The Ring: Worst-Case Optimal Joins in Graph Databases using (Almost) No Extra Space

DIEGO ARROYUELO, DCC, Escuela de Ingeniería, Pontificia Universidad Católica & IMFD, Chile
ADRIÁN GÓMEZ-BRANDÓN, Universidade da Coruña & CITIC & IMFD, Spain
AIDAN HOGAN, DCC, University of Chile & IMFD, Chile
GONZALO NAVARRO, DCC, University of Chile & IMFD, Chile
JUAN REUTTER, DCC, Escuela de Ingeniería, Pontificia Universidad Católica & Instituto de Ingeniería Matemática y Computacional, Pontificia Universidad Católica & IMFD, Chile
JAVIEL ROJAS-LEDESMA, DCC, Universidad de Chile & IMFD, Chile
ADRIÁN SOTO, FIC, Universidad Adolfo Ibáñez & IMFD, Chile

We present an indexing scheme for triple-based graphs that supports join queries in worst-case optimal (wco) time within compact space. This scheme, called a *ring*, regards each triple as a cyclic string of length 3. Each rotation of the triples is lexicographically sorted and the values of the last attribute are stored as a column, so we obtain the order of the next column by stably re-sorting the triples by its attribute. We show that, by representing the columns with a compact data structure called a wavelet tree, this ordering enables forward and backward navigation between columns without needing pointers. These wavelet trees further support wco join algorithms and cardinality estimations for query planning. While traditional data structures such as B-Trees, tries, etc., require 6 index orders to support all possible wco joins over triples, we can use one ring to index them all. This ring replaces the graph and uses only sublinear extra space, thus supporting wco joins in almost no space beyond storing the graph itself. Experiments querying a large graph (Wikidata) in memory show that the ring offers nearly the best overall query times while using only a small fraction of the space required by several state-of-the-art approaches.

We then turn our attention to some theoretical results for indexing tables of arity d higher than 3 in such a way that supports wco joins. While a single ring of length d no longer suffices to cover all $d!$ orders, we need much fewer rings to index them all: $O(2^d)$ rings with a small constant. For example, we need 5 rings instead of 120 orders for $d = 5$. We show that our rings become a particular case of what we dub *order graphs*, whose nodes are attribute orders and where stably sorting by some attribute leads us from an order to another, thereby inducing an edge labeled by the attribute. The index is then the set of columns associated with the edges, and a set of rings is just one possible graph shape. We show that other shapes, like for example a single ring instead of several ones of length d, can lead us to even smaller indexes, and that other more general shapes are also possible. For example, we handle $d = 5$ attributes within space equivalent to 4 rings.

Authors' addresses: Diego Arroyuelo, DCC, Escuela de Ingeniería, Pontificia Universidad Católica & IMFD, Santiago, Chile, diego.arroyuelo@uc.cl; Adrián Gómez-Brandón, Universidade da Coruña & CITIC & IMFD, A Coruña, Spain, adrian.gbrandon@udc.es; Aidan Hogan, DCC, University of Chile & IMFD, Santiago, Chile, ahogan@dcc.uchile.cl; Gonzalo Navarro, DCC, University of Chile & IMFD, Santiago, Chile, gnavarro@dcc.uchile.cl; Juan Reutter, DCC, Escuela de Ingeniería, Pontificia Universidad Católica & Instituto de Ingeniería Matemática y Computacional, Pontificia Universidad Católica & IMFD, Santiago, Chile, jreutter@ing.puc.cl; Javiel Rojas-Ledesma, jrojas@dcc.uchile.cl, DCC, Universidad de Chile & IMFD, Chile; Adrián Soto, adrian.soto@uai.cl, FIC, Universidad Adolfo Ibáñez & IMFD, Chile.

0362-5915/2024/6-ARTXXX $15.00
https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

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

Additional Key Words and Phrases: Worst-case optimal joins; graph patterns; graph databases; graph indexing; column stores; wavelet trees

ACM Reference Format:
Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, Juan Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2024. The Ring: Worst-Case Optimal Joins in Graph Databases using (Almost) No Extra Space. *ACM Trans. Datab. Syst.* XX, X, Article XXX (June 2024), 54 pages. https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

1 Introduction

Worst-case optimal (*wco*) join algorithms [53] are able to process join queries in time proportional to the *AGM bound* [8]: the maximum possible output size produced by the join over a relational database of a certain size. Such algorithms can be strictly better than traditional query plans using pairwise joins [53, 64], and thus represent a key advance for query processing over databases.

