The value w_j obtained by using the VEO predictor where the estimator is the size of the range $C_*[l\,..\,r]$ corresponds to the maximum number of triples that can participate in the intersection. Therefore, it is an upper bound to the size of the intersection of the set of triples.

We can obtain a better approximation of the intersection size by splitting the values of $C_*[l\,..\,r]$ into disjoint subsets of the alphabet and applying Eq. (1) to each. The sum of the weights of the subsets is a more refined approximation to the intersection. We can then refine the heursitic of Eq. (1) as

$$w_j = \sum_{\gamma \subset [1,\sigma]} \min\{w_{ij}^{\gamma} \mid x_j \text{ appears in triple } t_i\}, \tag{5}$$

where w_{ij}^{γ} is the weight of x_j in t_i for the partition γ of the alphabet. The partitions are disjoint and their union is $[1, \sigma]$.

The *ring* can exploit the wavelet trees to easily compute w_{ij}^{γ}. Let us consider a variable x_j that appears in a triple pattern t_i, with the possible values of x_j in the range $C_*[l\,..\,r]$. Our estimation algorithm starts in the range $B_{1,\sigma}[l\,..\,r]$ of the root. By mapping $[l\,..\,r]$ to its left child ($B_{1,m}$), where m is $\lfloor(\sigma+1)\rfloor/2$, we retrieve the range $B_{1,m}[l'\,..\,r']$ of the symbols from $C_*[l\,..\,r]$ that belong to the first half of the alphabet $[1, m]$. That range is computed in $O(1)$ time as $[rank_0(B_{1,\sigma}, l-1)+1\,..\,rank_0(B_{1,\sigma}, r)]$. The length of this range is equivalent to $w_{ij}^{[1,m]}$. By using $rank_1$ instead of $rank_0$ we compute the range of $[l\,..\,r]$ in the right child ($B_{m+1,\sigma}$), whose length is $w_{ij}^{[m+1,\sigma]}$. Since $w_{ij}^{[1,m]} + w_{ij}^{[m+1,\sigma]} = w_{ij}$ for every i, j, $(\min_i w_{ij}^{[1,m]}) + (\min_i w_{ij}^{[m+1,\sigma]}) \leq \min_i w_{ij}$ is a tighter upper bound on the size of the intersection.

In that procedure we obtain two partitions, $[1, m]$ and $[m+1, \sigma]$. If we continue mapping the current ranges in $B_{1,m}$ and $B_{m+1,\sigma}$ to their children, each previous partition splits into two other halves. Therefore, by repeating those steps k levels, we get up to 2^k partitions. The length of each range in a node of the k-th level matches a weight w_{ij}^{γ}. As k increases, the approximation of the heuristic improves (indeed, if we reach $k = \lg \sigma$ we obtain the actual size of the intersection, at cost $O(\sigma)$).

For example, consider two triples, t_i and $t_{i'}$, whose defined ranges are $[1\,..\,4]$ and $[5\,..\,8]$, respectively, in the sequence of Fig. 2. Note that their

k	γ	w_{ij}^{γ}	$w_{i'j}^{\gamma}$	min	w_j
0	$[1,6]$	4	4	4	4
1	$[1,3]$	2	0	0	2
	$[4,6]$	2	4	2	
2	$[1,2]$	1	0	0	0
	$[3,3]$	1	0	0	
	$[4,5]$	2	0	0	
	$[6,6]$	0	4	0	

Table 1: The partitions γ and weights obtained in the sequence of Fig. 2 with the ranges $[1,4]$ and $[5,8]$ of two triple patterns t_i and $t_{i'}$, respectively. Column k indicates the number of levels to traverse in the wavelet tree.

intersection is empty. Since both ranges have length 4, the weight w_j per Eq. (1) is 4. In order to apply Eq. (5), the algorithm starts at the root $B_{1,6}$ and maps each range to its children. For example, the range $B_{1,6}[1\,..\,4]$ maps to $B_{1,3}[1\,..\,2]$ and $B_{4,6}[1\,..\,2]$, thus $w_{ij}^{[1,3]} = 2$ and $w_{ij}^{[4,6]} = 2$. In the same way, from $B_{1,6}[5\,..\,8]$ we just reach $B_{4,6}[3\,..\,6]$, so $w_{i'j}^{[1,3]} = 0$ and $w_{i'j}^{[4,6]} = 4$. Consequently, $w_j = \min(w_{ij}^{[1,3]}, w_{i'j}^{[1,3]}) + \min(w_{ij}^{[4,6]}, w_{i'j}^{[4,6]}) = 2$, which is a closer bound to the actual intersection size. By descending one more level from the ranges in $B_{1,3}$ and $B_{4,6}$, we will obtain four partitions and w_j reaches 0. The partitions and weights obtained for each level are shown in Table 1.

