Noname manuscript No.
(will be inserted by the editor)

CompactLTJ: Space & Time Efficient Leapfrog Triejoin on Graph Databases

Diego Arroyuelo · Daniela Campos · Adrián Gómez-Brandón · Yuval Linker · Gonzalo Navarro · Carlos Rojas · Domagoj Vrgoč

Received: date / Accepted: date

Abstract Leapfrog Triejoin (LTJ) is arguably the most practical and popular worst-case-optimal (wco) algorithm for solving basic graph patterns in graph databases. Its main drawback is that it needs the database triples (subject, predicate, object) represented as paths in a trie, for each of the six orders of subject, predicate, and object. The resulting blowup in space makes most systems disregard LTJ or implement it only partially, which makes their corresponding algorithms non-wco. In this paper we show that, by using compact data structures, it is possible to build an index that at the same time matches the query time performance of the fastest classic wco index, and uses a fraction of the space of non-wco indices (which are much slower). Concretely, we make use of compact tree representations to store functional tries using one bit per trie edge, instead of one pointer, and further reduce the space by storing partial tries. Our most compact variant uses 5–6 times less space than classic wco implementations and 2–3 times less than classic non-wco systems. At solving queries, it is on par with the fastest classic wco system, and 30–40 times faster than non-wco systems. We further incorporate improved query resolution strategies into *CompactLTJ* variants, which makes it considerably faster than classic wco systems as well, on queries that do not output too many results. Finally, we show how *CompactLTJ* can incorporate dynamism without altering its performance, even under very demanding update regimes. We leave a public fully-functional implementation of *CompactLTJ* that can be directly used by practitioners.

Keywords Worst-case optimal joins; Leapfrog Triejoin; compact data structures; graph patterns; graph databases

D. Arroyuelo
DCC, Escuela de Ingeniería, Pontificia Universidad Católica de Chile & IMFD, Santiago, Chile
E-mail: diego.arroyuelo@uc.cl, dvrgoc@ing.puc.cl

D. Campos, Y. Linker, G. Navarro
IMFD & DCC, University of Chile, Santiago, Chile
E-mail: dcampos@dcc.uchile.cl,yuvallinkerg@gmail.com, gnavarro@uchile.cl

✉ A. Gómez-Brandón
IMFD & CITIC, Universidade da Coruña, A Coruña, Spain
E-mail: adrian.gbrandon@udc.es

C. Rojas
IMFD, Santiago, Chile
E-mail: carlos.rojas@imfd.cl

D. Vrgoč
Institute for Mathematical and Computational Engineering, Pontificia Universidad Católica de Chile & IMFD, Santiago, Chile.

1 Introduction

Natural joins are fundamental in the relational algebra, and generally the most costly operations to implement. A poor implementation choice can lead to unaffordable query times, so the implementation of joins has been a concern since the birth of the relational model. Apart from efficient algorithms to join two tables (i.e., pair-wise joins), database management systems sought optimized strategies (e.g., [43]) to join several tables (i.e., multijoins), where differences between good and bad plans can be huge in terms of efficiency. Multijoins were typically handled as sequences of pair-wise joins: a *query plan* was a binary expression tree where the leaves were the tables to join and the internal nodes were the (pair-wise) joins to carry out. The main optimization concern was to avoid huge intermediate results, much larger than the final outputs, at internal nodes of the expression tree.

The concept of a *worst-case optimal (wco)* algorithm [8] was coined to define a multijoin algorithm that does not produce those huge intermediate results. Formally, a wco algorithm takes time $\tilde{O}(Q^*)$, where Q^* is *the largest output size on some database instance* with the same table sizes of the given one ($\tilde{O}(Q^*)$ allows multiplying Q^* by terms that do

not depend, or depend only logarithmically, on the database size). They [8] proved that no pair-wise based multijoin algorithm can be wco. Several wco join algorithms were proposed since then [36, 37, 45, 21, 39, 35, 27].

Leapfrog Triejoin (LTJ) [45] is probably the simplest and most popular wco algorithm. At a high level, it can be regarded as reducing the multijoin by one *attribute* at a time, instead of by one *relation* at a time as in the pair-wise-join based query plans. LTJ chooses a suitable order in which the joined attributes will be *eliminated* (which means finding all their possible values in the output and branching on the subset of the output matching each such value). To proceed efficiently, LTJ needs the rows of each relation stored in a *trie* (or digital tree) where the root-to-leaf attribute order is consistent with the chosen attribute elimination order. Even though LTJ is wco with any elimination order, it turns out that, just like with the traditional query plans, there can be large performance differences when choosing different orders [45, 17]. This means, first, that choosing a good order is essential and, second, that LTJ needs tries storing each relation *in every possible order of its attributes*, that is, $d!$ tries for a relation with d attributes that can participate in joins.

This high space requirement shows up, in one form or another, in all the existing wco algorithms, and has become an obstacle to their full adoption in database systems. Wco algorithms are of particular interest in *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 [16] feature most prominently *basic graph patterns (BGPs)*, which essentially are a combination of multijoins and simple selections. The concept of wco algorithms, as well as LTJ, can be translated into solving BGPs on graph databases [17]. This is very relevant because typical BGPs correspond to large and complex multijoins by non-key attributes [39, 1, 20, 17], where non-wco algorithms can be orders of magnitude slower than wco ones [1]. Still, LTJ needs $3! = 6$ copies of the database in the form of tries, which even for three attributes is sufficiently space-demanding to discourage its full adoption (for example, one may restrict the query language to force labels to be constant in queries, so that only two tries are needed).

The implementation of various wco indices for graph databases seems to confirm that large space usage will be the price for featuring wco query times. For example, a wco version of Jena [17] doubles the space of the original non-wco version. Efficient wco implementations like Jena LTJ [17] and MillenniumDB [47] use around 14 times the space required to store the graph triples in raw form. The most popular systems for graph databases, like Jena [17], Virtuoso [13], RDF-3X [34], or Blazegraph [44], for example, give up on worst-case optimality in order to use "only" 5 to 7.5 times the size of a plain triple storage.

1.1 Our contribution

In this paper we show that, by using compact data structures, it is possible to achieve at the same time worst-case optimality—with an index that is as fast as the fastest classical ones and sometimes even faster—, while using much less space than the (orders of magnitude slower) classic indices—as little as 2.3 times the space of the raw triple data. More in detail:

1. We show how to implement the 6-trie wco LTJ algorithm in little space by adapting compact data structures for ordinal trees [18], in a way that requires only one *bit*, instead of one *pointer*, per trie edge. We further reduce space by storing only partial tries, using *trie switching* [5] to retain full functionality. The resulting structure, which we call *CompactLTJ*, uses 16%–18% of the space of classic LTJ implementations that store the 6 tries (MillenniumDB, Jena LTJ), and 30%–46% of the space used by other non-wco systems (Virtuoso, RDF-3X, Blazegraph). Our index matches the query time performance of the fastest wco system (MillenniumDB), while outperforming the others—particularly the non-wco systems—by a factor of 30–40.
2. We explore the use of adaptive variable elimination orders in LTJ, which recompute the best order as the join proceeds and better estimations are available. We further use an estimator for the next variable to bind that turns out to be more accurate. The combination obtains much more stable times than the traditional global-order strategy. For example, it makes *CompactLTJ* up to twice as fast as MillenniumDB to obtain the first 1000 results.
3. We incorporate dynamism into *CompactLTJ*, allowing the insertion and deletion of triples in the graph. By resorting to a recent technique to represent dynamic arrays and bitvectors so that the performance adapts to the frequency of the updates [30], we make *CompactLTJ* dynamic while retaining the performance of the static version, even under very high update frequencies (e.g., just 30% slower when receiving 1,000 updates between each pair of consecutive queries, and much less under more typical regimes).
4. We leave public versions of *CompactLTJ* that can be directly used by practitioners. These include our best static and dynamic variants and address practical issues like handling large outputs and using actual strings for IRIs and literals instead of numeric identifiers. The mapping from strings to internal identifiers and back is done through new compact data structures for string dictionaries, which add a modest amount of extra space and query time to the previous figures, both for static and dynamic variants of *CompactLTJ*.

Recent research [5] has shown that it is possible to go further in space reduction, so as to simulate the LTJ data

structures within 0.6 to 1.0 times the size of the raw triple data. This significant reduction has a cost in terms of time performance, however: we show in the experiments that *CompactLTJ* is 30–60 times faster than these compressed data structures. We also show that other recent indices that offer beyond-wco query time guarantees, like Graphflow [26], ADOPT [48], and EmptyHeaded [1], do outperform *CompactLTJ* on particularly difficult queries, but again use 3–8 times more space. The techniques we develop in this paper could be used to develop more compact versions of those more powerful indices as well.

An early partial version of this paper appeared in *Proc. GRADES'24* [4]. This extended version includes, most prominently, the trie-switching technique to further reduce space, the implementation of dynamism, and the handling of string identifiers, all with their corresponding experiments. We also describe a public software for practitioners.

2 Preliminary Concepts

2.1 Graph joins

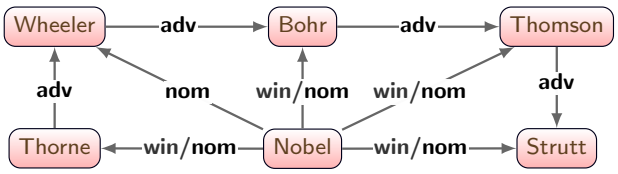
2.1.1 Edge-Labeled Graphs

Let $\mathcal{U}$ be a totally ordered, countably infinite set of *constants*, which we call the *universe*. In the RDF model [24], 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 (the *subject*) to vertex o (the *object*), with edge label p (the *predicate*). We call $N = |G|$ the number of triples in G. We also 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 $[1 \,..\, U]$, and will even assume $\mathcal{U} = [1 \,..\, U]$.

Example 1 Fig. 1 shows an example graph and the corresponding mapping of the constants in $\mathcal{U}$ to integers. Nodes are Physics researchers and the Nobel prize. Labels indicate researchers advised by others (adv), that were nominated to the Nobel prize (nom), and that won it (win). □

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 $\mathcal{U} \cap \mathcal{V} = \emptyset$. A *triple pattern* t is a tuple $(s, p, o) \in (\mathcal{U} \cup \mathcal{V})^3$,



Mapping

1=Bohr	2=Strutt	3=Thomson	4=Thorne	5=Wheeler
6=Nobel	7=adv	8=nom	9=win	

Triples

(1,7,3)	(3,7,2)	(4,7,5)	(5,7,1)	(6,8,1)	(6,8,2)	(6,8,3)
(6,8,4)	(6,8,5)	(6,9,1)	(6,9,2)	(6,9,3)	(6,9,4)	

Fig. 1: A labeled graph G with its string to integer mapping.

and a *basic graph pattern (BGP)* 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 BGP corresponds to a full conjunctive query (i.e., a *join query* plus simple selections) over the relational representation of the graph.

Let $\mathrm{vars}(Q)$ 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 $Q(G) := \{\mu : \mathrm{vars}(Q) \to \mathrm{dom}(G) \mid \mu(Q) \subseteq G\}$, called *solutions*, where $\mu(Q)$ denotes the image of Q under μ, that is, the result of replacing each variable $x \in \mathrm{vars}(Q)$ in Q by $\mu(x)$.

Example 2 A triple pattern, $(\mathsf{Nobel}, \mathsf{win}, x)$, on the graph G of Fig. 1, aims to bind variable x to all the values that make the triple (or edge) occur in G, namely Thorne, Bohr, Thomson, and Strutt. For example, in the first case, the triple $(\mathsf{Nobel}, \mathsf{win}, \mathsf{Thorne})$ is in G. Formally, and already mapping strings to integers, the query with this single triple is $Q_1 = \{(6, 9, x)\}$, and its evaluation on G is $Q_1(G) = \{\langle\mu(x) = 4\rangle, \langle\mu(x) = 1\rangle, \langle\mu(x) = 3\rangle, \langle\mu(x) = 2\rangle\}$.

Consider now the query formed by the triple patterns $(\mathsf{Nobel}, \mathsf{win}, x)$, $(\mathsf{Nobel}, \mathsf{win}, y)$, and (x, adv, y), which looks for pairs of Nobel winners where one was advised by the other. The answers are the pairs $(\mathsf{Bohr}, \mathsf{Thomson})$ and $(\mathsf{Thomson}, \mathsf{Strutt})$. On a relational table T_G with attributes (s, p, o), this query corresponds to the relational algebra formula

$$\rho(x/o)(\sigma_{s=\mathsf{Nobel},p=\mathsf{win}}(T_G)) \bowtie \rho(y/o)(\sigma_{s=\mathsf{Nobel},p=\mathsf{win}}(T_G)) \bowtie \rho(x/s, y/o)(\sigma_{p=\mathsf{adv}}(T_G)).$$

In our formalism, and translating to integers again, our query is $Q_2 = \{(6, 9, x), (6, 9, y), (x, 7, y)\}$. Its evaluation is $Q_2(G) = \{\langle\mu(x) = 1, \mu(y) = 3\rangle, \langle\mu(x) = 3, \mu(y) = 2\rangle\}$. □

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. [8], 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 still be applied to them by regarding each triple pattern as a relation formed by the triples that match its constants [17].