Leapfrog-TrieJoin (LTJ) is a seminal wco join algorithm based on iteratively "eliminating" attributes from a join query [64]. This algorithm will be defined in Section 2.2, where we illustrate the main idea over Figure 1 for now. The example includes the relations R, S, T, along with the query $Q = \mathsf{R} \bowtie \mathsf{S} \bowtie \mathsf{T}$ computing their natural join. To evaluate this query, LTJ first chooses an ordering of the attributes in Q, say $(\mathsf{x}, \mathsf{y}, \mathsf{z})$ (details of this ordering are discussed later). For the first attribute x, LTJ finds all constants a such that the query $\sigma_{\mathsf{x}=a}(\mathsf{R} \bowtie \mathsf{T})$ gives some solution, here joining all relations that mention x in the join (R and T). In this case $\sigma_{\mathsf{x}=1}(\mathsf{R} \bowtie \mathsf{T})$ and $\sigma_{\mathsf{x}=2}(\mathsf{R} \bowtie \mathsf{T})$ give solutions, while $\sigma_{\mathsf{x}=3}(\mathsf{R} \bowtie \mathsf{T})$ does not. We thus say that 1 and 2 eliminate x. Next LTJ eliminates y: for each constant a found to eliminate x in the previous step, we find all constants b that eliminate y, that is, such that $\sigma_{\mathsf{x}=a \wedge \mathsf{y}=b}(\mathsf{R} \bowtie \mathsf{S})$ gives solutions. Given $a = 1$, we find $b := 2$ and $b := 3$, while given $a = 2$ we find $b := 3$. We thus say that $(1, 2)$, $(1, 3)$ and $(2, 3)$ eliminate (x, y). Finally LTJ eliminates z: for each elimination (a, b) of (x, y) computed previously, we find all constants c that eliminate z. Given $(a, b) = (1, 2)$, we find $c := 4$; given $(a, b) = (1, 3)$, we again find $c := 4$; given $(a, b) = (2, 3)$ we find no valid eliminations. Since the tuples $(1, 2, 4)$ and $(1, 3, 4)$ eliminate all attributes (x, y, z), they are thus the final solutions of the query Q computed by LTJ.

In order for LTJ to satisfy wco guarantees, the constants that eliminate a given attribute must be enumerated efficiently. If the sequence of tuples of the relations are indexed in data structures that enable fast prefix lookups (tries, B-Trees, etc.) in the same order in which the variables are eliminated, then we can efficiently intersect all the candidates for the next elimination across the different tables. It is insufficient to index just one order per relation, however, because different attributes of a relation can be bound at different stages of query processing, and if the bound attributes do not form a prefix of the order, expensive post-filtering is required. Furthermore, even if all orderings are wco, in practice "choosing a good variable ordering is crucial for performance" [64], so wco implementations can ideally choose the order during query plan generation [33]. The number of index orders required to efficiently support wco join algorithms is then $d!$ for a relation of arity d, which makes storage requirements quickly unaffordable for arities as low as $d = 4$.

Several authors have proposed techniques to cope with this problem. Veldhuizen [64] proposes to load indexes lazily as needed by the LTJ algorithm, which though a practical compromise, slows down queries that require new indexes and puts a high bound on the required space. Freitag et al. [24] propose an indexing scheme that can be efficiently built during query execution, and demonstrate how to integrate these algorithms into a functioning relational database. Still, this implementation maintains the discussed stress on time and space. Navarro et al. [49] use a compact

R

x	y
1	2
1	3
2	3

S

y	z
2	4
3	4
3	5

T

z	x
2	3
3	2
4	1

R ⋈ S ⋈ T

x	y	z
1	2	4
1	3	4

Fig. 1. Example of three relations and their natural join

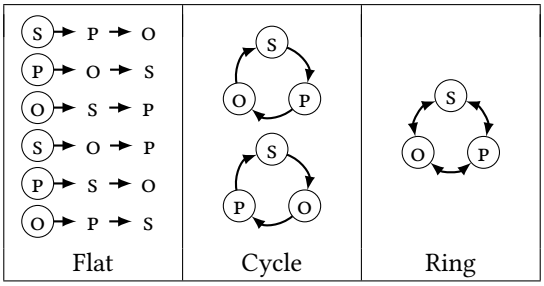


Fig. 2. Illustration of triple indexing schemes

data structure for wco joins that imposes a particular order, which eliminates one bit of every variable per round, and thus it does not need to index multiple orders. In exchange, its time complexity includes a factor that is exponential on the arity of the output relation. A variety of other wco algorithms have been proposed [1, 34, 36, 52, 54], yet they have focused mostly on refinements to improve time rather than space requirements. As such, the tradeoff between space and time in wco join algorithms remains, to the best of our knowledge, an open problem.