Note that we are assuming that the range in the *ring* contains the values of the variable to bind x_j. However, it is possible that in t_i the defined range does not correspond to that variable. Those cases can occur when only one position of t_i is bound. For instance, if the predicate of t_i is bound to a constant c, the *ring* defines the range $C_{\text{S}}[l\,..\,r]$ (T_{POS}). When the variable x_j is the object of t_i, there is no direct access to the values of x_j.

In that case, as in the main algorithm, we compute the size of each partition γ in $C_{\text{O}}[1\,..\,n]$ (T_{SPO}), without any interval restriction. With this approach, w_{ij}^{γ} is the number of triples whose object belongs to γ. From those, we just need to count the triples where its predicate has constant c. Since those partitions correspond to non-overlapping consecutive intervals in T_{OSP}, the number of triples whose predicate is c in each alphabet partition γ is obtained as $rank_c(C_{\text{P}}, e) - rank_c(C_{\text{P}}, s) + 1$, where $[s, e]$ is the range of γ in T_{OSP}. This range is computed as $s = A_{\text{O}}[c_s] + 1$ and $e = A_{\text{O}}[c_e + 1]$, c_s and c_e being the first and last symbol, respectively, of each alphabet partition γ.

7. Experimental results

We compare the compact indexing schemes described along the paper and various state-of-the-art alternatives, in terms of space usage and time for evaluating various types of BGPs.

Our experiments ran on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores, 15 MB cache, and 378 GB RAM.

7.1. Datasets and queries

We run our benchmarks over the Wikidata graph (Vrandecic and Krötzsch, 2014), which we choose for its scale, diversity, prominence, data model (i.e., labeled edges) and real-world query logs (Malyshev et al., 2018; Bonifati et al., 2019). This is the largest public graph we are aware of. The graph features $n = 958{,}844{,}164$ triples, which take 10.7 GB if stored in plain form using 32 bits for the identifiers.

An important advantage of using Wikidata is that we have access to real queries posed by users, with a mix of actual information needs (Malyshev et al., 2018). In search of challenging examples, we downloaded queries that gave timeouts, and selected queries with a single BGP, obtaining 1,295 unique queries. Those are classified into three categories: (I) 520 BGPs formed by a single triple pattern, which mostly measure the retrieval performance of the index; (II) 580 BGPs with more than one triple but only one variable appearing in more than one triple, which measure the performance of joins but do not distinguish good from bad VEOs (as long as the join variable is eliminated first, of course); (III) 195 complex BGPs, where the performance of different VEOs can be compared.

All queries are run with a timeout of 10 minutes and a limit of 1000 results (as originally proposed for WGPB (Hogan et al., 2019)). This measures the time the systems need to display a reasonable number of results. We also compare the systems without the limit of results, which measures throughput in cases where we need all the results. The space of the indices is measured in bytes per triple (bpt); a plain 32-bit storage requires 12 bpt.

7.2. Systems compared

Our experiments compare all indexing schemes described:

- Two *ring* variants (Section 3), `Ring`-large and `Ring`-small, corresponding to using plain or compressed bitvectors in the wavelet trees, respectively (these are called Ring and C-Ring, respectively, in the original paper (Arroyuelo et al., 2021, 2024)).

- Their corresponding unidirectional versions, which compute intersections using the wavelet tree (Section 5): `URing`-large and `URing`-small.
- Their extension to compute the standard VEO based on number of children (Section 6.2): `VRing`-large, `VRing`-small, `VURing`-large and `VURing`-small.
- Their extension to compute the refined estimators for the VEO (Section 6.3), both in their bidirectional and unidirectional variantes: `IRing`-large, `IRing`-small, `IURing`-large, and `IURing`-small. The number of levels that descend those estimators is configured to 3.
- Two *rdfcsa* variants (Section 4): `RDFCSA`-large represents Ψ in plain form; `RDFCSA`-small sets $t_\Psi = 16$ and uses Huffman and run-length encoding to compress Ψ.
- All the versions above compute the VEO in traditional ("global VEO") and in adaptive form (Section 6.1).

We also compare various prominent graph database systems:

- MillenniumDB (Vrgoc et al., 2023): A recently developed open-source graph database. We use here a specialized version that stores six tries in the form of B+-trees and supports full LTJ, with a sophisticated (yet global) VEO. We run MillenniumDB over a RAM disk to avoid using external memory.
- Jena LTJ (Hogan et al., 2019): An implementation of LTJ on top of Apache Jena TDB. All six different orders on triples are indexed in B+-trees, so the search algorithm is always WCO.
- RDF-3X (Neumann and Weikum, 2010): Indexes a single table of triples in a compressed clustered B+-tree. The triples are sorted and those in each tree leaf are differentially encoded. RDF-3X handles triple patterns by scanning ranges of triples and features a query optimizer using pair-wise joins.
- Virtuoso (Erling, 2012): The graph database hosting the public DBpedia endpoint, among others. It provides a column-wise index of quads with an additional graph (g) attribute, with two full orders (PSOG, POSG) and three partial indices (SO, OP, GS) optimized for patterns with constant predicates. It supports nested loop joins and hash joins.