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. [8] proved that there are queries Q for which no plan using only pair-wise joins can be wco.

Example 3 The query Q_2 of Ex. 2 is a so-called triangle query. When it has no constants, the maximum output size of a triangle query on a graph of N edges is $O(N^{3/2})$. All pairwise-join strategies used in relational databases take time $\Theta(N^2)$ to solve the triangle query on certain graphs, and thus are not wco. A wco algorithm must solve the triangle query in time $\tilde{O}(N^{3/2})$. □

We describe next the the most frequently implemented wco algorithm.

2.2.2 Leapfrog TrieJoin (LTJ)

We describe a popular wco algorithm, Leapfrog Triejoin [45], originally designed for natural joins in relational databases, as it is adapted for BGP matching on labeled graphs [17]. This algorithm relies on the trie representation of the graph edges (i.e., the triples). To work properly, LTJ requires $3! = 6$ different tries to be stored, each representing the triples in a specific order of their components. The need for this requirement shall be made clear later. We call SPO, SOP, POS, PSO, OSP, and OPS these orders. For each triple $(s, p, o) \in G$, there is a corresponding root-to-leaf path labeled s, p, and o, in the SPO trie. Similarly, there is a path labeled s, o, and p in the SOP trie, and so on for the remaining orders. Consequently, each trie has height 3 and N leaves.

Example 4 Fig. 2 shows the 6 tries corresponding to the graph of Fig. 1; the root of each trie indicates the order. Disregard for now the marks τ on some nodes. □

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. 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 on $\text{vars}(Q)$, which we call a *variable elimination order (VEO)*.

Each triple pattern t_i, for $i = 1, \ldots, q$, is 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 tries in the 6 orders.

The algorithm starts at the root of every τ_i and descends by the children that correspond to the constants in t_i. We then proceed to the variable elimination phase. Assume the order of the variables is $\langle x_1, \ldots, x_v \rangle$ and let $Q_j \subseteq Q$ be the triple patterns that contain variable x_j. Starting with the first variable, x_1, LTJ finds each $c \in \text{dom}(G)$ such that for every $t \in Q_1$, c is a child of the current node of trie τ of t (if the trie τ of t is consistent with the VEO, then the children of its current node contain precisely the possible values c for x_1).

During the execution, we keep a mapping μ with the solutions of Q. As we find each constant c suitable for x_1, we *bind* x_1 to c, that is, we set $\mu = \langle x_1 := c \rangle$ and branch on this value c. In this branch, we go down by c in all the tries τ of triples $t \in Q_1$. We now repeat the same process with Q_2, finding suitable constants d for x_2 and increasing the mapping to $\mu = \langle x_1 := c, x_2 := d \rangle$, 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_j, LTJ backtracks and continues with the next binding for Q_{j-1}. When this process finishes, the algorithm has reported all the solutions for Q.

Example 5 Let us follow LTJ over the solution of query $Q_2 = \{t_1 = (6, 9, x), t_2 = (6, 9, y), t_3 = (x, 7, y)\}$ of Ex. 2, using the VEO $\langle x, y \rangle$. From the tries of Fig. 2, we will use SPO as the tries $\tau_1 = \tau_2$ for the triple patterns t_1 and t_2, and PSO as the trie τ_3 for the triple pattern t_3 (if we had chosen the VEO $\langle y, x \rangle$, τ_3 should have been POS). We then descend by the constants 6 and 9 in both τ_1 and τ_2, and by 7 in the trie τ_3. We reach the nodes marked τ_1, τ_2, and τ_3 in the figure.

We now bind variable x, whose candidates descend from the current nodes of τ_1 and τ_3. Their children in common are 1, 3, and 4. LTJ branches with each of those candidates. The branch with $\langle \mu(x) = 1 \rangle$ descends by the corresponding nodes in τ_1 and τ_3; we mark the reached nodes τ_1^x and τ_3^x. We now bind variable y, whose candidates descend from the current nodes in τ_2 and τ_3. Their only common children is

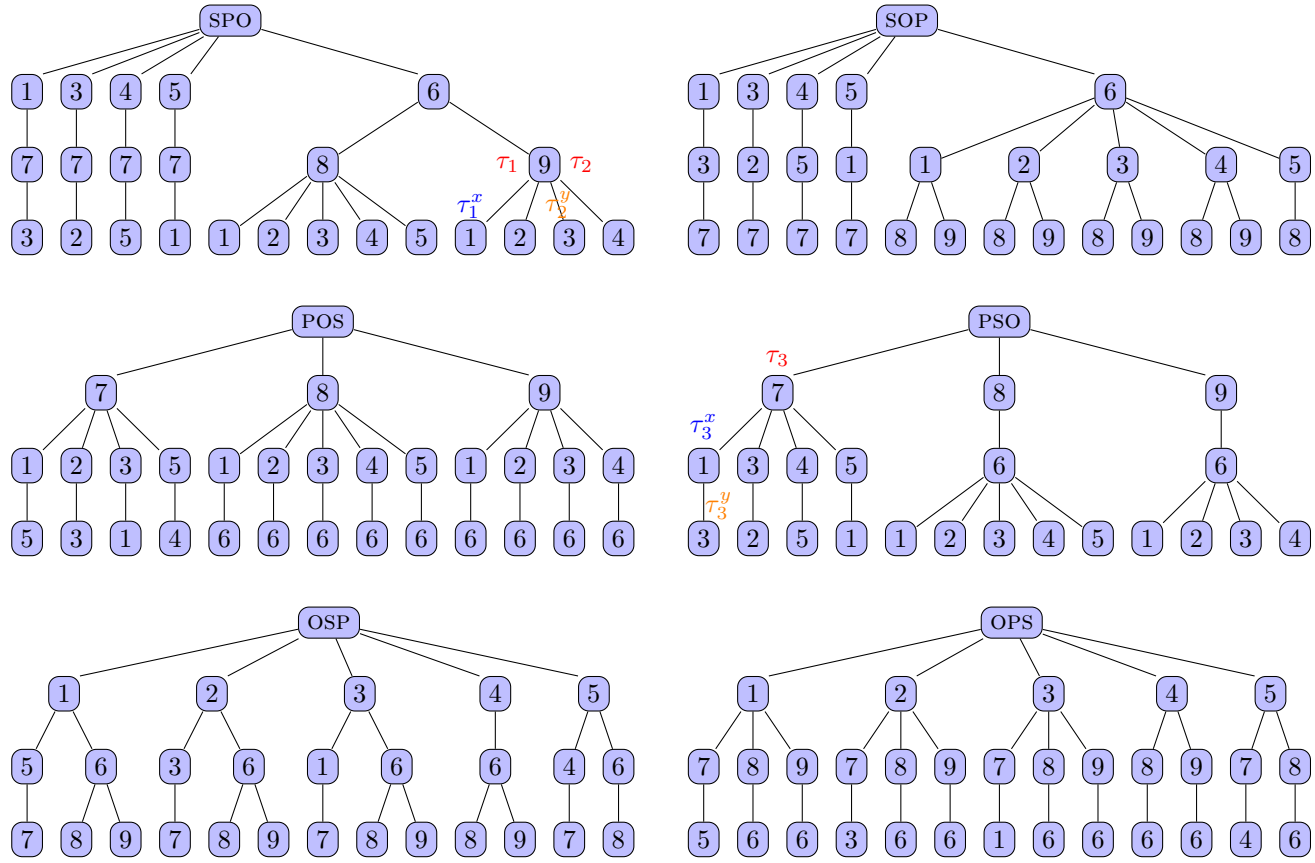


Fig. 2: The six tries corresponding to the graph of Fig. 1. The τ marks on some nodes correspond to Ex. 5.

3, so we complete the binding $\langle \mu(x) = 1, \mu(y) = 3 \rangle$; we mark the reached nodes τ_2^y and τ_3^y in the figure. Since we have bound all the variables, we deliver this binding as the first solution of $Q_2(G)$. The other solution is found when branching with $\langle \mu(x) = 3 \rangle$ and then growing it to $\langle \mu(x) = 3, \mu(y) = 2 \rangle$. Instead, the branch with binding $\langle \mu(x) = 4 \rangle$ does not produce any solution. □

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 $\mathsf{leap}(v_i, c)$, which finds the smallest constant $c_i \geq c$ within the children of the current node v_i in trie τ_i; if there is no such value c_i, $\mathsf{leap}(v_i, c)$ returns a special value $\perp$.

The intersection process is as follows. Assume that we are handling variable x_j from the VEO. For each $t_i \in Q_j$, we are at a specific node v_i in trie τ_i. The goal is to find the values that appear in the list of children of all such nodes v_i. We keep the next candidate to be in the intersection, $c_{\min}$, which is initialized to c. For each trie τ_i, we update $c \leftarrow \mathsf{leap}(v_i, c)$. After traversing all nodes v_i, if $c_{\min} = c$, we return $c_{\min}$ as the next value in the intersection. Otherwise, we reset $c_{\min}$ to c and restart the scan of trie nodes.

Every time a value c is returned, we know that c appears as a child of every v_i. We then launch, as explained, a recursive branch with the new binding $\langle x_{i_j} := c \rangle$. Upon returning from that recursion branch, we remove the binding $\langle x_j := c \rangle$ from μ and continue looking for other elements in the intersection. The intersection terminates when some $\mathsf{leap}()$ returns $\perp$, in which case the recursive call returns to Q_{j-1} (or LTJ finishes if $j = 1$).

Operation $\mathsf{leap}()$ uses exponential search [45, 9]: For each current node v_i we record where the previous $\mathsf{leap}()$ ended within its child list, and each new exponential search starts from that position. As a result, each $\mathsf{leap}()$ runs in $O(\lg \ell)$ time, where ℓ is the distance between the two latest ending positions and lg is the logarithm in base 2. While any polylogarithmic time guarantees that LTJ runs in wco time [45], exponential search is particularly effective in practice as $\mathsf{leap}()$ takes less time on nodes with more children.

Algorithm 1 shows the pseudocode for LTJ. It builds on just three primitives on the tries:

1. $\mathsf{child}(v, i)$, which descends to the ith child of node v,
2. $\mathsf{degree}(v)$, which computes the degree of a node, and
3. $\mathsf{access}(v, i)$, which reads the value of the ith child of v.

Algorithm 1 Evaluating the BGP $Q = \{t_1, \ldots, t_q\}$ with trie nodes $\tau_1, \ldots, \tau_q$ and variable ordering $\langle x_1, \ldots, x_v \rangle$. Symbol ':' in line 7 denotes concatenation.

LTJ($\{\tau_1, \ldots, \tau_q\}$)

1: **for** $i \in [1 \,..\, q]$ **do**
2: $v_i \leftarrow$ descend from the root of τ_i by the constants of t_i
3: leapfrog($\emptyset, 1$) // v_i and x_j are global variables

leapfrog(μ, j)

1: **if** $j = v + 1$ **then output** μ as a solution in $Q(G)$
2: **else**
3: $V \leftarrow \{v_i,\ t_i \in Q_j\}$ // i.e., x_j appears in t_i
4: $c \leftarrow$ seek($V, j, 1$)
5: **while** $c \neq \perp$ **do**
6: descend by c in every $v_i \in V$
7: leapfrog($\mu : \langle x_j := c \rangle, j + 1$)
8: return to the parent in every $v_i \in V$
9: $c \leftarrow$ seek($V, j, c + 1$)

seek(V, j, c)

1: **while** true **do**
2: $c_{\min} \leftarrow c$
3: **for** $v_i \in V$ **do**
4: $c \leftarrow$ leap(v_i, c)
5: **if** $c = \perp$ **then return** $\perp$
6: **if** $c_{\min} = c$ **then return** $c_{\min}$

Operations child and access are required at line 2 of LTJ and line 6 of leapfrog, while degree and access are required by leap(). Note that returning to the parent node, in line 8 of leapfrog, can be done by just remembering it before descending by c in line 6. Note also that, in this line 6, we do not really need to find the children that lead to c because we have already found them in seek. Storing those nodes is also useful to speed up the exponential searches of leap in line 4 of seek, so as to start from where the previous search ended.

