Space & Time Efficient Leapfrog Triejoin

Diego Arroyuelo
DCC, PUC & IMFD
Santiago, Chile

Daniela Campos
DCC, University of Chile
Santiago, Chile
dcampos@dcc.uchile.cl

Adrián Gómez-Brandón
Universidade da Coruña & IMFD
A Coruña, Spain
adrian.gbrandon@udc.es

Gonzalo Navarro
DCC, University of Chile & IMFD
Santiago, Chile
gnavarro@dcc.uchile.cl

Carlos Rojas
IMFD
Santiago, Chile
cirojas6@uc.cl

Domagoj Vrgoč
IMC, PUC Chile & IMFD
Santiago, Chile
dvrgoc@ing.puc.cl

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, and their corresponding algorithms be non-wco. In this paper we show that, by using compact data structures, it is possible to build an index that at the same matches the query time performance of the fastest classic wco index, and uses half 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. The resulting structure, called *compactLTJ*, uses 25% of the space of classic wco implementations and 45%–65% of classic non-wco systems. At solving queries, it is on par with the fastest classic wco system, and two orders of magnitude faster than non-wco systems. We further incorporate improved query resolution strategies into *compactLTJ*, which makes it considerably faster than any other alternative to display the first query results.

CCS Concepts

• **Theory of computation → Database query processing and optimization (theory)**; **Data structures and algorithms for data management**.

Keywords

Worst-case optimal joins; Leapfrog Triejoin; graph patterns; graph databases; graph indexing; similarity joins; nearest-neighbor graphs

ACM Reference Format:
Diego Arroyuelo, Daniela Campos, Adrián Gómez-Brandón, Gonzalo Navarro, Carlos Rojas, and Domagoj Vrgoč. 2018. Space & Time Efficient Leapfrog Triejoin. In *Proceedings of ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

SIGMOD '24, June 9–15, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Natural joins are fundamental in the relational algebra, and generally the most costly operations. A bad implementation choice can lead to unaffordable query times, so they have been a concern since the beginnings of the relational model. Apart from efficient algorithms to join two tables (i.e., solve pair-wise joins), database management systems sought optimized strategies (e.g., [27]) to solve joins between several tables (i.e., multijoins), as differences between good and bad plans could be huge in terms of efficiency. A *query plan* for a multijoin was a binary expression tree where the leaves were the tables to join and the internal nodes were the pair-wise joins to perform.

After half a century of revolving around this pairwise-join-based strategy, it was found that it had no chance to be optimal [4], as it could generate intermediate results (at internal nodes of the expression tree) that were much larger than the final output. The concept of a *worst-case optimal (wco)* algorithm [4] was coined to define a multijoin algorithm taking 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). Several wco join algorithms were proposed since then [14, 23–26, 29].

Leapfrog Triejoin (LTJ) [29] 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 classical 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 [11, 29]. 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.

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 [10] 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 [11]. This is very relevant because typical BGPs correspond to large and complex multijoins [1, 11, 13, 26], 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 implementation.

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 [11] doubles the space of the original non-wco version. Efficient wco implementations like Jena LTJ [11] and MillenniumDB [31] use around 14 times the space required to store the graph triples in raw form. The most popular systems for graph databases, like Jena [11], Virtuoso [7], RDF-3X [22], or Blazegraph [28], 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—just 3.3 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 [12], in a way that requires only one *bit*, instead of one *pointer*, per trie edge. The resulting structure, which we call *compactLTJ*, uses about 25% of the space of classic LTJ implementations that store the 6 tries (MillenniumDB, Jena LTJ), and 45%–65% 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 two orders of magnitude.
(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 the first million results faster than the traditional global-order strategy, making the *compactLTJ* index, for example, twice as fast as MillenniumDB to obtain the first 1000 results.

Recent research [3] 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 also two orders of magnitude faster than these compressed data structures. We also show that other recent indices that offer beyond-wco query time guarantees, like Graphflow [18], ADOPT [32], and EmptyHeaded [1], do outperform *compactLTJ* on particularly difficult queries, but again use 2–4 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.