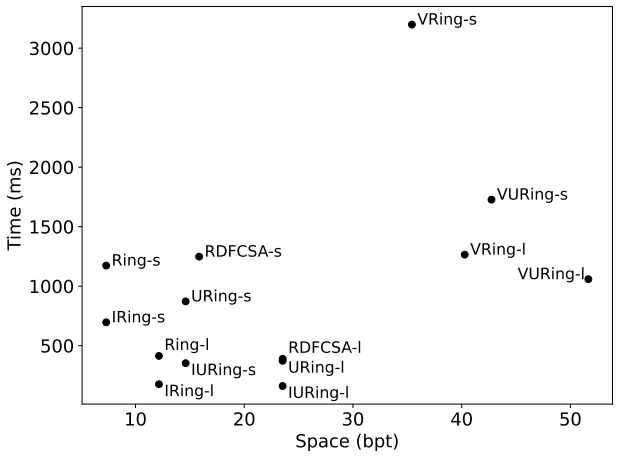

Figure 5: Index space and the averaged query times of the adaptive variants in msec, limiting outputs to 1000 results. Suffixes *s* and *l* mean *small* and *large*, respectively.

- Blazegraph (Thompson et al., 2014): The graph database system hosting the official Wikidata Query Service (Malyshev et al., 2018). We run the system in triples mode, with B+-trees indexing orders SPO, POS, and OSP. It supports nested-loop joins and hash joins.

The code was compiled with g++ with flags -std=c++11 and -O3; some alternatives have extra flags to enable third party libraries. Systems are configured per vendor recommendations.

7.3. Results

Table 2 shows the index space in bytes-per-triple (bpt) and some general statistics on the query times obtained on our benchmark, when we limit the results to 1000. We list all the *ring* and *rdfcsa* variants first, and then other systems. Fig. 5 illustrates the tradeoff between index space and average query time for our adaptive variants.

7.3.1. The general picture

It is immediately evident that adaptiveness and the use of the refined estimator is always a good strategy, especially in terms of robustness: average times and timeouts are considerably reduced in all cases. Adaptiveness speeds up all `Ring` variants by a factor of 2–11, whereas the refined estimation further speeds up the adaptive variants by a factor of 1.7–2.5. The

System	Space (bpt)	Average Gl	Average Ad	Median Gl	Median Ad	Timeouts Gl	Timeouts Ad
`Ring`-small	7.30	3056	1173	24	24	5	0
`IRing`**-small**	**7.30**	2568	**696**	27	**27**	4	**0**
`Ring`-large	12.15	2256	414	8	8	3	0
`IRing`**-large**	**12.15**	2021	**176**	8	**8**	3	**0**
`URing`-small	14.61	2779	872	20	20	4	1
`IURing`-small	14.61	2300	353	24	20	3	0
`URing`-large	23.53	1481	373	8	8	0	0
`IURing`-large	23.53	1319	161	8	8	0	0
`VRing`-small	35.42	4594	3198	24	24	4	3
`VRing`-large	40.28	3067	1265	8	8	5	1
`VURing`-small	42.74	3467	1727	20	20	1	2
`VURing`-large	51.65	2124	1059	8	8	0	1
`RDFCSA`-small	15.85	2323	1248	8	8	1	1
`RDFCSA`**-large**	**23.54**	579	**390**	2	**2**	0	**0**
MillenniumDB	**156.78**	**96**		**27**		**0**	
Jena LTJ	168.84	1930		162		1	
Virtuoso	60.07	4880		50		8	
RDF-3X	85.73	8230		126		13	
Blazegraph	90.79	9220		54		14	

Table 2: Space and query times (in msec) of all the systems, limiting results to 1000, with Gl(obal) and Ad(aptive) VEOs. Timeouts count queries exceeding 10 min.

dominating strategies, each with its own space-time niche and all using the adaptive variant, are:

- The tiny variant: `IRing`-small uses just 7.30 bpt (nearly half of a plain storage of the triples), and solves queries with a median of 27 msec and an average of 0.70 sec.

- The small variant: `IRing`-large uses nearly the space of the triples in plain form (12.15 bpt) and solves queries with a median of 8 msec and an average of 0.18 sec.