2.3 Variable Elimination Orders (VEOs)

Veldhuizen [45] showed that if 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 [45, 17]. 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 [17, 47, 5] computes, for each variable x_j, its minimum weight

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

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 [48] 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. We will compute adaptive VEOs, instead, simply as a variant of the formula presented above for global VEOs [17].

Other systems go even further in this beyond-wco path. Building on the well-known Yannakakis' instance-optimal algorithm for acyclic queries [51], EmptyHeaded [1] applies a so-called Generalized Hypertree Decomposition [15], which decomposes cyclic queries into a tree where the nodes are cyclic components, so as to solve the nodes using a wco algorithm [36] and then apply Yannakakis' algorithm on the resulting acyclic query on the intermediate results. Graphflow [26], Umbra [33], and Free Join [49] are examples of systems that integrate wco joins with pairwise joins in order to generate hybrid plans for evaluating graph queries. Other approaches like Tetris [21], Minesweeper [38] and Panda [2] also offer guarantees finer than just wco.

2.4 Trie switching

Trie switching [5, Sec. 7.2.1] is a mechanism to decrease the space required by the six tries of LTJ, at the expense of possibly increasing query times by a small margin. The idea is that some parts of some tries are redundant with others and can be deleted. Consider the tries SPO and PSO. Once we have descended by instantiated values of p and s in PSO, reaching node v, and need to work (e.g., intersect) the children o of node v, we could instead *switch* to the equivalent node v' in the trie SPO, by descending by s and p from its root. The children of v and v' are the same, and therefore we can omit the children o of v in the trie PSO.

By using trie switching we can then store some *partial tries*, from which we switch to others in case of need. For example, we can store only the tries SPO, PS, POS, OP, OSP, and SO, thereby saving 3 of the 6 last levels of the tries, which are the biggest (each last level has exactly N nodes). Further, a clever implementation can share the first level of the tries PSO and PO, SOP and SP, and OPS and OS, further saving 3 of the 6 first levels of the tries (of total size $|\mathrm{dom}(G)|$).

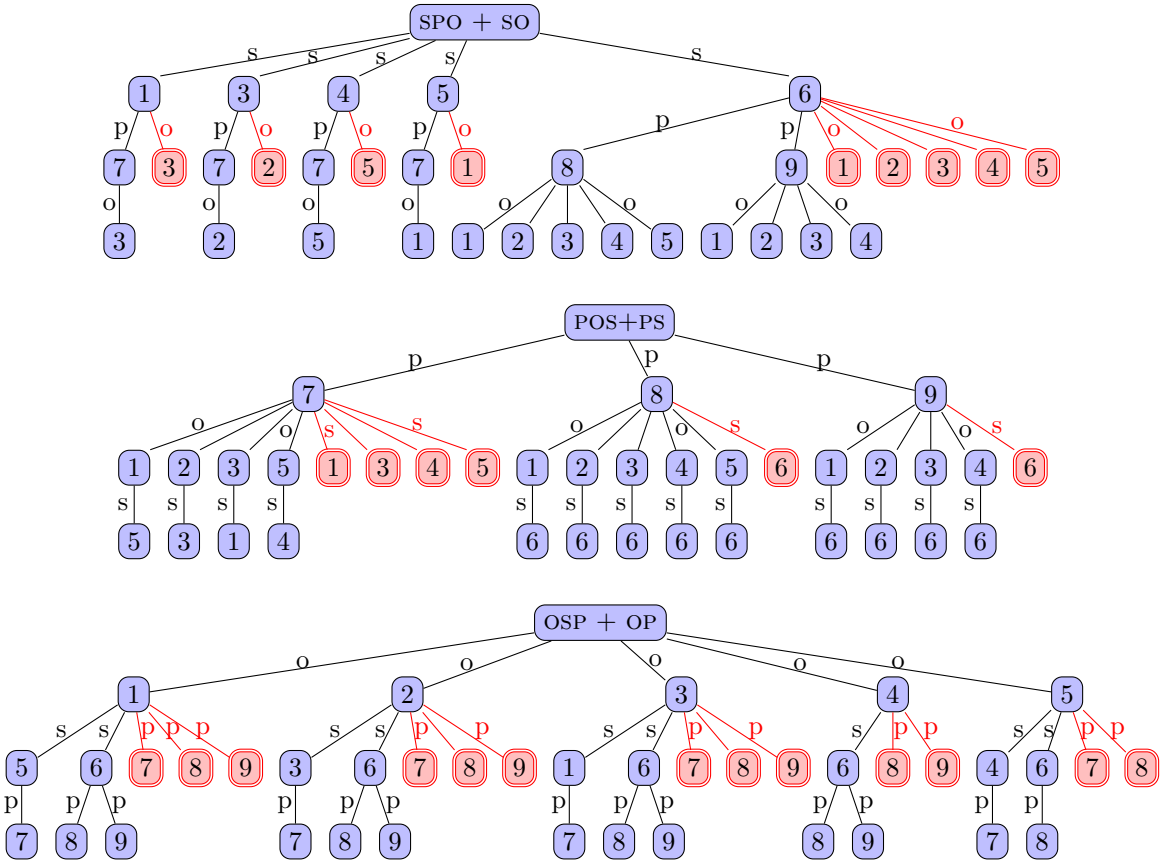


Fig. 3: A partial trie configuration for the six tries of Fig. 2. The set of red edges and nodes (which are also doubly-circled) corresponds to a second set of children that descend by another attribute from the same first-level nodes.

Example 6 Fig. 3 shows the partial tries corresponding to our running example. □

Overall, we expect a space saving between $1/4$ and $1/2$ with the use of partial tries (we will obtain around $1/3$ in practice in this paper). In exchange, some slowdown is to be expected due to the need to switch between tries, reentering another trie from the root instead of directly navigating to the children of the current node.

3 CompactLTJ: Leapfrog Triejoin on Compact Tries

We now introduce our compact representation of the LTJ tries, and combine them with techniques that improve the performance of the original proposal. Our index, *CompactLTJ*, represents separately the trie topology and the edge labels.

3.1 Trie topology

The Level-Order Unary Degree Sequence (LOUDS) [18] is a representation of n-node tree topologies using just $2n + o(n)$ bits. It is obtained by traversing the tree levelwise (with each level traversed left to right). We append the *encoding* 0^d1 of each traversed node to a bit sequence T, where d is the number of children of the node. The final sequence T represents the tree using two bits per node: a 0 in the encoding of its parent and a 1 to terminate its own encoding. A bitvector representation of T then needs $2n + o(n)$ bits, and allows navigating the tree in constant time.

Example 7 Fig. 4 shows the LOUDS representation of the trie SPO of Fig. 2 (ignore the bottom-right part and node names for now). □

Our trie topologies are particular in that all the leaves have the same depth, 3. Therefore, every internal node at depths 0–2 have children, and thus we can reduce their encoding to $0^{d-1}1$. The leaves need not be encoded, which further saves space. In the original encoding, then, every node with d children spends $d + 1$ bits (leaves, with $d = 0$, included). In our new code, it spends d bits per node (leaves included, which means they disappear). Thus, we save n bits and therefore halve the original space [18].

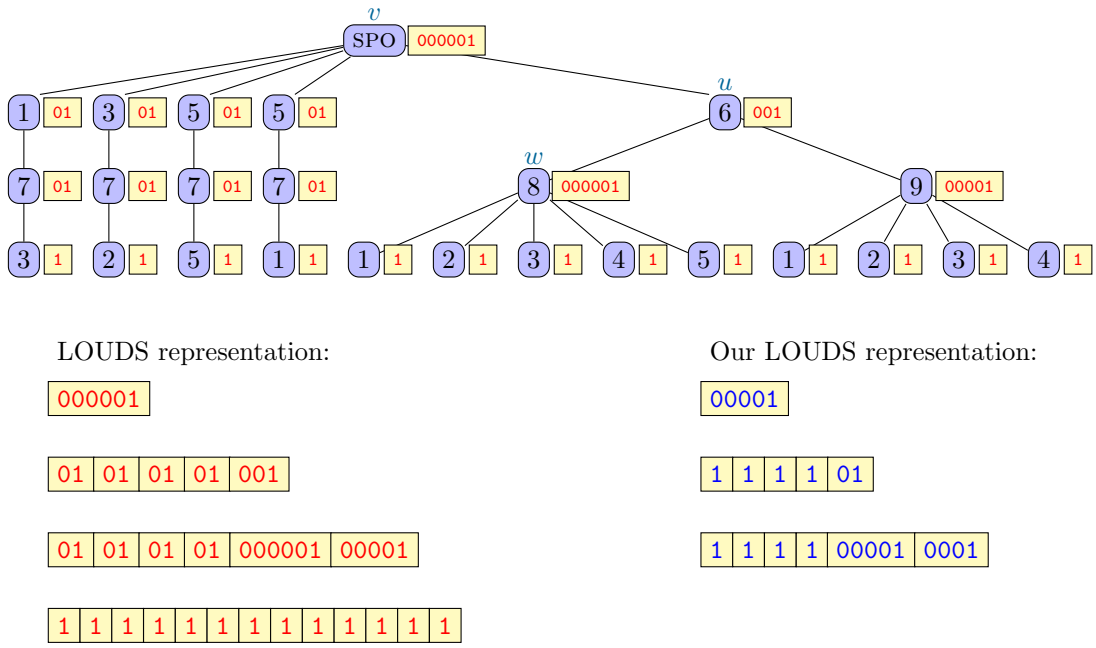


Fig. 4: The LOUDS representation of the trie SPO of Fig. 2. Besides each node, with d children, we show its encoding 0^d1. On the bottom left, the levelwise concatenation of the encodings. The LOUDS representation concatenates all the levels. The bottom right shows our shorter representation, which uses $0^{d-1}1$ instead of 0^d1 and removes the leaves.

Lemma 1 *Our representation uses $n-1$ bits on a trie of n nodes.*

Proof An internal node with d children is encoded as $0^{d-1}1$ and leaves are not encoded; therefore we store exactly as many bits as edges in the trie, that is, $n-1$ bits. □

Our encoding also simplifies the traversal compared to the original LOUDS [18]. We will use the position *preceding* the encoding of a node as its trie identifier v (e.g., $v=0$ for the root). The navigation to children makes use of the primitive $\mathsf{select}(T,j)$, which is the position of the jth occurrence of bit 1 in T. This primitive can be supported in $O(1)$ time using just $o(n)$ additional bits of space on top of T [11, 28] (see Appendix A for a description of this algorithm). With this primitive, we can navigate our representation as follows.

Lemma 2 *For every $v \geq 0$ and $i \geq 1$, it holds*

$$\mathsf{child}(v,i) = \mathsf{select}(T, v+i).$$

Proof Bitvector T lists, in levelwise order, all the trie nodes, using one 1-terminated code per node. Since the code consists of exactly one bit per child, T also lists all the trie edges, in levelwise order, using one bit per edge. The targets of those edges also form a levelwise enumeration of the nodes, just missing the root. It follows that the edge leading to the ith child of v is at position $T[v+i]$, and its target is the $(v+i+1)$th node in levelwise order (since the first is the root). The identifier of that node is the position preceding its code in T, which is the position of the $(v+i)$th 1 in T. □

Example 8 The bottom-right of Fig. 4 shows our more compact representation. For the trie SPO, we have:

$T =$ 00001 111101 1111000010001.

For example, the identifier of the root is $v=0$ and that of its fifth child is $u = \mathsf{child}(0,5) = \mathsf{select}(T, 0+5) = 9$. The encoding of node $u=9$ is at $T[u+1\,..\,u+\mathsf{degree}(u)] = T[10\,..\,11] =$ 01. See v and u in Fig. 4. □

In order to implement $\mathsf{leap}()$, we also need to determine the number of children of a node v. This is the distance from v to the next 1 in T:

$$\mathsf{degree}(v) = \mathsf{selectnext}(T, v+1) - v,$$

where the primitive $\mathsf{selectnext}(T,k)$ gives the position of the leftmost occurrence of 1 in $T[k\,..]$. This primitive can also be computed in $O(1)$ time using $o(n)$ additional bits of space; see again Appendix A.