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 [17], an *edge-labeled graph* is a finite set of *triples* $G \subseteq \mathcal{U}^3$, where each triple $(s, p, o) \in \mathcal{U}^3$ encodes the directed edge $s \xrightarrow{p} o$ from vertex s to vertex o, with edge label p. We call $\text{dom}(G) = \{s, p, o \mid (s, p, o) \in G\}$ 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 $U = \max \text{dom}(G)$. 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 $\mathcal{U} = [1 \,..\, U]$.

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

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

2.2 Worst-case optimal joins

2.2.1 The AGM bound A well-established bound to analyze join algorithms is the *AGM bound*, introduced by Atserias et al. [4], 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 [11].

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

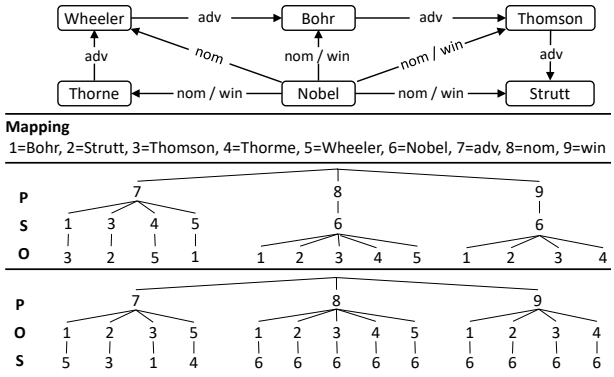

Figure 1: A labeled graph G' with its string to integer mapping and tries for orders PSO and POS.

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

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

2.2.2 Leapfrog TrieJoin (LTJ) We describe the Leapfrog Triejoin algorithm [29], originally designed for natural joins in relational databases, as it is adapted for BGP matching on labeled graphs [11].

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

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

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

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

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

2.3 Variable Elimination Orders (VEOs)

Veldhuizen [29] 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 [11, 29]. 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 [3, 11, 31] computes, for each variable x_j, its minimum weight

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

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

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

In this paper we explore the use of *adaptive* VEOs, which are defined progressively as the query processing advances, and may differ for each different binding of the preceding variables. ADOPT [32] is the first system combining LTJ with adaptive VEOs. The next variables to bind are chosen using reinforcement learning, by partially exploring possibly upcoming orders, and balancing the cost of exploring with that of the obtained improvements. Our adaptive VEOs will be computed, instead, simply as a variant of the formula presented above for global VEOs [11].

Other systems go even further in this beyond-wco path. Building on the well-known Yannakakis' instance-optimal algorithm for acyclic queries [34], EmptyHeaded [1] applies a so-called Generalized Hypertree Decomposition [9], which decomposes cyclic queries into a tree where the nodes are cyclic components, so as

to solve the nodes using a wco algorithm [24] and then apply Yannakakis' algorithm on the resulting acyclic query on the intermediate results. Graphflow [18], Umbra [8], and Free Join [33] 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 [14] and Panda [2] also go beyond wco.

3 CompactLTJ: LTJ 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.

3.1 Trie representation

The Level-Order Unary Degree Sequence (LOUDS) [12] 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.

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: we spend exactly *one bit* per trie edge, that is, n bits for a trie of n nodes, halving the original space [12].

Our encoding also simplifies the traversal compared to the original LOUDS [12]. We will use the position preceding the encoding of a node as its trie identifier $v \geq 0$. The identifier of the ith child of v, for $i \geq 1$, is $\mathsf{child}(v, i) = \mathsf{select}_1(T, v + i)$, which is the position of the $(v + i)$th occurrence of bit 1 in T. Operation select can be supported in $O(1)$ time using just (n) additional bits of space on top of T [6, 19].

The formula works because T simultaneously enumerates, in levelwise order, the trie edges (one bit per edge) and their target nodes (one 1-terminated encoding per node, leaves omitted). The number of children of v can also be computed in $O(1)$ time as $\mathsf{degree}(v) = \mathsf{selectnext}_1(T, v + 1) - v$, where $\mathsf{selectnext}_1(T, v + 1)$ is the position of the leftmost occurrence of 1 in $T[b + 1 . .]$ and can also be computed in $O(1)$ time using $o(n)$ additional bits of space.