- The medium variant: `RDFCSA`-large roughly doubles that space (23.54 bpt) and reduces the median to nearly 2 msec. It also offers much better average times on simple queries and with unlimited number of results, as seen later. `IURing`-large, using the same space as `RDFCSA`-large, has a worse median but a better average time, 0.16 sec. It is not

System	Space (bpt)	Type I Avg	Type I Med	Type II Avg	Type II Med	Type III Avg	Type III Med
`IRing`-small	7.30	12	8.0	380	36	3455	97
`IRing`-large	12.15	3.9	2.9	93	11	881	32
`IURing`-large	23.53	4.6	4.0	75	12	832	36
`RDFCSA`-large	23.54	0.6	0.3	18	2.9	2611	14

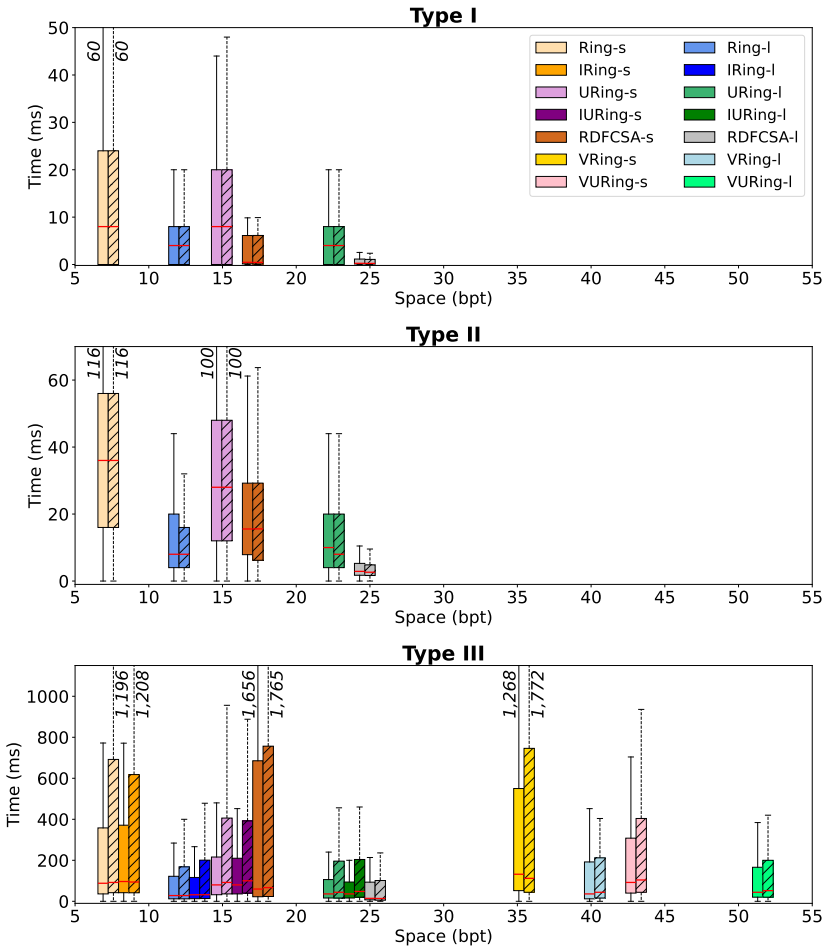

Figure 6: Time distribution, in milliseconds, per query type, limiting outputs to 1000 results. The figures show boxplots for the smaller variants, marking the median inside. Horizontal positions are slightly shifted for visibility; see Table 2 for the exact space values. Hatched/empty boxes refer to global/adaptive VEOs. The numbers on top indicate where the capped whiskers reach.

so interesting, however, in comparison with IRing-large, which uses half the space and is only marginally slower on average.

URing *variants.* The one-directional ring, URing, improves the times of Ring by 10%–50% depending on the variant, in exchange for doubling its space. The largest improvements turn out not to be so relevant, however, because they correspond to the small URing version, which loses by 20%–50% to the large corresponding Ring version, while using about the same space. The large URing versions, on the other hand, are only 10% faster than their corresponding Ring version, while doubling their space. Using similar space as URing, RDFCSA-large turns out to be more interesting in that it offers more stable times, with a median of 2 instead of 8. RDFCSA-small, instead, is not competitive.

VRing *variants.* It is also apparent that computing w_{ij} as the number of leaf descendants for choosing VEOs using Eq. (1) performs much better than the original formula (Hogan et al., 2019) that uses the number of children of the node: the VRing variants are much larger and slower than their corresponding Ring counterparts. In the case of adaptive VRing, where the VEO is recomputed for every binding, this is worsened by the fact that computing w_{ij} on the VRing takes $O(\lg n)$ time, as opposed to $O(1)$ on the Ring.

Classical systems. The best performing classical systems are generally WCO: MillenniumDB and (way behind) Jena LTJ, which use about 13 times more space than our "small" variant (IRing-large). MillenniumDB is almost twice as fast as IRing-large on the average, but has a higher median. This suggests it incurs a base cost of a few tens of milliseconds for every query, even the easy ones. No other classical index (including the non-WCO ones) is competitive with our compressed variants.