Example 9 Continuing with Ex. 8, the first child of $u=9$ is $w = \mathsf{child}(9,1) = \mathsf{select}(T, 9+1) = 15$. Node $w=15$ has $\mathsf{degree}(15) = \mathsf{selectnext}(15+1) - 15 = 20 - 15 = 5$ children. See w in Fig. 4. □

3.2 Node identifiers

The node identifiers are stored in a compact array L, each one using $\lceil \lg U \rceil$ bits. The identifiers in L are deployed in

the same levelwise order of the edges T, so the identifiers of the children of node v are all consecutive, in $L[v+1\,..\,v+\mathsf{degree}(v)]$. This yields

$$\mathsf{access}(v,i) = L[v+i]$$

and allows implementing $\mathsf{leap}()$ efficiently by using exponential search from the current position. In our representation, we define $\mathsf{leap}([i,j],c)$ as the smallest $k \in [i\,..\,j]$ such that $L[k] \geq c$, or $j+1$ if no such k exists. With this convenient notation, the children of node v are searched as $\mathsf{leap}([v+1, v+\mathsf{degree}(v)], c)$.

Example 10 For our trie SPO in Fig. 4, the index would store

$T =$ 00001 111101 1111000010001

$L =$ 13456 777789 3251123451234

where, for example, the fifth ($i=5$) child of the root ($v=0$) descends by $L[0+5] = 6$ (to $u=9$, as shown before). The first child of u, by $L[9+1]=8$, leads to $w=15$. The children of w have labels $L[16\,..\,20] = 12345$. It then holds, for example, that $\mathsf{leap}([16,20],4) = 19$. □

3.3 xCompactLTJ: Using partial tries

We implement trie switching as follows. Consider the tries SPO and SOP in Fig. 2. Per trie switching, we choose to represent only SPO and SO (see Fig. 3); from the second we can switch to trie OSP if we need to access predicates from SO.

Example 11 Fig. 4 shows how the three levels of trie SPO are represented. The first level, corresponding to S, is represented with bits 00001 and identifiers 13456, exactly as the first level of SO (and SOP, which is not represented). To represent SO, then, we reuse the first level of SPO, and store only the second level, O, of the trie:

$T' =$ 00001 111100001

$L' =$ 13456 325112345

where the gray nodes are not represented (see Fig. 2 again, or the red nodes at the top of Fig. 3). □

As explained, in case we want to descend from a leaf of SO to the predicates, we reenter the trie OSP with the current values of o and s, respectively. This is the case where the use of partial tries may entail some time overhead.

In total, we represent 12 trie levels instead of 18. We call *xCompactLTJ* the *CompactLTJ* version using partial tries.

3.4 UnCompactLTJ: A non-compact variant

As a baseline to determine the slowdown incurred with compact representations like LOUDS, compared to classical ones, we introduce a version called *unCompactLTJ*, which is a minimal non-compact trie representation.

The *unCompactLTJ* index stores an array P of *pointers*, one per internal node, deployed in the same order of LOUDS. For each internal node v we store $P[v+1] = \mathsf{child}(v,1)$, that is, a pointer to its first child, knowing that the others are consecutive, that is, $\mathsf{child}(v,i) = P[v+i]$. Each pointer uses $\lceil \lg n \rceil$ bits, as it is a position within an array of n elements. The number of children of node v is simply $\mathsf{degree}(v) = P[v+1] - P[v]$, assuming $P[0] = 0$ if needed. Its array L of edge labels is identical to that of *CompactLTJ*, so $\mathsf{access}(v,i) = L[v+i]$ still holds.

Example 12 For our same Ex. 8 we have

$$P = \langle 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 20, 24 \rangle$$

(where $24 = |T|$ is a terminator). The root 0 has $P[1] - P[0] = 5$ children, at positions $P[1\,..\,5]$. Its fifth child descends by label $L[0+5] = 6$ to node $P[0+5] = 9$, which has two children, since $L[6] - L[5] = 2$. Its first child descends by label $L[P[5]+1] = L[9+1] = 8$ to node $P[9+1] = 15$. Node 15 has $P[11] - P[10] = 5$ children, at $L[16\,..\,20] = 12345$. □

In exchange for nearly doubling the space of *CompactLTJ*, *unCompactLTJ* has explicit pointers just like classical data structures, and does not spend time in computing addresses. As we show in the experiments, *unCompactLTJ* is only marginally faster than *CompactLTJ*, so the slowdown due to using compact data structures is mild. Still, *unCompactLTJ* uses half the space of Jena LTJ [17], a classic index that supports LTJ using the six tries (implemented as B+-trees).

3.5 Adaptive VEOs

Our orthogonal contribution is the study of improved VEOs on our compact LTJ tries, which deviate from the VEO defined in Section 2.3. The first improvement is the use of *adaptive* VEOs; the second is on the w_{ij} estimator used.

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 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.

3.6 CompactLTJ*: Better VEO predictors

The *CompactLTJ* index uses the original estimator based on the number of children of v, which is easily computed in constant time as $w_{ij} = \mathsf{degree}(v)$. We now define an alternative version, *CompactLTJ**, which computes w_{ij} as the number of leaf descendants of v. This is $w_{i,j} = N$ if v is in the first level (i.e., the root), and $w_{i,j} = \mathsf{degree}(v)$ if v is in the third level (i.e., just above the leaves). For the second level, we compute in constant time

$$w_{ij} = \mathsf{child}(v + \mathsf{degree}(v), 1) - \mathsf{child}(v, 1).$$

We argue that 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.

4 Dynamic CompactLTJ

A pitfall of most compact indices that support wco query times on BGPs [3, 5, 4] is that they are static, that is, they do not support insertions or deletions of triples in the graph. There is a good reason for this: while basic bitvector operations like select and selectnext can be supported in constant time in the static case [18, 28, 11], a lower bound of $\Omega(\lg n / \lg\lg n)$ when bit flips are allowed [14] imposes a significant gap between static and dynamic solutions. A gap of 10x does show up in practice [41, 12] and permeates through all the compact data structures, as bitvectors are basic components in most of them.

A recent development called *adaptive dynamic bitvectors* [30] offers a new tradeoff on this gap, however, that is especially relevant in our application: if a bitvector receives, on average, one bit insertion or deletion per q query operations (like select or selectnext), then all the operations run in $O(\lg(n/q))$ amortized time. Significant time improvements are shown over a classic dynamic implementation for $q \geq 1{,}000$ or so, and times very close to those of static implementations for about $q \geq 10{,}000$.

Now consider our bitvector T. A graph query Q running in time $\tilde{O}(Q^*)$ implies, essentially, that T will receive $\tilde{O}(Q^*)$ bitvector queries, which is typically a massive amount: in our benchmark, we carry out about a billion select operations per query limited to 1,000 results, and nearly 40 billion without limiting the results. This implies that, even if T received one update after every graph query Q, q would be between 10^9 and 10^{11}, and thus we could expect a performance very close to that of the static scenario. Further, we expect much fewer updates per graph query in practical applications. For example, Wikidata receives about 6,000–15,000 queries and 200–500 updates per minute[1], which corresponds roughly to 0.1 to 0.01 updates per graph query.

We now describe how the solution for bitvectors [30] is applied on T and integrated with an analogous solution for arrays, which is applied on L. The solution is adapted to the queries we need in order to obtain a dynamic version of *CompactLTJ* that performs almost as well as the static one. We will show that our solution is robust even under much more demanding conditions, where it receives up to 1,000 updates per graph query.

4.1 Insertions and deletion of triples

The insertion/deletion of a triple (s, p, o) in/from G requires updating the six tries.

Example 13 Assume we want to add to the graph of Fig 1 the fact that Thomson also advised Rutheford. This is expressed in a new triple $(\mathsf{Rutheford}, \mathsf{adv}, \mathsf{Thomson})$ we add to G. Let us assign Rutheford the integer identifier 10, so the triple we wish to add is $(s, p, o) = (10, 7, 3)$. This triple, reading its components in the 6 possible orders, must be inserted in the six tries. In particular, we must insert $(3, 10, 7)$ in the trie OSP. □

For insertion, we traverse the trie from the root. We first descend in the trie as much as possible with the elements of the triple t to insert, using $\mathsf{leap}()$ to find the correct child. Once $\mathsf{leap}()$ indicates that the child to follow does not exist, we must start inserting it, until we insert the whole triple t. Algorithm 2 shows the details, which we explain by following the insertion of our example tuple. It makes use of the primitive $\mathsf{insert}(A, c, i)$ on T and L, which inserts c at position i in A; we describe its implementation later.

[1] See https://grafana.wikimedia.org/d/000000170/wikidata-edits?from=now-7d&orgId=1&to=now&viewPanel=1 and https://grafana.wikimedia.org/d/000000489/wikidata-query-service?orgId=1&refresh=1m&viewPanel=44&from=now-7d&to=now for on-the-fly statistics.

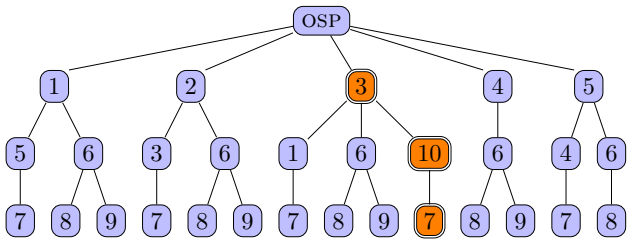


Fig. 5: The OSP trie of Fig. 2 after inserting triple $(3, 10, 7)$.

Example 14 The representation of trie OSP from Fig. 2 is as follows

$T =$ 00001 010101101 1011011010111

$L =$ 12345 563616646 7897897898978

We will insert tuple $t = (3, 10, 7)$ in this trie. Variable $exists$ in Alg. 2 indicates that the node to follow (initially $v = 0$, the root) exists. We take the first element of t, $c = 3$. Line 5 computes $v' = \mathsf{leap}([1, 5], c) = 3$, where $\mathsf{access}(0, 3) = c$, meaning that the child with identifier c exists. Line 7 descends to that child, $v = \mathsf{child}(0, 3) = 9$. The second tuple component, $c = 10$, is sought again in line 5 with $v' = \mathsf{leap}([10, 11], c) = \bot$, meaning that c is larger than every child of v. Line 10 establishes its insertion point, $v' = 12$. It then adds the new child c to the existing node v: it inserts a 0 at position $v' - 1 = 11$ of T and $c = 10$ at position $v' = 12$ of L, obtaining

$T =$ 00001 0101001101 1011011010111

$L =$ 12345 5636160646 7897897898978

where we have underlined the insertions. Note that in case the insertion point v' is within the children of v, we instead insert the 0 at position $T[v']$ in line 12. Line 14 descends to the (still nonexistent) child of the newly created node, $v \leftarrow \mathsf{child}(9, 3) = 24$, and $exists$ is set to false.

Once the desired node does not exist, the third tuple component, $c = 7$, is inserted, in lines 17 and 18, at positions $v + 1 = 25$ in T. We create a leaf by inserting a 1 in T and a c in L. Line 19 descends by that inserted position to keep inserting the elements of t, though in our example we have finished. The final result is

$T =$ 00001 0101001101 10110110110111

$L =$ 12345 5636160646 78978978978978

Fig. 5 highlights the path inserted on the OSP trie. □

We proceed in reverse order to delete a triple t: we enter the trie recursively, looking for each component c using $\mathsf{leap}()$, until reaching the leaf v corresponding to t. Then we remove nodes as we return from the recursion. To remove the leaf we use the primitive $\mathsf{delete}(T, v)$ and $\mathsf{delete}(L, v)$, which removes the given element from a bitvector or array and will be described later. There are three cases: (i) $T[v] = 0$, (ii) $T[v - 1, v] = 01$, (iii) $T[v] = 1$ and ($v = 1$ or $T[v - 1] = 1$). In case (i), the encoding of the parent node is 0^d1 with $d > 0$, so we are not removing its only child. To obtain the new correct encoding, $0^{d-1}1$, we perform $\mathsf{delete}(T, v)$ and $\mathsf{delete}(L, v)$, and the deletion process terminates. Case (ii) is similar, but there is a 1 at $T[v]$, so to obtain $0^{d-1}1$ we must do $\mathsf{delete}(T, v - 1)$ instead of $\mathsf{delete}(T, v)$ (plus $\mathsf{delete}(L, v)$). In case (iii), the signature of the parent is just 1, so we are deleting its only child. We then perform $\mathsf{delete}(T, v)$ and $\mathsf{delete}(L, v)$, and must keep removing the current node v as we return from the recursion.

Algorithm 2 Inserting tuple t in trie τ.