Graph databases can be modeled as relations of low fixed arity, and as such support wco join algorithms as well. Various works have found that wco join algorithms are well-suited for graph databases because, as a consequence of representing low arity relations, graph queries tend to feature many joins [1, 33, 34, 54]. For example, RDF graphs are sets of subject–predicate–object triples of the form (s, p, o). Each triple can be interpreted as an edge $s \xrightarrow{p} o$ in a labeled graph, or as a tuple of a relation of arity 3. The SPARQL query language for RDF is then based on *triple patterns* (equivalent to relational selections by equality) and *basic graph patterns* (equivalent to natural joins on those selections). Various compact RDF stores [3, 7, 13, 14, 17, 20, 43, 56, 67] handle triple patterns, and some can even handle basic graph patterns of some complexity [3, 7, 13, 14, 17, 43, 67]. None of those support wco join algorithms, however. This is not only a matter of implementing a new algorithm on existing data representations: as per our preceding discussion, even in the favorable case of arity 3, a complete index supporting wco joins will typically require $3! = 6$ index orders, leading to high levels of redundancy and high space requirements.

Henceforth, we refer to queries combining multiple equality selections and natural joins as *join queries*. Join queries correspond to basic graph patterns in the case of graph databases.

Contribution. We propose an indexing scheme, called a *ring*, that greatly reduces space requirements for supporting wco joins. Our scheme indexes triples (s, p, o) denoting a labeled graph, or equivalently, a relation of arity 3 with attributes S, P, and O. From an abstract viewpoint, the ring regards the attributes as *bidirectional cyclic strings* of length 3, so that any attribute reordering can be read from somewhere in the cycle in some direction. This allows the ring to support wco joins with only one index order, and thus within sublinear space beyond storing the graph itself.

Figure 2 illustrates three possible indexing schemes, circling the attribute from which the index order starts. In the (traditional) "flat" indexing scheme, we require six orders for wco joins using LTJ, specifying constants for attributes in sequence of the given order; we can then read the first unbound attribute in sorted order. For example using the order (P)→ O → S we can efficiently return: predicates in order, then objects of a given predicate, and then subjects of a given predicate and object; however, we cannot efficiently find the subjects of a given object. The "cycle" scheme encapsulates the cyclic indexing scheme of Brisaboa et al. [13], in which we can start at any attribute and proceed in the order shown, reading the first unbound attribute. Hence, for the case of triples, two orders are needed to cover all the possible patterns. This idea is related to ours, but cannot be easily extended to support LTJ. We thus propose the "ring scheme", where we can traverse the attributes in either direction, supporting wco graph joins with only one order.

We show how to implement the ring index using a compact data structure called a wavelet tree [30, 47], and characterize the space it requires. We prove that this implementation enables the evaluation of wco joins over graphs using an LTJ-style algorithm. Our experiments on a subgraph of Wikidata [66] with a billion edges and a real query log shows that the ring uses 8% extra space on top of the raw integer data (while compressing the data to 65% if we consider its string form) and 4–6 times less space than various prominent non-wco implementations (Jena, RDF-3X, Virtuoso, Blazegraph), while being 2–6 times faster on average to solve basic graph patterns. Two prominent wco systems, Jena LTJ [33] and EmptyHeaded [1], use 10 and 160 times more space than the ring, respectively, and only Jena LTJ outperforms it, by 20% on average. Only Qdag [49], a recent wco succinct index, is smaller than our basic ring index and faster than it in small cyclic queries, but hundreds of times slower overall. A compressed ring variant we develop further reduces the space of the ring to 60% and matches the space of Qdag. While 40% slower than the ring, the compressed ring is still faster on average than all the non-wco systems and Qdag.