7.3.2. Query types and space-time tradeoffs

Fig. 6 shows the time distributions on each query type. As expected, adaptive query plans and refined estimation of the intersections make no difference with respect to global ones in queries of type I and II, except for a few small gaps. Instead, they make a big difference in queries of type III, more in terms of robustness (lower average, less dispersion) than in the medians. The refined estimation of intersections has, as we have seen, a large impact on averages, but is not noticeable in terms of distribution. This shows that the refinement is mostly useful to avoid few, but very high, query times that are produced when using the coarser measure.

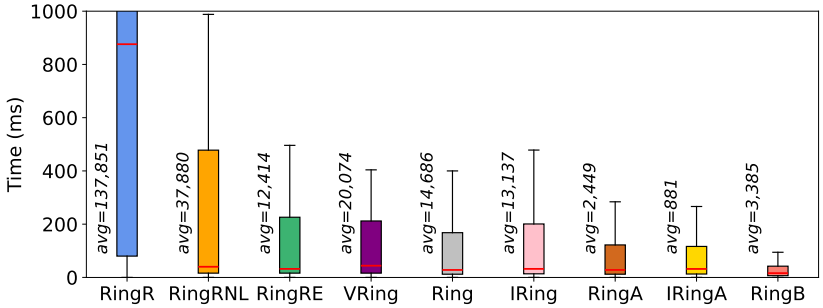

Figure 7: Comparing the (global) VEOs of `VRing`, `Ring`, and `IRing` on queries of type III limited to 1000 results, with variants that choose the VEO at random or optimally.

Using twice the space of `IRing`-large, `RDFCSA`-large performs better, most clearly on the simpler query types, where the time to extract the output tuples dominates. On those queries `RDFCSA`-large exploits its faster access to the data. Its smaller version, `RDFCSA`-small, instead, is dominated by `IRing`-large on queries of type II and III (but not on type I).

The table on top of the figure shows only the best performing variants, which are all adaptive. It clearly shows how times decrease steadily as we use more space. The exception is that, on type-III queries, `IRing`-large fares better than `RDFCSA`-large. In this type of queries the query resolution strategy is more important than the mere time to access the data.

7.3.3. Variable elimination orders

As already noted, the `VRing` variants increase the space while worsening query times. This shows up especially on the average times of Table 2, where the `VRing` adaptive variants are 2–3 times slower than their corresponding `Ring` variants. In part this is due to the $O(\lg n)$ time invested in computing the number of children, what can be confirmed by the fact that, on global VEOs where this computation is done only a few times, the `VRing` variants are "only" 20%–50% slower. But still, with global VEOs, where this $O(\lg n)$ burden is insignificant, the results show that the `Ring` variants generate better VEOs than their `VRing` counterparts, apart from computing them faster. The fact that the medians and boxplot distributions are much closer than the averages show that, while most generated VEOs are similar, `Ring` avoids very bad VEOs that `VRing` sometimes generates.

This leads to the question of how good is the original strategy described in Section 2.3. To answer it, we created `Ring` variants that choose the VEO at

random, in nonadaptive form for simplicity, and compare them in Fig. 7. The variant `Ring`R, which uses a completely random order, is not competitive at all. `Ring`RNL, which leaves the lonely variables to the end, does much better, showing the convenience of this strategy. The results improve even more on `Ring`RE, which in addition avoids eliminating variables that are disconnected from previously eliminated ones, if possible. Note that this is just like `Ring`, yet using random values to estimate the weights w_{ij}. `VRing` distributes only slightly better than such a random estimation (and, actually, worsens a lot on the average), but `Ring`, which uses the number of leaf descendants to compute w_{ij}, performs noticeably better. `IRing`, which refines the estimation of the intersections, performs similarly to `Ring` when using a global VEO. This changes on the adaptive versions: `Ring`A sharply outperforms `Ring`, and `IRing`A further outperforms `Ring`A.

An important question here is how much more margin for improvement do we have by choosing VEOs. To partially answer it, we executed `Ring` with *all the possible global VEOs*, and chose the best time for each query to create an ideal variant called `Ring`B. As the number of orders to try is the factorial of the number of variables, we reduced the search space by considering only the non-lonely variables and forcing the others to be connected with some eliminated variable in some triple if possible, as `Ring` does; we believe the best time should always be within that search space. Further, we did not optimize those queries with 7 or more variables; we just used the time obtained by `Ring` on those. The times of `Ring`B are then an *upper bound* to the best times `Ring` could possibly obtain by choosing a good VEO (note that we also leave out adaptive orders). Even so, Fig. 7 shows that `Ring`B sharply outperforms `IRing`, our best global VEO, by a factor of about 4 on the average and 2 in the median. `Ring`B actually outperforms our best adaptive VEO, `IRing`A, by a factor of about 2 in the median, but interestingly, `IRing`A is almost 4 times faster on the average. This shows that, in many cases, the adaptive VEOs outperform *the best possible* global VEO. On the other hand, the experiment shows that it is still possible to improve a lot upon our current VEOs.