$\mathsf{trieInsert}(\tau, t)$

```
1:  v ← root of τ
2:  exists ← true
3:  for c ∈ t do
4:      if exists then
5:          v' ← leap([v + 1, v + degree(v)], c)
6:          if v' ≠ ⊥ and access(v, v' − v) = c then
7:              v ← child(v, v' − v)
8:          else
9:              if v' = ⊥ then
10:                 v' ← v + degree(v) + 1
11:                 insert(T, 0, v' − 1)
12:             else insert(T, 0, v')
13:             insert(L, c, v')
14:             v ← child(v, v' − v)
15:             exists ← false
16:     else
17:         insert(T, 1, v + 1)
18:         insert(L, c, v + 1)
19:         v ← child(v, 1)
```

4.2 Updates with partial tries

Partial tries may not require updates upon insertions and deletions in the graph. For example, if we inserted the triple (p, o, s) in the full trie POS and Alg. 2 did not create a new node in the second level for (p, o), then we do not need to insert the pair (o, p) in the partial trie OP. In case we do need to insert (o, p), we reuse the work done when inserting in the full trie OSP and found the insertion point v' in the first level.

Example 15 The trie OP in Fig. 3 is represented with just the sequences T' and L' corresponding to the red edges, which are complementary to those for OSP shown in Ex. 14.

$T' =$ 00001 0010010010101

$L' =$ 12345 7897897898978

where the elements in gray are not represented. In Ex. 14 we had obtained position $v' = 3$ for the node in first level. To

insert in OP, we would continue the insertion in T' and L' at position $v = \mathsf{child}(0, 3) = 11$. In this case, the insertion is not necessary because the pair $(p, o) = (7, 3)$ already exists in POS (see Fig. 2). Indeed, if we tried to insert the pair here, we would compute $v' = \mathsf{leap}([12, 14], 7) = 12$, and since $\mathsf{access}(v, 1) = L[12] = 7$, we would not insert it.

If, instead, o (and thus (o, p) and (p, o)) did not exist, o would be inserted in the first level of OSP. We would then simply insert in the second level of OP at position $\mathsf{child}(0, v') + 1$, inserting 1 in T' and p in L' of OP. □

Deletions are analogous. We determine that we must delete (o, p) from OP if the deletion of (p, o, s) from POS deleted the node representing (p, o).

4.3 Dynamic representation of T and L

Adaptive dynamic bitvectors [30] represent a bit sequence as a balanced binary tree with three kinds of nodes:

- *dynamic leaf*: allocates space for b elements (for a parameter b) and supports updates. Queries on the leaf are solved by sequential scanning.
- *static leaf*: stores subsequences with more than b elements and does not support updates. It precomputes some extra information to speed up queries, for example to solve select in constant time.
- *internal node*: stores pointers to both children and various statistics about the subtree rooted at the node. For example, *size* records the number of elements represented in the subtree and *ones* records the number of 1s.

To insert or delete a bit at a given position, the algorithm descends from the root of the binary tree guided by the *size* of the nodes, until reaching a leaf. If the leaf is dynamic, the update is applied. Otherwise, the leaf is static and the update is not possible. By a procedure called *split*, the leaf is recursively halved until the update reaches a dynamic leaf and then can be applied on it.

Queries proceed analogously. For example, select proceeds from the root to a leaf guided by the number of *ones* in the subtrees. When reaching a leaf, it completes the query by scanning if the leaf is dynamic, or using the precomputed structures if it is static; see Alg. 3. When a high amount of queries traverse an internal node, it is *flattened* into a static leaf, so as to speed up further queries.

We will use adaptive dynamic bitvectors to represent our bitvector T of Section 3. In the implementation, they also provide a version to handle arrays of values, which we use to represent our array of identifiers L. In our compact tries, inserting/deleting a trie node boils down to updating the same positions in T and L. We then modify their implementation so as to store T and L aligned together in the leaves of the same binary tree. This is advantageous because we typically need to descend by the binary trees only once, and also save some space.

Algorithm 3 Running $\mathsf{select}(j)$, $\mathsf{selectnext}(i)$, and $\mathsf{leap}([b, e], c)$ on a binary tree representing a dynamic bitvector, rooted at v.

$\mathsf{dynSelect}(v, j)$

if v is an internal node **then**
  if $v.\text{left.ones} \geq j$ **then return** $\mathsf{dynSelect}(v.\text{left}, j)$
  else return $\mathsf{dynSelect}(v.\text{right}, j - v.\text{left.ones})$
if v is a static leaf **then**
  return $\mathsf{select}(j)$ on the $O(1)$-time structure of v
else return $\mathsf{select}(j)$ by sequentially scanning v's sequence

$\mathsf{dynSelectNext}(v, i)$

if v is an internal node **then**
  if $v.\text{left.size} < i$ **then**
    return $\mathsf{dynSelectNext}(v.\text{right}, i - v.\text{left.size})$
  $pos \leftarrow \mathsf{dynSelect}(v.\text{left}, i)$
  if $pos \neq \perp$ **then return** pos
  else return $\mathsf{dynSelectNext}(v.\text{right}, 1)$
if v is a static leaf **then**
  return $\mathsf{selectnext}(i)$ on the $O(1)$-time structure of v
else return $\mathsf{selectnext}(i)$ by sequentially scanning v's sequence

$\mathsf{dynLeap}(v, b, e, c)$

if $b = 1$ and $e = v.\text{size}$ and $c > v.\text{last}$ **then return** $\perp$
if v is an internal node **then**
  $s \leftarrow v.\text{left.size}$
  if $s \geq e$ **then return** $\mathsf{dynLeap}(v.\text{left}, b, e, c)$
  if $s < b$ **then return** $\mathsf{dynLeap}(v.\text{right}, b - s, e - s, c)$
  if $c \leq v.\text{left.last}$ **then return** $\mathsf{dynLeap}(v.\text{left}, b, s, c)$
  return $\mathsf{dynLeap}(v.\text{right}, 1, e - s, c)$
return $\mathsf{leap}([b, e], c)$ using exponential search on v's sequence

The implementation of dynamic bitvectors [30] already provides the query select, which we use to implement child, but we had to incorporate support for selectnext and leap. See Alg. 3 again.

- $\mathsf{selectnext}(v, i)$ is solved by running a top-down traversal, where in each internal node we go right if i is on the right child. If i is on the left child, we first try to find the answer on the left child, and if it is not there, we search the right child from the first position. If the recursion ends in a dynamic leaf, selectnext is solved by a sequential scan; if it ends on a static leaf, it is solved in constant time with its precomputed structures.
- $\mathsf{leap}([b, e], c)$ is invoked only when $L[b\,..\,e]$ is increasing. To compute it efficiently, we maintain in each internal node v the field *last*, which stores the last value of L in the subtree rooted at v. We descend from the root, left or right as long as $[b\,..\,e]$ is completely included in one child. If we arrive at an internal node v whose children split $[b\,..\,e]$, then we continue to the left child if

$c \leq v.\text{left}.\text{last}$, otherwise we continue to the right child. Further, we immediately return from v without an answer if its subtree is completely contained in $[b\,..\,e]$ and $v.\text{last} < c$. In the leaves (both static and dynamic), the answer is found by running an exponential search.

5 Experimental Results

We compare our compact indexing schemes with 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, 378 GB RAM.

5.1 Datasets and queries

We run a benchmark over the Wikidata graph [46], which we chose for its scale, diversity, prominence, data model (i.e., labeled edges) and real-world query logs [23, 10]. For now we assume that node and label identifiers are integers in a contiguous range, which can be obtained after a suitable preprocessing of the graph; we consider later how to deal with the actual strings. Our Wikidata graph features $N = 958{,}844{,}164$ triples, which take 10.7 GB if stored in plain form using 32 bits for the identifiers.

We consider a real-world query log [23]. 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.

Example 16 Query Q_1 from Ex. 2 is of type I, whereas query Q_2 is of type III. An example of a query of type II is $Q = \{(6, 9, x), (x, 7, y)\}$, which finds Nobel prize winners and their advisors. □

All queries were run with a timeout of 10 minutes and a limit of 1000 results (as originally proposed for WGPB [17]). 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.

System	Space (bpt)	Average (msec) Global	Average (msec) Adaptive	Median (msec) Global	Median (msec) Adaptive
`xCLTJ`	27.56	489	415	0.9	1.5
`xCLTJ*`	27.56	246	78	1.4	1.3
`CLTJ`	39.58	421	394	0.6	0.6
`CLTJ*`	39.58	187	52	0.6	0.6
`UnCLTJ`	56.34	454	375	0.6	0.5
`UnCLTJ*`	56.34	204	46	0.6	0.5

Table 1: Space and query times of the compact LTJ variants, limiting results to 1000, with global and adaptive VEOs.

We leave the experiments on dynamic representations to the end of the section. Our dynamic CLTJ variants perform identically to the static ones when there are no updates, thanks to the use of dynamic adaptive bitvectors and sequences.

5.2 Compact LTJ variants

Table 1 compares the indices *CompactLTJ*, *xCompactLTJ*, and *unCompactLTJ* described in Section 3, calling them respectively `CLTJ`, `xCLTJ`, and `UnCLTJ`. The versions `CLTJ*`, `xCLTJ*`, and `UnCLTJ*`, in turn, use the VEO predictor described in Section 3.6. All of them compute the VEO in traditional form ("global VEO") and in adaptive form (Section 3.5). No variant gave any timeout in this experiment.

The space of the `CLTJ` index is just 3.3 times the size of the raw data encoded as a set of n 32-bit triples, whereas `UnCLTJ` uses 4.7 times the size (i.e., 42% more than `CLTJ`). The reward for using that 42% extra space is not significant, which shows that the space reduction obtained with `CLTJ` comes at essentially no loss in time performance. On the other hand, `xCLTJ` uses just 2.3 times the size of the raw data, which is a 30% space reduction over the space of `CLTJ`. This space reduction comes at a price in time, however: the adaptive variant of `xCLTJ*` is 50% slower than that of `CLTJ*` on the average, and over twice as slow in the median. The extra time owes to the need of switching tries.

While the medians of all the different variants are in the range of 0.5–1.5 milliseconds per query, the averages show that some query strategies yield much more stable times, and thus a lower average. The large difference between average and median query times shows that, although many queries are solved fast, there are others that take much longer, and it is important to better deal with them. In particular, combining adaptive VEOs with the modified VEO predictor (Section 3.6) reduces the average query times by almost an order of magnitude, to around 40–80 milliseconds. Using adaptive VEOs alone produces a very modest improvement, and using the modified VEO predictor with global VEOs only halves the time. The sharp improvement is obtained with the combination of both techniques.

In the sequel we will use only the variants xCLTJ∗, CLTJ∗, and UnCLTJ∗ with adaptive VEOs.

5.3 Comparison with other systems

We now put our results in context by comparing our compact LTJ indices with various graph database systems:

Wco systems: Systems that guarantee the AGM bound.
- Ring [5], a recent compressed in-memory representation that simulates all the 6 tries in a single data structure. Ring-large and Ring-small correspond to the versions called Ring and C-Ring, respectively, in their paper.
- MillDB [47]: 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 MillDB over a RAM disk to avoid using external memory.
- Jena LTJ [17]: An implementation of LTJ on top of Apache Jena TDB. All six different orders on triples are indexed in B+-trees, so the search is always wco.

Non-wco systems: Older systems not reaching the AGM bound, yet well established and optimized.
- RDF-3X [34]: 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. It handles triple patterns by scanning ranges of triples and uses a query optimizer over pair-wise joins.
- Virtuoso [13]: 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.
- Blazegraph [44]: The graph database system hosting the official Wikidata Query Service [23]. We run the system in triples mode, with B+-trees indexing orders SPO, POS, and OSP. It supports nested-loop joins and hash joins.

Beyond-wco systems: Recent systems combining wco and non-wco strategies. We could run only one at this scale.
- UmbraDB [33]: A system based on relational tables whose query plans use binary joins but may introduce wco plans for some sub-queries. Those plans are executed with LTJ using hash-based tries that are built on the fly as needed. Queries are compiled into executable multithreaded code. Because this system builds most indexes at query time, we measured the memory usage of the process.

We exclude Graphflow [26], Kùzu [19], DuckDB [42], ADOPT [48], and EmptyHeaded [1] because our server does not have enough memory to build or run them (they did not even run on another server with 768 GB of RAM). Most of those are beyond-wco systems. Section 5.5 compares them on harder queries and a smaller graph, where their stronger join strategies can be put in action.

In all systems, 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.

Table 2 shows the resulting time, space, and timeouts. A first observation is that, while the Ring variants use considerably less space than our smaller variant, xCLTJ (2.3–3.8 times less space, even less than the raw data), this comes at a considerable price in time performance: the Ring variants are 30–40 times slower than xCLTJ∗ on average, and 6–18 times slower in the median. While the small space of the Ring variants can be crucial to operate in main memory where other representations do not fit, xCLTJ∗ (and CLTJ∗) are much faster alternatives when they fit main memory. This is in part because the compressed representation of the Ring takes $O(\lg n)$ time to access any value, whereas CLTJ variants access them in constant time, and in part because the Ring's simulation of trie traversal operations is considerably more complex.