The edge labels are stored in a compact array L, each label using $\lceil \lg U \rceil$ bits. The labels in L are deployed in the same levelwise order of the edges T, so the labels corresponding to the children of node v are all consecutive, in $L[v + 1 . . v + \mathsf{degree}(v)]$. This allows implementing $\mathsf{leap}()$ efficiently using exponential search from the current position.

3.2 Indices

Our main index, *compactLTJ*, is exactly as described above, comprising T and L. For the trie PSO in Fig. 1, it would store

$$T = 001\ 000111\ 1111000010001$$
$$L = 789\ 134566\ 3251123451234$$

where, for example, the second ($i = 2$) child of the root ($v = 0$) descends by $L[0 + 2] = 8$ and leads to $u = \mathsf{child}(0, 2) = \mathsf{select}_1(T, 0 + 2) = 7$. The encoding of $u = 7$ is at $T[u + 1 . . u + \mathsf{degree}(u)] = T[8 . . 8] = 1$. Its only child, by $L[7{+}1] = 6$, leads to $w = \mathsf{child}(7, 1) = \mathsf{select}_1(T, 7{+}1) = 13$. Node $w = 13$ then has $\mathsf{degree}(13) = \mathsf{selectnext}(14) - 13 = 5$ children, at $L[14 . . 18] = 12345$.

3.3 UnCompactLTJ

We also introduce a version called *unCompactLTJ*, which is a minimal non-compact trie representation. The *unCompactLTJ* index stores an array $P[0 . .]$ of *pointers*, one per internal node, deployed in the same order of LOUDS. Pointers are positions in the array using $\lceil \lg n \rceil$ bits. Each internal node v stores in $P[v]$ a pointer to its first child, knowing that the others are consecutive. Its number of children is simply $P[v + 1] - P[v]$. Its array L of edge labels is identical to that of *compactLTJ*. For our example above we have $P = \langle 1, 4, 8, 9, 10, 11, 12, 13, 14, 19, 23 \rangle$ (23 is a terminator).

In exchange for nearly doubling the space of *compactLTJ*, *unCompactLTJ* has explicit pointers just like classical data structures, so it does not spend time in computing addresses. As we show in the experiments, *unCompactLTJ* still uses half the space of Jena LTJ [11], a classic index that supports LTJ using the six tries (implemented as B+-trees).

4 Improved Variable Elimination Orders

Our second 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 use of the w_{ij} estimator.

4.1 Adaptive VEOs

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

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

In the adaptive setting, we do not check anymore that the new variable shares a triple with a previously eliminated one; this aimed to capture the fact that those triples would be more selective when some of their positions were bound, but now we know exactly the size of those progressively bound triples. The lonely variables are still processed at the end.

4.2 Computing the 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 $w_{ij} = \text{degree}(v)$. We now define an alternative version, *compactLTJ**, which computes w_{ij} as the number of leaf descendants of v. This is computed as $w_{i,j} = n$ if v is in the first level, and $w_{i,j} = \text{degree}(v)$ if v is in the third level. For the second level, we compute in constant time $w_{ij} = \text{child}(v + \text{degree}(v), 1) - \text{child}(v, 1)$.

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.

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, and 378 GB RAM.

5.1 Datasets and queries

We run two benchmarks over the Wikidata graph [30], which we choose for its scale, diversity, prominence, data model (i.e., labeled edges) and real-world query logs [5, 16]. The 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 [16]. In search of challenging examples, we downloaded queries that gave timeouts, and selected queries with a single BGP, obtaining 1,295 unique queries. Those are classified into three categories: (I) 520 BGPs formed by a single triple pattern, which mostly measure the retrieval performance of the index; (II) 580 BGPs with more than one triple but only one variable appearing in more than one triple, which measure the performance of joins but do not distinguish good from bad VEOs (as long as the join variable is eliminated first, of course); (III) 195 complex BGPs, where the performance of different VEOs can be compared.

All queries are run with a timeout of 10 minutes and a limit of 1000 results (as originally proposed for WGPB [11]). 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.