7.3.4. Not limiting the number of results

The case without limits in the number of answers is shown in Table 3, where for succinctness we left only the best performing of the classical indices. Fig. 8 shows space-time tradeoffs with respect to the average adaptive times.

Although the times are much higher and thus the scale measures seconds, the dominant variants are the same as before. An important difference,

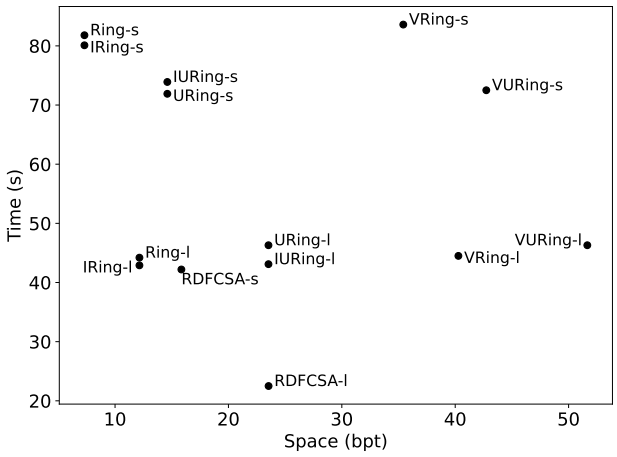

Figure 8: Index space and the averaged query times of the adaptive variants in seconds, not limiting outputs. Suffixes *s* and *l* mean *small* and *large*, respectively.

however, is that adaptiveness and refined estimation of intersections now have little impact on the times. One reason for this is that now the cost to report so many results dominates the overall query time, thereby reducing the relative impact of using better or worse techniques to produce them. Indeed, our times limited to 1000 results suggest that adaptive VEOs produce results *sooner* along the query process than global VEOs.

The fact that the enumeration of results dominates makes `RDFCSA`-large the clear winner of "medium" size, not only on the median but also on the average, where it outperforms `IRing`-large by a factor of two. MillenniumDB also benefits from reporting many results, because of its locality-friendly data layout. It outperforms `IRing`-large by a factor of 3.5 and `RDFCSA`-large by a factor of 1.8 on the average. On the medians, MillenniumDB is 16 times faster than `IRing`-large and 4 times faster than `RDFCSA`-large.

Table 4 summarizes the main statistics on the best performing variants, separated by query type. It can be seen that `RDFCSA`-large now clearly outperforms all the `IRing` variants even on type-III queries. The recommendation for users is then to choose, from those alternatives, the largest one they can fit in main memory: `IRing`-small using 7.30 bpt, `IRing`-large using 12.15 bpt, `RDFCSA`-large using 23.54 bpt, or, if possible, MillenniumDB using 156.78 bpt. We note that these systems may be part of wider applications where not all the main memory can be allocated to them, so it is very valuable to have

System	Space (bpt)	Average Gl	Average Ad	Median Gl	Median Ad	Timeouts Gl	Timeouts Ad
Ring-small	7.30	83.6	81.8	2.9	2.9	101	99
IRing-small	**7.30**	82.3	**80.1**	2.7	**2.7**	99	**94**
Ring-large	12.15	46.8	44.2	0.9	0.9	59	53
IRing-large	**12.15**	45.4	**42.9**	0.8	**0.8**	58	**54**
URing-small	14.61	75.1	71.9	2.2	2.0	94	86
IURing-small	14.61	72.7	73.9	2.0	2.0	89	90
URing-large	23.53	47.4	46.3	1.0	1.0	59	58
IURing-large	23.53	45.2	43.1	0.8	0.8	58	55
VRing-small	35.42	84.6	83.6	2.9	2.9	99	97
VRing-large	40.28	45.2	44.5	0.9	0.9	55	53
VURing-small	42.74	75.1	72.5	2.2	2.1	89	85
VURing-large	51.65	46.7	46.3	1.0	1.0	55	56
RDFCSA-small	15.85	43.3	42.2	0.9	0.9	44	43
RDFCSA-large	**23.54**	22.5	**21.3**	0.2	**0.2**	31	**26**
MillenniumDB	**156.78**	**12.0**		**0.05**		**16**	

Table 3: Space and query times (in sec) of all the systems, with Gl(obal) and Ad(aptive) VEOs, not limiting the results. Timeouts count queries exceeding 10 min.

indices that can perform reasonably well within little space.

8. Conclusions

We have introduced new compact indices, combined with novel query resolution strategies, to solve Basic Graph Patterns on graph databases using Leapfrog TrieJoin (LTJ), the leading worst-case-optimal (WCO) multijoin algorithm. Concretely:

- We uncover a *space-time tradeoff* formed by compact indices. The Pareto-optimal variants use from about 0.6 to 2.0 times the space needed to store the triples in raw form. These compressed indices are outperformed only by the classic LTJ implementation in MillenniumDB (Vrgoc et al., 2023), which uses 14 times the space needed to store the triples.