Interestingly, the CLTJ∗ variants are faster than non-compact wco systems that use 6 tries represented in classic form: MillDB and Jena LTJ. The faster one, MillDB, uses 4 times the space of CLTJ∗ and is twice as slow on average and 50 times slower in the median. The classic non-wco systems are somewhat smaller, but still 2–3 times larger than xCLTJ∗ and 60–120 times slower on average. UmbraDB outperformed the non-wco systems and Jena LTJ in time, but it is dominated in time and space by MillDB.

Table 3 shows how the times distribute across the three query types. It is interesting that MillDB is much slower than the CLTJ∗ variants only for query types I and II, which are the easy ones, whereas the average times on the hardest queries, of type III, are closer (and MillDB outperforms xCLTJ∗). This observation, and the consistently larger median times, suggest that MillDB performs some internal setup per query that requires several tens of milliseconds. We return to this point next.

On the other hand, some non-wco indices are competitive with MillDB (and outperform Jena-LTJ) on type I queries, but worsen on type II, and worsen much more on type III, as expected from theory. Similarly, UmbraDB is outperformed by the best non-wco systems by a factor of 2 on type I queries, matches them on type II queries, and sharply outperforms them on type III queries, showing that it handles complex queries well. Still, it is 2–5 times slower than MillDB for all query types (4 times slower in total).

System	Space (bpt)	Average (msec)	Median (msec)	Timeouts (> 10 min)
Ring-small	7.30	3056	24	5
Ring-large	12.15	2256	8	3
xCLTJ∗	27.56	78	1.3	0
CLTJ∗	39.58	52	0.6	0
UnCLTJ∗	56.34	46	0.6	0
Virtuoso	60.07	4880	50	8
RDF-3X	85.73	8230	126	13
Blazegraph	90.79	9220	54	14
MillDB	156.78	96	27	0
Jena LTJ	168.84	1930	162	1
UmbraDB	199.67	402	224	0

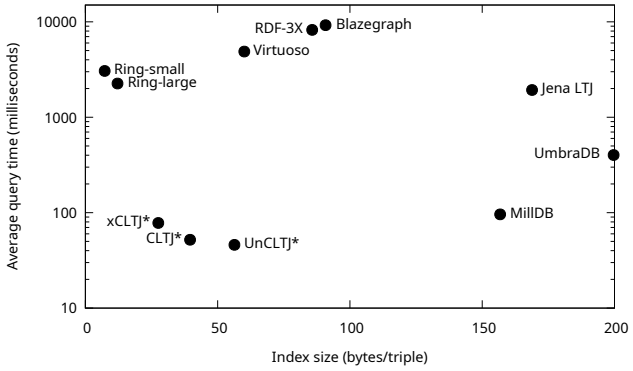


Table 2: Space and average query times of various systems, limiting results to 1000.

System	Space (bpt)	Type I Avg	Type I Med	Type II Avg	Type II Med	Type III Avg	Type III Med
Ring-small	7.30	12	8.0	380	36	6620	88
Ring-large	12.15	3.7	4.0	97	8.0	2448	28
xCLTJ∗	27.56	3.1	0.4	16	1.7	466	5.0
CLTJ∗	39.58	2.3	0.3	15	0.9	299	3.1
UnCLTJ∗	56.34	1.9	0.2	13	0.8	259	2.5
MillDB	156.78	50	20	79	27	267	73
Jena LTJ	168.84	449	65	691	175	9418	1059
Virtuoso	60.07	51	27	374	58	24769	126
RDF-3X	85.73	359	62	390	94	53144	1827
Blazegraph	90.79	44	27	6451	110	68011	310
UmbraDB	199.67	97	99	353	259	1363	543

Table 3: The best performing indices, separated by query type, limiting outputs to 1000 results. Times are in msec.

5.4 Not limiting the number of results

The case without limits in the number of answers is shown in Table 4. The times are much higher and thus the scale measures seconds. An important difference is that adaptiveness has almost no 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.

System	Space (bpt)	Average Gl	Average Ad	Median Gl	Median Ad	Timeouts Gl	Timeouts Ad
xCLTJ	27.56	15.7	15.7	0.08	0.11	21	20
xCLTJ∗	27.56	18.5	16.2	0.11	0.11	26	20
CLTJ	39.58	13.6	14.2	0.05	0.06	18	18
CLTJ∗	39.58	15.4	13.9	0.06	0.06	19	18
UnCLTJ	56.34	13.1	13.7	0.05	0.06	16	18
UnCLTJ∗	56.34	16.0	14.3	0.06	0.07	22	19

Table 4: Space and query times (in sec) of compact LTJ variants, with Gl(obal) and Ad(aptive) VEOs, not limiting the results. Timeouts count queries exceeding 10 min.

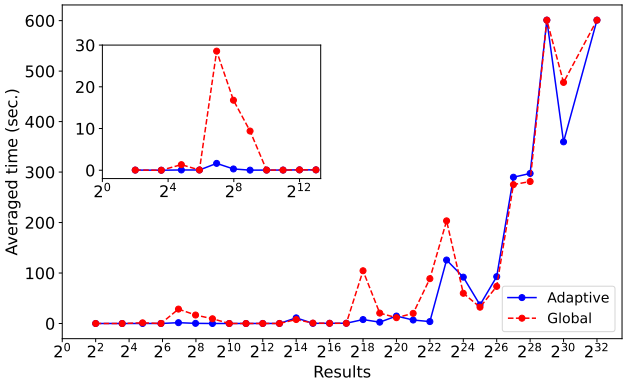


Fig. 6: Average time per type-III query as a function of the output size, for both variants of CLTJ∗.

To confirm this intuition, Fig. 6 shows the average query times as a function of their output size, for CLTJ∗ with global and adaptive VEOs. As it can be seen, the adaptive variant is much more robust than the global one. When the result is so large that the time to output it blurs the time taken to obtain it, both lines become similar. This is typically the case of queries with several lonely variables, whose binding is (wisely) left to the end and the query must output their Cartesian product. When we average over all the queries, as in Table 4, the average time is dominated by those queries with massive outputs.

The fact that a much larger fraction of the time is spent in outputting results also makes xCLTJ∗ similar in average query times to CLTJ∗, while still being 30% smaller. Its median times are still twice as large, though.

Tables 5 and 6 show the results of the best performing variants (in time or space), globally and by query type. MillDB fares better than with the limit, becoming similar to the CLTJ∗ variants and outperforming them on queries of type I and II, arguably because of the better locality of reference of the B+-trees to report many results. On queries of

System	Space (bpt)	Average (sec)	Median (sec)	Timeouts (> 10 min)
Ring-small	7.30	83.6	2.9	101
Ring-large	12.15	46.8	0.9	59
xCLTJ∗	27.56	16.2	0.11	20
CLTJ∗	39.58	13.9	0.06	18
UnCLTJ∗	56.34	14.3	0.07	19
MillDB	156.78	12.0	0.05	16

Table 5: Space and query times of the best performing indices, not limiting the results.

System	Space	Type I		Type II		Type III	
	(bpt)	Avg	Med	Avg	Med	Avg	Med
Ring-small	7.30	25.9	0.112	106.7	7.93	157.1	21.79
Ring-large	12.15	10.5	0.044	53.1	2.57	107.7	7.53
xCLTJ∗	27.56	1.9	0.003	16.6	0.39	52.8	0.91
CLTJ∗	39.58	1.3	0.001	15.1	0.24	45.9	0.60
UnCLTJ∗	56.34	1.3	0.001	14.8	0.23	47.8	0.71
MillDB	156.78	0.3	0.013	9.7	0.17	50.0	0.65

Table 6: The best performing indices, separated by query type, without limiting the results. Times are in seconds.

type III, where the query plan matters most, they all perform similarly.

5.5 Harder queries

We have compared the systems on real-life queries and at large scale. We now study more in depth the ability of various systems to handle particular query shapes that can be difficult to handle, on a smaller dataset. From our previous systems, we include the best wco systems and the beyond-wco system UmbraDB. We also include new systems that could not be run at full scale:

- DuckDB [42]: A non-wco relational query engine that stores the tables in columnar form on disk. It uses vectorized and pipelined models to implement optimized binary join plans.
- Kùzu [19]: A graph query engine that indexes property graphs using unsorted adjacency lists. It uses a cost-based dynamic programming to produce plans that mix wco and pairwise joins. For the joins, hash tables are created on the fly from the unsorted lists.
- Graphflow [26]: The predecessor of Kùzu, which uses instead in-memory sorted adjacency lists.
- ADOPT [48]: The first wco algorithm using adaptive VEOs on LTJ. It uses exploratory search and reinforcement learning to find near-optimal orders, using actual execution times as feedback on the suitability of orders. We include variants using one and 70 threads.
- EmptyHeaded [1]: An implementation of a more general algorithm than LTJ, which applies a generalized hypertree decomposition [15] on the queries and uses a combination of wco algorithms [36] and Yannakakis' algorithm [51]. It offers worst-case time guarantees that are stronger than the AGM bound. Triples are stored in 6 tries (all orders) in main memory.

Those systems use too much memory on our Wikidata graph. For example, Graphflow stores one structure per predicate, which makes it usable with few predicates only: on a subset containing $< 10\%$ of our Wikidata graph [5], it failed to build even in a machine with 730 GB of Java heap space. ADOPT did not build correctly either. EmptyHeaded runs but it uses 1810 bpt, over 10 times more than Jena LTJ. DuckDB and Kùzu exceeded the main memory space at query time, even on a machine with 768 GB of RAM.

In this section we compare them over a smaller graph used in previous work [39], soc-LiveJournal1, the largest from the Stanford Large Network Dataset Collection [22], with $68{,}993{,}773$ *unlabeled* edges. We test different query shapes (see previous work for a detailed description [39]) including trees (1-tree, 2-tree, 2-comb), paths (3-path, 4-path), paths connecting cliques (2-3-lollipop, 3-4-lollipop), cliques (3-cliques, 4-cliques), and cycles (3-cycles, 4-cycles). We include 10 queries for each tree, path, and lollipop, and 1 for each clique and cycle. This is the same benchmark used for ADOPT [48], except that we do not force the clique and cycle variables to be different, and we choose for the constant any random value such that the query has occurrences. We set a 30-minute timeout and do not limit the number of results.

Since there are no labels, the Ring variants need not store the data for predicates, and the compact LTJ solutions store only two orders, PSO and POS (PS and POS with partial tries). Graphflow is tested on the cliques and cycles only because the implementation does not support constants in the BGPs.

Table 7 shows spaces and times. Interestingly, CLTJ∗ and UnCLTJ∗ get close to the space of the compressed Ring solutions, and xCLTJ∗ uses significantly less. Graphflow, ADOPT/Kùzu and EmptyHeaded/DuckDB use 2, 3, and over 4 times more space, respectively, than CLTJ∗. MillDB uses 5.5 and UmbraDB uses over 8 times more space.

The tree and path queries are solved in around a millisecond (and in many cases a tenth of a millisecond) by the CLTJ∗ variants. The largest version, UnCLTJ∗, is only 5%–15% faster than xCLTJ∗ (with only one exception where it is 45% faster) and almost twice as large. The next best systems (the larger Ring, ADOPT, and MillDB, all of them wco), are many times slower than the CLTJ∗ variants in all cases.