5.2 Compact LTJ variants

Table 1 compares the indices *compactLTJ*, *compactLTJ**, and *unCompactLTJ* described in Section 3), calling them respectively `CLTJ`, `CLTJ*`, and `UnCLTJ`. All of them compute the VEO in traditional ("global VEO") and in adaptive form (Section 4.1). No variant gave any timeout.

The space of the `CLTJ` index is just 3.4 times the size of the raw data encoded as a set of n 32-bit triples, whereas `UnCLTJ` uses 4.8 times the size (i.e., 40% more than `CLTJ`). The reward for using that 40% extra space is not significant, which shows that the space reduction obtained with `CLTJ` comes at essentially no loss in time performance.

System	Space (bpt)	Average (msec) Global	Average (msec) Adaptive	Median (msec) Global	Median (msec) Adaptive
`CLTJ`	40.90	381	331	0.5	0.5
`CLTJ*`	40.90	175	43	0.5	0.5
`UnCLTJ`	57.66	347	327	0.5	0.5
`UnCLTJ*`	57.66	157	40	0.5	0.5

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

While the medians of all the different variants are similar, half a millisecond 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 4.2) reduces the average query times by almost an order of magnitude, to around 40 milliseconds. Using adaptive VEOs alone produces a very modest improvement, and using the modified VEO predictor with global VEOs only halves the time.

In the sequel we will use only the variants `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:

- Ring [3], 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 [31]: 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 [11]: An implementation of LTJ on top of Apache Jena TDB. All six different orders on triples are indexed in B+-trees, so the search algorithm is always wco.
- RDF-3X [22]: Indexes a single table of triples in a compressed clustered B+-tree. The triples are sorted and those in each tree leaf are differentially encoded. RDF-3X handles triple patterns by scanning ranges of triples and features a query optimizer using pair-wise joins.
- Virtuoso [7]: 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 [28]: The graph database system hosting the official Wikidata Query Service [16]. We run the system in

System	Space (bpt)	Average (msec)	Median (msec)	Timeouts (> 10 min)
Ring-small	7.30	3056	24	5
Ring-large	12.15	2256	8	3
CLTJ∗	40.90	43	0.5	0
UnCLTJ∗	57.66	40	0.5	0
MillDB	156.78	96	27	0
Jena LTJ	168.84	1930	162	1
Virtuoso	60.07	4880	50	8
RDF-3X	85.73	8230	126	13
Blazegraph	90.79	9220	54	14

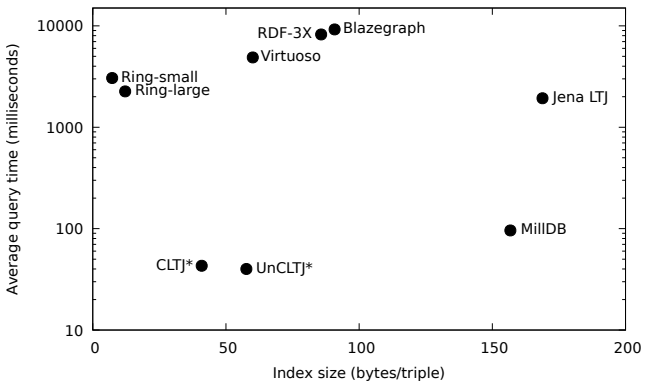

Table 2: Space and query times of various systems, limiting results to 1000. The plot shows the space/time tradeoff.

triples mode, with B+-trees indexing orders SPO, POS, and OSP. It supports nested-loop joins and hash joins.

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

We exclude Graphflow [18], ADOPT [32], and EmptyHeaded [1] because we have not enough memory to build them. Section 5.5 compares them on a smaller graph.

Table 2 shows the resulting time, space, and timeouts. A first observation is that, while the Ring variants use considerably less space than CLTJ (3.4–5.5 times less space, even less than the raw data), this comes at a considerable price in time performance: the Ring variants are 2 orders of magnitude slower than CLTJ∗ on average, and 1–2 in the median. While its small space can be crucial to operate in main memory where other representations do not fit, CLTJ∗ is a much faster alternative when it fits in main memory. Interesting, CLTJ∗ and UnCLTJ∗ are faster than classic 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 non-wco classic systems are somewhat smaller—47% to 126% larger than CLTJ∗—but two orders of magnitude slower.