- We combine those new indices with *adaptive variable elimination orders*, in contrast with the global orders in use. We show that adaptively choosing the next variable to eliminate along the query process yields

System	Space	Type I		Type II		Type III	
	(bpt)	Avg	Med	Avg	Med	Avg	Med
`IRing`-small	7.30	25.5	0.100	105.4	7.80	150.4	21.89
`IRing`-large	12.15	10.3	0.034	52.1	2.23	103.1	5.85
`IURing`-large	23.53	10.2	0.036	51.8	2.53	104.7	7.14
`RDFCSA`-large	23.54	3.1	0.004	24.5	0.74	60.6	1.84

Table 4: The best performing indices, separated by query type, without limiting the results. Times are given in seconds.

results much sooner: our adaptive query plans are up to 5 times faster to obtain the first 1000 results, and they even outperform in many cases the *best possible* non-adaptive plan. The time to obtain all the results, instead, is nearly the same.

- We show that using the total number of leaves descending from an LTJ trie node yields much better variable elimination orders compared with the classic measure of the number of children of the node: while the latter approximates the cost of performing the next intersection, the former better estimates the whole future cost. Our better estimation speeds up query resolution by a factor over 2, and almost 8 when combined with adaptive plans.

- We also show that the estimation of the intersection size can be refined by using features that are unique of our compact representation, which further speeds up the adaptive query times by a factor of up to 2.4. Further, we show that there is much space for improvement in terms of choosing a good variable ordering when generating query plans.

Overall, our new representations outperform the original *ring* by a factor up to 13 to produce the first 1000 results, while using the same space, and by 2 overall using about twice the space; this doubled space is still several times less than those of classical indices. Classical WCO and non-WCO indices are (often sharply) outperformed by our fastest variants. Only one of those, using 4 times the space of our largest relevant variant, outperforms it by a factor 2 on the average.

We remark that our compact indices run in main memory and would not be disk-friendly. While their compactness make them fit in memory for larger datasets, a relevant future work direction is to design compact representation

formats for disk or distributed memory, where compactness translates into fewer I/Os or communication at query resolution time.

Another limitation of our compact indices is that they do not yet support updates. A way to support updates is to replace the wavelet tree bitvectors of all the *ring* structures by dynamic variants (Mäkinen and Navarro, 2008). In this case, a node or triple can be inserted or deleted in the graph in $O(\lg n \lg \sigma)$ time. This, however, multiplies all the operation times by $O(\lg n)$, which is a high price especially if we consider that updates are not only infrequent in many use cases, but also that queries require many more accesses than updates. A recent development of dynamic bitvectors that are sensitive to the frequency of queries versus updates (Navarro, 2025) yields a promising implementation of dynamism in our scenario: we anticipate almost no increase in query times by supporting efficient updates in the graph. Further, we believe that several of our findings, in particular those related to better variable orderings, are independent from dynamism and would hold unchanged in the dynamic scenario.

Data availability statement

The data and source code that support the findings of this study are openly available in zenodo at `https://zenodo.org/records/13141588` and `https://zenodo.org/records/13142779`, respectively.

Acknowledgements

Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile. Antonio Fariña and Adrián Gómez-Brandón are funded in part by MCIN/AEI/10.13039/501100011033: grant PID2020-114635RB-I00 (EXTRACompact); by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe": PID2021-122554OB-C33 (OASSIS) and PID2022-141027NB-C21 (EARTHDL); by MCIN/AEI/10.13039/501100011033 and "NextGenerationEU" / PRTR: grants TED2021-129245B-C21 (PLAGEMIS), PDC2021-120917-C21 (SIGTRANS) and by GAIN/Xunta de Galicia: GRC: grants ED431C 2021/53. CITIC is funded by the Xunta de Galicia and co-financed by the EU through FEDER Galicia: grant ED431G 2023/01. Gonzalo Navarro is funded in part by Fondecyt Grant 1-230755, Chile.

References

Aberger, C.R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., Ré, C., 2017. Emptyheaded: A relational engine for graph processing. ACM Transactions on Database Systems 42, article 20.

Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A., 2024. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. ACM Transactions on Database Systems 29, article 5.

Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J., Rojas-Ledesma, J., Soto, A., 2021. Worst-case optimal graph joins in almost no space, in: Proc. 47th ACM International Conference on Management of Data (SIGMOD), pp. 102–114.

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

Barbay, J., Claude, F., Navarro, G., 2013. Compact binary relation representations with rich functionality. Information and Computation 232, 19–37.

Bonifati, A., Martens, W., Timm, T., 2019. Navigating the maze of Wikidata query logs, in: Proc. World Wide Web Conference (WWW), pp. 127–138.

Brisaboa, N., Cerdeira-Pena, A., de Bernardo, G., Fariña, A., Navarro, G., 2023. Space/time-efficient RDF stores based on circular suffix sorting. The Journal of Supercomputing 79, 5643–5683.