More in detail, the slower Ring is 7 to 130 times slower than xCLTJ∗, and the faster Ring is 2.5 to 11 times slower. DuckDB, the non-wco system, takes stably from half to one second for all these queries, which is 3–4 orders of mag-

System	Space	1-tree	2-tree	2-comb	3-path	4-path	2-3-lolli	3-4-lolli	3-clique	4-clique	3-cycle	4-cycle
Ring-small	5.52	0.071	0.030	54.6 E–4	90.7 E–5	0.037	5.89	535	timeout	timeout	timeout	timeout
Ring-large	7.59	28.6 E–5	66.4 E–4	15.2 E–4	30.1 E–5	0.010	1.61	135	timeout	timeout	timeout	timeout
xCLTJ∗	**3.42**	5.45 E–5	9.98 E–4	3.15 E–4	12.4 E–5	12.9 E–4	0.144	14.5	741	timeout	656	timeout
CLTJ∗	6.46	5.66 E–5	8.83 E–4	2.32 E–4	12.8 E–5	10.9 E–4	0.136	13.5	735	timeout	569	timeout
UnCLTJ∗	6.55	**4.70 E–5**	**8.78 E–4**	**2.17 E–4**	**9.64 E–5**	**10.0 E–4**	**0.134**	**12.9**	697	timeout	561	timeout
Graphflow	13.54								83.3	975	80.9	timeout
Kùzu	18.83	0.062	4, 61	0.170	0.137	timeout	8.88	out-mem	timeout	timeout	timeout	timeout
ADOPT-1	20.09	0.817	1.67	1.03	1.28	1.15	6.52	timeout	1337	timeout	885	timeout
ADOPT-70	20.09	0.837	1.75	1.21	1.12	1.56	3.60	105	105	timeout	106	timeout
EmptyHeaded	28.65	5.76 E–5	1.32	9.68 E–4	45.5 E–5	0.506	11.0	315	**14.0**	**326**	**13.1**	**1006**
DuckDB	31.13	0.609	0.750	0.640	0.595	0.699	0.948	out-mem	timeout	timeout	timeout	timeout
MillDB	35.30	131 E–5	98.8 E–4	32.0 E–4	148 E–5	0.015	0.357	38.0	timeout	timeout	1434	timeout
UmbraDB	52.86	0.028	0.085	0.046	0.041	0.058	0.240	102	416	timeout	timeout	timeout

Table 7: Space in bpt and median time in seconds (timeout is 1800) for various systems on graph soc-LiveJournal1.

nitude slower than xCLTJ∗. MillDB, the best wco system from previous experiments, is much faster than DuckDB, but still 1–2 orders of magnitude slower than xCLTJ∗. ADOPT, the remaining wco system, is 3–4 orders of magnitude slower than xCLTJ∗ in these queries (parallelization does not help in this case). The performance of beyond-wco systems is varied. The most stable one is UmbraDB, which takes 28–85 milliseconds on those queries, being 2–3 orders of magnitude slower than xCLTJ∗. Kùzu also fares well (though not as well as UmbraDB) on some queries, but much worse in others, being 3–4 orders of magnitude slower than xCLTJ∗. Only EmptyHeaded shows times comparable to those of xCLTJ∗ (at most 4 times slower) on the smaller queries, but it is still up to 3 orders of magnitude slower on the larger ones.

The lollipop shapes are harder to solve, but the CLTJ∗ variants still handle the larger one in less than 15 seconds, and are an order of magnitude faster than all the other alternatives, except UmbraDB, which is only 1.7 times slower than xCLTJ∗ on the smaller shape, and MillDB, which is about 2.5 times slower in both shapes. The next best performing systems are the large Ring and the parallel ADOPT.

EmptyHeaded finally takes over on the hardest shapes, cliques and cycles, where it is 3–6 times faster than Graphflow, 7–8 times faster than the parallel ADOPT, 30 times faster than UmbraDB, 50 times faster than the CLTJ∗ variants, and 110 times faster than MillDB, when those other systems do not timeout. We note that the CLTJ∗ variants are still faster than sequential ADOPT in these shapes.

5.6 The dynamic case

We now show how efficient are our update algorithms and, more importantly, how enabling dynamism affects the query time performance of our compact representations. We build the indices on a randomly chosen 80% of the Wikidata graph, and the remaining 20% is used for insertions. On those indices, we compute the 1,295 queries from the Wikidata query log in different scenarios, without limiting the number of results. The scenarios are set depending on the number of updates per query: from 1000 queries between each pair of updates to 1000 updates between each pair of queries. Each update can be an insertion or deletion of a triple; the type of operation and triple are chosen at random.

System	Insertion	Deletion
xCLTJ∗	11.42	11.45
MillDB	3.73	3.03
Virtuoso	22.85	14.30

Table 8: Average time in milliseconds for each type of update in different systems.

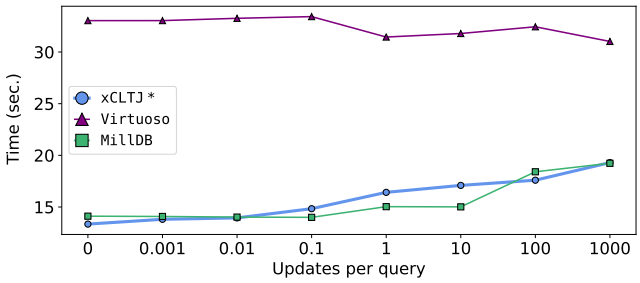


Fig. 7: Average time performance of some indices with different number of updates per query, without limiting the number of results ($x < 1$ means $1/x$ queries per update). The configuration with 0 updates is run on the static data structure; the others on the dynamic one.

Table 8 shows the individual insertion and deletion times for some systems, which are averaged from the updates performed under the scenario of 1000 updates per query. We compare xCLTJ∗ with MillDB and Virtuoso; the other systems have considerably higher update times. The update times of xCLTJ∗ are around 10 milliseconds, outperforming those of Virtuoso; MillDB is instead three times faster.

Fig. 7 illustrates the query time performance obtained when queries are mixed with updates. For xCLTJ∗, the ver-

sion without updates (i.e., the static case) yields an average time of 11.3 seconds per query. The performance worsens as the number of updates increases, but only mildly, up to 14.4 seconds with 1000 updates per query. Hence, even in highly dynamic scenarios, with huge amounts of updates, the average query times worsen by less than 30%. The times of Virtuoso, which uses fully dynamic data structures (not adaptive ones, like `xCLTJ`$*$), stay over 30 seconds per query. The times of MillDB, which also uses a fully dynamic data structure (a B+-tree), increase at about the same pace of those of `xCLTJ`$*$, probably due to progressively more fragmented B+-tree leaves (the structure built on 80% of the triples with bulk-loading produces full leaves).

We also measure the space per triple of the dynamic version after all the updates and queries are performed. Notice that the cases are not fully comparable because the represented triples are not the same. In general, the dynamic version uses around 10% more space than the static one. As the number of updates increases to 1000, however, this overhead rises to 18%. Even in such an extreme scenario, the space of `xCLTJ`$*$ on the whole Wikidata graph would increase to 33 bpt. The conclusions we have drawn over the static version remain essentially unaltered in the dynamic case.

6 A Complete System

There is a gap between a proof-of-concept research prototype and a system that can be readily used by practitioners. Although we do not aim for full-fledged software, we have invested in closing this gap.

6.1 String identifiers

An important aspect that prototypes neglect is that the node and label identifiers in RDF are strings (IRIs and literals). Assuming that both are integers in a contiguous range greatly simplifies developments and lets researchers focus on the most important aspects of complex query processing. A real system, however, must map the query strings into those integer identifiers, and the identifiers of the resulting triples into their corresponding strings. While translating the query strings to identifiers is not time-critical because they are short, the efficiency of translating the resulting triples is relevant in queries that output many results. From the systems we have compared with, MillDB, Jena LTJ, RDF-3X, Virtuoso, Blazegraph, UmbraDB, Kùzu, and DuckDB do handle strings, while the Ring, Graphflow, ADOPT, and EmptyHeaded do not.

Storing the string identifiers is challenging when we aim at compact representations, because the strings may add up to a significant size if stored in uncompressed form. In our dataset of Section 5.1, the strings in plain form occupy 12.4 GB, more than the triples in integer-encoded form. In a static scenario, it has been shown [5] that compressed string dictionaries [25] can represent those strings in just 3.68 additional bytes per triple. The translation of each output triple takes 7–14 microseconds (the effect would be hardly noticeable in Table 2, for example, as times would grow by 7–14 milliseconds). While their static dictionary can be plugged directly into *CompactLTJ*, the dynamic version of *CompactLTJ* requires a dynamic compact dictionary where strings can be inserted and deleted. This is challenging because most of the compact static dictionary representations [25] cannot be maintained upon updates.

Static dictionaries may store the strings in some convenient order (typically lexicographic) and let the string identifiers be their position in that order. This is not possible in a dynamic dictionary, as one would need to update a massive amount of identifiers in the collection when a string is inserted or deleted. For this reason, we store both the strings s_i and their identifiers k_i. Using Front Coding, we represent a dictionary $\mathcal{D} = \{\langle s_0, k_0\rangle, \langle s_1, k_1\rangle, \ldots, \langle s_n, k_n\rangle\}$, listed in lexicographic order, as follows:

$$FC[0] = k_0 \cdot s_0,$$
$$FC[i] = k_1 \cdot \texttt{lcp}(s_{i-1}, s_i) \cdot \texttt{suffix}(s_i, \texttt{lcp}(s_{i-1}, s_i)),$$

where $\cdot$ represents byte concatenation, $\texttt{lcp}(s_i, s_j)$ is the longest common prefix between strings s_i and s_j, and $\texttt{suffix}(s_i, j)$ is the suffix of s_i starting after position j. The integers for the identifiers k_i and `lcp` are encoded with VByte [50], while `suffix` is stored in plain form.

To retrieve the string s_i we need to read all the previous ones. To reduce that cost, we split $\mathcal{D}$ into *buckets* of maximum length $\sigma_{\max}$, which is then the maximum cost to retrieve a string by starting from the first string s_0 of its bucket (note that, to scan a bucket, we must iteratively obtain all its strings s_i, but the time spent is $\sigma_{\max} \leq \sum_i |s_i|$).

To easily find the identifier of a string $s \in \mathcal{D}$, a binary search tree is maintained with one bucket per leaf, in left-to-right order. Each internal node stores a pointer to the bucket stored at the leftmost leaf of its right child; note that the string s_0 of the bucket is stored in plain form. By comparing s with the strings s_0 of the pointed buckets, we descend from the root of the tree to the leaf storing the bucket where s should be stored, and then scan the bucket. The total search time to obtain the identifier k of s (or determine that $s \notin \mathcal{D}$) is then $O(|s| \lg n + \sigma_{\max})$.

The most pressing operation on dictionaries, however, is the opposite: mapping an identifier k to its corresponding string s (this has to be done for each result). To optimize that operation, we store an array $A[1\,..\,n]$, where $A[k]$ stores a pointer to the bucket that contains $\langle s, k\rangle$. In that way, s is reconstructed by just scanning one bucket until finding the identifier $k_i = k$, and then retrieving s_i, in time $O(\sigma_{\max})$.

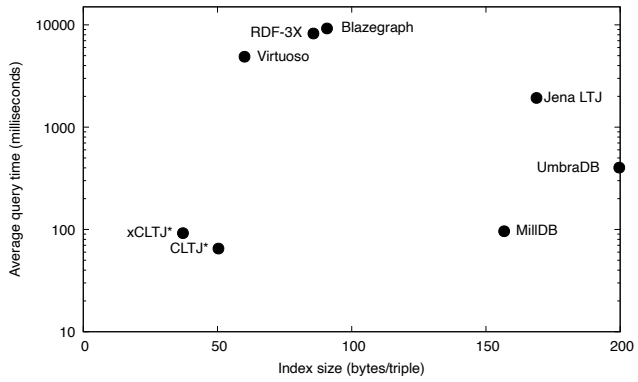


Fig. 8: Space/time trade-off in complete systems handling strings, limiting results to 1000. To emulate a realistic dynamic scenario for the CLTJ variants, we use their query times when there are 0.1 updates per query, as in Wikidata.

The dictionary supports string deletions and insertions. Deleting the string s with identifier k implies removing its pair $\langle s_i, k_i\rangle = \langle s, k\rangle$ from the bucket pointed to by $A[k]$. After the removal we must reencode s_{i+1}, now with respect to s_{i-1}. The bucket is essentially rewritten, in time $O(\sigma_{\max})$. The entry $A[k]$ is now obsolete and the identifier k is free to be used by another insertion. A linked list of those free identifiers is maintained using the same obsolete entries $A[k]$ to store the "next" pointers. Hence, after the removal of k, $A[k]$ becomes the first element of that list. For simplicity we do not enforce policies of minimum bucket sizes, except that we remove the leaf corresponding to an empty bucket (and remove an internal node too). Deletion costs $O(\sigma_{\max})$ time.

To insert a new string s we first find an identifier k for it. This is the first element of the linked list (which is then removed from the list), or the value $n+1$ if the list is empty (and now $n+1$ is the number of strings). We use the binary tree to find the bucket where to insert s, and make $A[k]$ point to it. We scan the bucket up to the insertion point, and rewrite it from there. Concretely, if s falls between s_{i-1} and s_i, we insert $\langle s, k\rangle$ after s_{i-1}, encoding s with respect to s_{i-1}, and reencode s_i with respect to s. In case the bucket overflows, we split it by half and create a new internal node as the parent of both halves. The insertion cost is $O(|s| \lg n + \sigma_{\max})$.

In our experimental setup, the dynamic dictionary adds an extra space of 6.81 bpt, and transforms the identifiers of each output triple to strings in around 8.6 microseconds.