Table 3 shows how the times distribute across the three query types, for the best systems. It is interesting that MillDB is much

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
CLTJ∗	40.90	1.8	0.2	12	0.8	243	2.8
UnCLTJ∗	57.66	1.8	0.3	12	0.8	228	2.4
MillDB	156.78	50	20	79	27	267	73

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

System	Space (bpt)	Average Gl	Average Ad	Median Gl	Median Ad	Timeouts Gl	Timeouts Ad
CLTJ	40.90	12.2	13.2	0.05	0.06	15	17
CLTJ∗	40.90	14.7	13.4	0.06	0.06	17	17
UnCLTJ	57.66	12.4	13.0	0.05	0.06	15	17
UnCLTJ∗	57.66	14.5	13.2	0.06	0.06	17	17

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.

slower than CLTJ∗ and UnCLTJ∗ 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 (yet CLTJ∗ is still faster). This suggests that MillDB is not intrinsically slower, but rather performs some internal setup per query that requires several tens of milliseconds. We return to this point next.

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. Indeed, our times limited to 1000 results suggest that adaptive VEOs produce results *sooner* along the query process than global VEOs. To confirm this intuition, Fig. 2 shows the time queries take until they deliver a certain number of results, for CLTJ∗ with global and adaptive VEOs. As it can be seen, the adaptive variant is much faster to deliver the first few million results, but the global VEO takes over—by a slight margin—since then.

Tables 5 and 6 show the results of the best performing variants, globally and by query type. MillDB fares better than with the limit, becoming similar to CLTJ∗/UnCLTJ∗ 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 type III, where the query plan matters most, CLTJ∗ and UnCLTJ∗ are slightly faster.

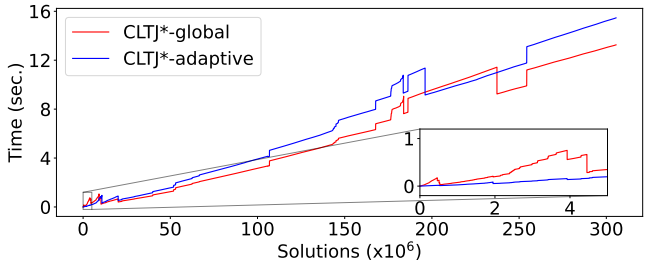

Figure 2: Average time per query until it returns a given number of solutions, for both variants of CLTJ∗. We measure type III queries, as long as there are at least 10 active queries to average; the curve is not always increasing because queries disappear from the set once they deliver all their results.

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
CLTJ∗	40.90	13.4	0.06	17
UnCLTJ∗	57.66	13.2	0.06	17
MillDB	156.78	12.0	0.05	16

Table 5: Space and query times of the best systems, 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
CLTJ∗	40.90	1.2	0.001	14.0	0.20	44.4	0.60
UnCLTJ∗	57.66	1.1	0.001	13.8	0.19	43.4	0.59
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 given in seconds.

5.5 Beyond wco systems

The alternative systems we have compared either are not wco, or use the basic LTJ with some global VEO. In this section we compare our new compact indices, with their improved query resolution strategies, against systems that use more sophisticated ones:

- Graphflow [18]: A graph query engine that indexes property graphs using in-memory sorted adjacency lists and supports hybrid plans blending wco and pairwise joins.
- ADOPT [32]: 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 [9] on the queries and uses a combination of wco algorithms [24] and Yannakakis' algorithm [34]. 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 [3], 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.

In this section we compare them over an even smaller graph used in previous work [26], soc-LiveJournal1, the largest from the Stanford Large Network Dataset Collection [15], with 68,993,773 *unlabeled* edges. We test different query shapes (see previous work for a detailed description [26]) 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 [32], 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. 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. Graphflow, ADOPT and EmptyHeaded use 2, 3, and over 4 times more space, respectively. The tree and path queries are solved in microseconds by CLTJ∗/UnCLTJ∗, while the slowest Ring is up to 25–50 times slower. ADOPT is 4–5 orders of magnitude slower in these queries (parallelization does not help in this case). EmptyHeaded is from twice as slow to 3–4 orders of magnitude slower.