Clark, D., 1996. Compact Pat Trees. Ph.D. thesis. University of Waterloo.

Claude, F., Navarro, G., Ordóñez, A., 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. Information Systems 47, 15–32.

Erling, O., 2012. Virtuoso, a hybrid RDBMS/graph column store. Data Engineering Bulletin 35, 3–8.

Gagie, T., Kärkkäinen, J., Navarro, G., Puglisi, S., 2013. Colored range queries and document retrieval. Theoretical Computer Science 483, 36–50.

Gagie, T., Navarro, G., Puglisi, S.J., 2012. New algorithms on wavelet trees and applications to information retrieval. Theoretical Computer Science 426-427, 25–41.

Grossi, R., Gupta, A., Vitter, J., 2003. High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850.

Harris, S., Seaborne, A., Prud'hommeaux, E., 2013. SPARQL 1.1 Query Language. W3C Recommendation. `https://www.w3.org/TR/sparql11-query/`.

Hogan, A., Reutter, J.L., Soto, A., 2020. In-database graph analytics with recursive SPARQL, in: Proc. 19th International Semantic Web Conference (ISWC), pp. 511–528.

Hogan, A., Riveros, C., Rojas, C., Soto, A., 2019. A worst-case optimal join algorithm for SPARQL, in: Proc. 18th International Semantic Web Conference (ISWC), pp. 258–275.

Ji, S., Pan, S., Cambria, E., Marttinen, P., Yu, P.S., 2022. A survey on knowledge graphs: Representation, acquisition, and applications. IEEE Transactions on Neural Networks and Learning Systems 33, 494–514.

Kalinsky, O., Etsion, Y., Kimelfeld, B., 2017. Flexible caching in trie joins, in: Proc. 20th International Conference on Extending Database Technology (EDBT), pp. 282–293.

Khamis, M.A., Ngo, H.Q., Ré, C., Rudra, A., 2016. Joins via geometric resolutions: Worst case and beyond. ACM Transactions on Database Systems 41, article 22.

Mäkinen, V., Navarro, G., 2008. Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4, article 32.

Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A., 2018. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph, in: Proc. 17th International Semantic Web Conference (ISWC), pp. 376–394.

Manber, U., Myers, G., 1993. Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22, 935–948.

Manola, F., Miller, E., 2004. RDF Primer. W3C Recommendation. `http://www.w3.org/TR/rdf-primer/`.

Munro, I., 1996. Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 37–42.

Navarro, G., 2014. Wavelet trees for all. Journal of Discrete Algorithms 25, 2–20.

Navarro, G., 2025. Practical adaptive dynamic bitvectors. Software Practice and Experience 55, 1539–1559.

Navarro, G., Mäkinen, V., 2007. Compressed full-text indexes. ACM Computing Surveys 39, article 2.

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

Ngo, H.Q., 2018. Worst-case optimal join algorithms: Techniques, results, and open problems, in: Proc. 37th Symposium on Principles of Database Systems (PODS), pp. 111–124.

Ngo, H.Q., Porat, E., Ré, C., Rudra, A., 2012. Worst-case optimal join algorithms, in: Proc. 31st Symposium on Principles of Database Systems (PODS), pp. 37–48.

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

Nguyen, D., Aref, M., Bravenboer, M., Kollias, G., Ngo, H.Q., Ré, C., Rudra, A., 2015. Join processing for graph patterns: An old dog with new tricks, in: Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES), pp. 1–8.

Pusch, L., Conrad, T.O.F., 2024. Combining LLMs and Knowledge Graphs to reduce hallucinations in question answering. CoRR 2409.04181.

Raman, R., Raman, V., Rao, S.S., 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3, article 43.

Sadakane, K., 2003. New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 294–313.

Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G., 1979. Access path selection in a relational database management system, in: Proc. ACM International Conference on Management of Data (SIGMOD), pp. 23–34.

Sun, Q., Luo, Y., Zhang, W., Li, S., Li, J., Niu, K., Kong, X., Liu, W., 2024. Docs2kg: Unified knowledge graph construction from heterogeneous documents assisted by large language models. CoRR 2406.02962.

Thompson, B.B., Personick, M., Cutcher, M., 2014. The Bigdata®RDF Graph Database, in: Linked Data Management, pp. 193–237.

Veldhuizen, T.L., 2014. Triejoin: A simple, worst-case optimal join algorithm, in: Proc. 17th International Conference on Database Theory (ICDT), pp. 96–106.

Vrandecic, D., Krötzsch, M., 2014. Wikidata: A free collaborative knowledgebase. Communications of the ACM 57, 78–85.

Vrgoc, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J., 2023. MillenniumDB: An open-source graph database system. Data Intelligence 5, 560–610.

Wang, J., Trummer, I., Kara, A., Olteanu, D., 2023. ADOPT: Adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning. Proceedings of the VLDB Endowment 16, 2805–2817.