Fig. 8 compares the systems that handle strings, measuring query times of the CLTJ variants under a realistic dynamic scenario of 0.1 updates per query, as in Wikidata (recall Section 4). The space usage of `xCLTJ*` rises to 37.13 bpt, which is still about $2/3$ of the space used by Virtuoso, the smallest system handling strings, and 4 times less than the space of MillDB. It is still 53 times faster than Virtuoso and 5% faster than MillDB. Our slightly larger alternative, `CLTJ*`, now uses 53.51 bpt, which is still 19% less than Virtuoso and 3 times less than MillDB. In exchange, it is 40% faster than `xCLTJ*`, 75 times faster than Virtuoso, and 48% faster than MillDB.

6.2 Public software

The code is publicly available at this Github repository: `https://github.com/adriangbrandon/cltj`. The software includes the benchmarks of our experimental evaluation. For each query, the benchmark outputs the identifier of the query, the number of results, and the elapsed time in nanoseconds. All of this information is limited by a semicolon. We provide the bash scripts utilized to build the indices and execute the benchmark queries.

In addition, we developed an API to simplify the usage of our software. To demonstrate how to use the API there are one example of a command line where you can interact directly with its interface, and others that show how to use the API from `C++` code. More information about how to use the API is available in the `Readme.md` file of the repository. The datasets used in the experiments are available at `https://zenodo.org/records/15117967`.

7 Conclusions

We have shown that it is possible to implement the Leapfrog Triejoin (LTJ) algorithm, which solves Basic Graph Patterns on graph databases in worst-case-optimal (wco) time, within affordable space usage and without giving up on time performance. Precisely, we introduced a representation we call *CompactLTJ*, which uses one bit per trie edge instead of one pointer, while supporting trie navigation functionality in time similar to a classic pointer-based representation. Further, we implemented trie switching, which allows us store partial tries that retain the same *CompactLTJ* functionality and good performance, while slashing its space usage.

The fastest classic LTJ implementation we are aware of, MillenniumDB [47], uses about 14 times the space needed to represent the graph triples in plain form (i.e., each as three 32-bit integers). Our smallest *CompactLTJ* variant reduces this factor to 2.3—a 5.5-fold space reduction—while retaining MillenniumDB's time performance, and surpassing it in many cases. Other classic representations, many of which are non-wco, use 2–3 times the space used by *CompactLTJ* and are 30–40 times slower.

We also implemented a dynamic version of *CompactLTJ*, which enables efficient insertion and deletion of triples with little overhead on top of the space and time of the static implementation. This breaks a long-standing limitation of compact representations and puts *CompactLTJ* on par with the functionality needed in real scenarios.

These results can change the landscape of indices for graph databases, as they show that it is feasible to implement the wco LTJ algorithm in memory within little space—much less than what is used by popular non-wco systems, in both static and dynamic scenarios. We have also explored some techniques—adaptive variable elimination orders and new predictors of the cost of choosing a variable—that make *CompactLTJ* considerably more robust on the bad cases of the standard solution.

Finally, we enabled *CompactLTJ* to handle string identifiers, not just integers, which is what most RDF systems can handle. We left a public implementation that allows researchers and practitioners use our best static and dynamic variants, handling integer or string identifiers, under various convenient modes of operation. We expect this to be useful for benchmarking, for research, and for using *CompactLTJ* on actual deployments.

7.1 Future work

Our experiments showed that more sophisticated "beyond-wco" indices, like Graphflow [26], ADOPT [48], and EmptyHeaded [1], were faster than *CompactLTJ* on some query shapes that are very hard to handle. A promising future work direction is to implement those query strategies on top of compact data structures, which could lead to even stronger indices that are space-affordable (recall that those stronger systems exceeded a generous amount of main memory when run over our dataset, so reducing their space usage is very valuable). An analogous challenge is to use compact data structures to represent factorized databases [40], using wco or non-wco strategies.

We remark that *CompactLTJ* runs in main memory and would not be disk-friendly. While its compactness makes it 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.

Finally, we plan to extend *CompactLTJ* to handle property graphs, where nodes have attributes of various types, with their own query operators. Extending the supported functionality to regular path queries (RPQs) is another significant step. While this has already been done using compact data structures [6, 7, 32], the integration with BGPs into conjunctive RPQs (CRPQs) is a significant challenge.

Funding
Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile. A.G. is funded in part by MCIN/AEI/10.13039/5011000-11033: grant PID2020-114635RB-I00 (EXTRACompact); by MCIN/AEI/10.13039/501100011033 and "Next-GenerationEU"/ PRTR: grant TED2021-129245B-C21 (PLAGEMIS); by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe": PID2022-141027NB-C21 (EARTHDL); and CITIC receives subsidies from Xunta de Galicia and FEDER Galicia (Ref. ED431G 2023/01). G.N. is funded in part by Fondecyt Grant 1-230755, Chile.

Competing interests
The authors declare no competing interests.

References

1. C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems*, 42(4), 2017.
2. M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*, pages 429–444, 2017.
3. D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):article 8, 2022.
4. D. Arroyuelo, D. Campos, A. Gómez-Brandón, G. Navarro, C. Rojas, and D. Vrgoc. Space & time efficient leapfrog triejoin. In *Proc. 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, page article 2, 2024.
5. D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, 29(2):article 5, 2024.
6. D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Optimizing RPQs over a compact graph representation. *The Very Large Databases Journal*, 33:349–374, 2024.
7. D. Arroyuelo, A. Gómez-Brandón, and G. Navarro. Evaluating regular path queries on compressed adjacency matrices. *The Very Large Databases Journal*, 34:article 2, 2025.
8. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
9. J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Alg.*, 4(1):1–18, 2008.
10. A. Bonifati, W. Martens, and T. Timm. Navigating the maze of Wikidata query logs. In *Proc. World Wide Web Conference (WWW)*, pages 127–138, 2019.
11. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
12. S. Dönges, S. Puglisi, and R. Raman. On dynamic bitvector implementations. In *Proc. 32nd Data Compression Conference (DCC)*, pages 252–261, 2022.
13. O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *Data Engineering Bulletin*, 35(1):3–8, 2012.
14. M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.
15. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 21–32, 1999.
16. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, 2013. `https://www.w3.org/TR/sparql11-query/`.

17. A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019.
18. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
19. G. Jin, X. Feng, Z. Chen, C. Liu, and S. Salihoglu. Kùzu graph database management system. In *Proc. 13th Conference on Innovative Data Systems Research (CIDR)*, 2023.
20. O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *Proc. 20th International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.
21. M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems*, 41(4):22, 2016.
22. J. Leskovec. Stanford Large Network Dataset Collection: LiveJournal social network. https://snap.stanford.edu/data/soc-LiveJournal1.html.
23. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*, pages 376–394, 2018.
24. F. Manola and E. Miller. *RDF Primer*. W3C Recommendation. 2004. `http://www.w3.org/TR/rdf-primer/`.
25. M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016.
26. A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment*, 12(11):1692–1704, 2019.
27. A. Mhedhbi, A. Deshpande, and S. Salihoglu. Modern techniques for querying graph-structured databases. *Foundations and Trends in Databases*, 14(2):72–185, 2024.
28. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
29. G. Navarro. *Compact Data Structures — A practical approach*. Cambridge Univ. Press, 2016.
30. G. Navarro. Adaptive dynamic bitvectors. In *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 204–217, 2024. Journal version to appear in `https://doi.org/10.1002/spe.3433`.
31. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
32. G. Navarro and J. Robert. Compressed graph representations for evaluating regular path queries. In *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–232, 2024.
33. T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *Proc. 10th Conference on Innovative Data Systems Research, (CIDR)*, 2020.
34. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19:91–113, 2010.
35. H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th Symposium on Principles of Database Systems (PODS)*, pages 111–124, 2018.
36. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proc. 31st Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.
37. H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4): 5–16, 2013.
38. H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra. Beyond worst-case analysis for joins with Minesweeper. In *Proc. 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 234–245, 2014.
39. D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 2:1–2:8, 2015.
40. D. Olteanu and M. Schleich. Factorized databases. *SIGMOD Record*, 45(2):5–16, 2016.
41. N. Prezza. A framework of dynamic data structures for string processing. In *Proc. 16th International Symposium on Experimental Algorithms (SEA)*, pages 11:1–11:15, 2017.
42. M. Raasveldt and H. Mühleisen. DuckDB: An embeddable analytical database. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 1981–1984, 2019.
43. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 23–34, 1979.
44. B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata®RDF Graph Database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.
45. T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.
46. D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.
47. D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil-Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. MillenniumDB: An open-source graph database system. *Data Intelligence*, 5(3):560–610, 2023.
48. J. Wang, I. Trummer, A. Kara, and D. Olteanu. ADOPT: Adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning. *Proceedings of the VLDB Endowment*, 16(11):2805–2817, 2023.
49. Y. R. Wang, M. Willsey, and D. Suciu. Free Join: Unifying worst-case optimal and traditional joins. *Proc. 49th ACM International Conference on Management of Data (SIGMOD)*, 1(2): 150:1–150:23, 2023.
50. H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.
51. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*, pages 82–94, 1981.

A Constant-time Select on Bitvectors

For self-containedness, we describe how operation select on a bitvector $B[1\,..\,n]$ is implemented in constant time using $o(n)$ additional bits of space [11, 28]. The reader can also consult surveys [31, Sec. 6.1] or books [29, Sec. 4.3.3].

In principle, we could offer constant-time select by precomputing and storing all the values $S[j] = \mathsf{select}(B, j)$ in an array $S[1\,..\,r]$, where r is the number of 1s in B. The total required space, $r \lg n$ bits, can be way more than the desired $o(n)$, however.

Instead, we logically divide B into *blocks* of b 1s, for some $b = \Theta(\lg^2 n)$. Given a query $\mathsf{select}(B, j)$, we know that the argument j belongs to the block number $\lceil j/b \rceil$. Note that blocks may span areas of different lengths in B. We say that a block is *long* if it spans more than $\lg^4 n$ bits in B, and *short* otherwise.

Since there are at most $n/\lg^4 n$ long blocks in B, we can afford storing all the select answers that fall in long blocks, using $b \lg n = O(\lg^3 n)$ bits per block. This yields a total of $O(\lg^3 n) \cdot n/\lg^4 n = O(n/\lg n) = o(n)$ bits of space. Queries that fall in long blocks are then solved in $O(1)$ time.

On the other hand, there can be many short blocks, so the same trick cannot be used. Instead, representing a position inside them requires only $\lg(\lg^4 n) = O(\lg \lg n)$ bits. We repeat the idea inside short blocks: we logically divide them into *miniblocks* containing m 1s, for some $m = \Theta((\lg \lg n)^2)$. We say that miniblocks are *long* if they span more than $(\lg \lg n)^4$ bits in B, and *short* otherwise.

Analogously to long blocks, we can afford to store all the answers to select queries inside long miniblocks, as there are in total $n/(\lg \lg n)^4$ of them and the answers inside each miniblock require $m \cdot O(\lg \lg n) = O((\lg \lg n)^3)$ bits of space, which yields a total of $O(n/\lg \lg n) = o(n)$ bits. We thus also answer in $O(1)$ time the queries that fall in long miniblocks of short blocks.

We are left with the short miniblocks inside short blocks. Since these are of length $O((\lg \lg n)^4) \subseteq O(\lg n)$, select queries inside them can be solved by scanning their bits sequentially. This is done by reading chunks of $(\lg n)/2$ bits and using the chunk as a number to address a table T that, for every possible chunk, tells how many 1s are there, and their positions, in the chunk. Table T has $2^{(\lg n)/2} = \sqrt{n}$ entries, each using $O(\lg n \lg \lg n)$ bits, which is $o(n)$ in total. We then process the $O(\lg n)$ bits of short miniblocks with $O(1)$ accesses to T, which also yields constant time.

A.1 Solving selectnext

To solve $\mathsf{selectnext}(B, i)$, we cut B into blocks of *fixed* length $b = \Theta(\lg^2 n)$. Each block stores the position of the first 1 after it, which requires $O(n/\lg n)$ bits of space.

Blocks are cut into miniblocks of fixed length $m = \Theta(\lg n)$. Each miniblock stores the position of the first 1 after it, or zero if the position falls after the block to which the miniblock belongs. Since such a position can be stored in $O(\lg \lg n)$ bits, the total space is $O(\lg \lg n \cdot n/\lg n) = o(n)$ bits.

To compute $\mathsf{selectnext}(B, i)$ in $O(1)$ time, we first scan the $O(\lg n)$ bits that follow i inside its miniblock, in constant time using a table analogous to T where the first 1 in each possible chunk is precomputed. If a 1 following i is not found inside the miniblock of i, we access the position stored in the miniblock. If the position is zero, we access the position stored in the block. This completes the process.