The lollipop shapes are harder, but CLTJ∗/UnCLTJ∗ still handle them in at most 10 seconds, being 1–2 orders of magnitude faster than ADOPT and EmptyHeaded. The parallel ADOPT is 3 times faster than EmptyHeaded in these shapes.

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, and 35–40 times faster than CLTJ∗/UnCLTJ∗. We note that the latter are still twice as fast as sequential ADOPT.

6 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.

The fastest classic LTJ implementation we are aware of, MillenniumDB [31], uses about 14 times the space needed to represent the

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	71.0 E−4	296 E−4	54.6 E−4	90.7 E−5	365 E−4	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	101 E−4	1.61	135	timeout	timeout	timeout	timeout
CLTJ∗	6.46	4.57 E−5	**6.44 E−4**	**1.46 E−4**	8.89 E−5	7.14 E−4	**0.116**	**10.4**	565	timeout	457	timeout
UnCLTJ∗	6.55	**2.83 E−5**	6.49 E−4	1.50 E−4	**5.11 E−5**	**6.76 E−4**	**0.113**	**10.2**	558	timeout	452	timeout
Graphflow	13.54								83.3	975	80.9	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**

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

graph triples in plain form (i.e., each as three 32-bit integers). Our *compactLTJ* reduces this factor to 3.3—a four-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 1.5 to 2.3 times the space used by *compactLTJ* and are two orders of magnitude slower.

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 reasonable space—less than what is used by popular non-wco systems. We have also explored some techniques—adaptive variable elimination orders and new predictors of the cost of choosing a variable—that speed up *compactLTJ* considerably for retrieving the first million results. This is relevant in applications that are interactive or where obtaining some results suffices.

More sophisticated "beyond-wco" indices, like Graphflow [18], ADOPT [32], and EmptyHeaded [1], instead, are faster than LTJ 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.

We remark that our compact indices run in main memory and would not be disk-friendly. While their compactness make them fit in memory for larger datasets, a relevant future work direction is to design compact representation formats for disk or distributed memory, where compactness translates into fewer I/Os or communication at query resolution time.

Another limitation of our compact indices is that they do not support updates. When updates are infrequent, one can maintain (comparatively few) inserted and deleted tuples in a classic data structure and consider them when solving queries, rebuilding the static data structure when the classic one becomes too large. When updates are frequent, one can resort to more sophisticated versions of the rebuilding scheme [20], or to the use of dynamic compact data structures [21].

Acknowledgments

Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile. A.G. is funded in part by MCIN/AEI/10.13039/501100011033: grant PID2020-114635RB-I00 (EXTRACompact); by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe": PID2021-122554OB-C33 (OASSIS) and PID2022-141027NB-C21 (EARTHDL); by MCIN/AEI/10.13039/501100011033 and "NextGenerationEU"/ PRTR: grants TED2021-129245B-C21 (PLAGEMIS), PDC2021-120917-C21 (SIGTRANS) and by GAIN/Xunta de Galicia: GRC: grants ED431C 2021/53, and CIGUS 2023-2026. G.N. is funded in part by Fondecyt Grant 1-230755, Chile.

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

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

[5] 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.

[6] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[7] O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *Data Engineering Bulletin*, 35(1):3–8, 2012.

[8] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(11):1891–1904, 2020.

[9] 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.

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

[11] 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.

[12] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[13] 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.

[14] 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.

[15] J. Leskovec. Stanford Large Network Dataset Collection: LiveJournal social network. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[16] 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.

[17] F. Manola and E. Miller. *RDF Primer*. W3C Recommendation. 2004. `http://www.w3.org/TR/rdf-primer/`.

[18] 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.

[19] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[20] J. I. Munro, Y. Nekrich, and J. S. Vitter. Dynamic data structures for document collections and graphs. In *Proc. 34th ACM Symposium on Principles of Database Systems (PODS)*, pages 277–289, 2015.

[21] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[22] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19:91–113, 2010.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.
[30] D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.
[31] 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.
[32] 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.
[33] 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.
[34] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*, pages 82–94, 1981.