Finally, we return to the general problem of indexing several higher arity relations to support wco join algorithms in the context of relational databases. Although we need more than one ring to support every possible attribute order in dimension $d > 3$, we show that bidirectionality and cyclicality lead to using much fewer than $d!$ orders, namely $O(2^d)$ with small multiplying constants. For example, for arity $d = 5$, we require just 5 rings instead of $5! = 120$ traditional indexes. We further reduce the space by introducing a more general indexing model akin to *column stores* [60], where each column is stored separately in some convenient order. The ring, in this model, corresponds to storing columns of attributes S, P and O, but sorted in a way that allows one to quickly retrieve tuples or ranges of tuples from just these three columns (note that just storing S, P and O in their original order does not allow this, as we cannot efficiently retrieve, for example, all subjects connected to a given object). We extend this idea to higher dimensions, where using our model we can check if a set of ring-like columns allows for efficient navigation and wco joins, by looking at a completeness property of what we call the *order graph*. Using our framework we show that, for example, using a large ring (that repeats some columns) we can reduce the space for arity $d = 5$ to the equivalent of 4 indexes, and that even more general graph shapes can be used. Those results enable the use of wco join algorithms supported by previously-indexed relations on arities that are intractable with current techniques.

Our indexes are static, that is, they must be fully rebuilt in order to reflect updates to the database. While this is acceptable in various scenarios and our construction algorithms are efficient, we show in the end how our indexes can handle fine-grained sequences of queries and updates without full reconstructions, at the cost of a logarithmic penalty factor in the query times.

Limitations. Our proposed indexing scheme, though efficient in space, is based on an in-memory data structure that relies heavily on random accesses, which makes it difficult to migrate effectively

to disk, and could render it slower than in-memory indexes that feature more sequential access. The implemented scheme is currently read-only; however, we discuss updates in the final section of the paper, and how they could be addressed in future work. We currently focus on evaluating basic graph patterns; though support for other features of graph query languages could be simply layered on top, it may be possible in the future to optimize such features by pushing them to lower-level operations over the index. The indexing schemes for relations of arity $d > 3$ are discussed theoretically, but not yet implemented herein. Aside from the limitations stemming from frequent random accesses, which appear fundamental to our proposal, all other issues mentioned here can be addressed as part of future work.

Conference version. This article is an extended version of a conference publication [6] that introduced the ring, showed how one ring is sufficient to index graphs for wco joins, and presented experiments over the Wikidata graph. This article includes the following additional contributions:

- The conference version presented the ring in terms of the Burrows–Wheeler transform [15]. We present a new formulation of the ring in terms of stable sorting on column databases, which we hope will be more accessible to a broader audience not familiar with text indexing, and which naturally leads us to a more general approach for indexing higher-arity relations.
- We have performed additional optimizations to the ring implementation, which considerably improved its performance.
- We have expanded our implementation and experiments to include a specific ring variant for the case of triple patterns where the predicate component is always constant.
- We present new results on the number of rings required to index relations of arity d. We further present a novel indexing scheme, which we call *order graphs*, that generalizes the techniques of the ring and allows us to save further space.
- We describe in detail how our index can be modified to support updates on the fly.
- We include more detailed discussion and proofs for all results throughout the paper.

2 Related works and concepts

We introduce key concepts relating to graph joins, wco join algorithms, and compact data structures. Table 1 gives a glossary of the terms used throughout the paper.

2.1 Graph joins and patterns

2.1.1 Graphs. We assume the domain of graphs to be drawn from a totally ordered, countably infinite set $\mathcal{U}$ of constants called the *universe*. A *triple* $(s, p, o) \in \mathcal{U}^3$ then encodes a directed edge $s \xrightarrow{p} o$, labeled p, from node s to node o. A set of triples forms a labeled *graph*, as seen in Figure 3. Given a graph G, we denote by $|G|$ its amount of triples and by $\mathrm{dom}(G)$ the *domain* of G, that is, the subset of $\mathcal{U}$ used as constants in G. Given a constant $u \in \mathcal{U}$, we denote by $u + 1$ the next element in the total order after u in $\mathcal{U}$.

2.1.2 Graph patterns. A *basic graph pattern* is a graph in which some constants may be replaced by *variables* that can be matched against another graph. To be more precise, let $\mathcal{V}$ be an infinite set of variables, disjoint from $\mathcal{U}$. A *triple pattern* is then 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 represents an atomic query over the graph (equivalent to equality-based selections on a single ternary relation), and thus a basic graph pattern corresponds to a full conjunctive query (a.k.a. *join query*) over the relational representation of the graph. Let $\mathrm{vars}(Q)$ denote the set of variables used in 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

Table 1. Glossary of the main notation used throughout the paper.

Symbol	Meaning	Symbol	Meaning
$\mathcal{U}$	A totally ordered, countably infinite set of constants, called the *universe*, Sec. 2.1.1	Q_S	For a query Q and a set $S \subseteq \text{vars}(Q)$, the set of triple patterns of Q containing some variable in S, Sec. 2.2.3
$G, n := \|G\|$	A labeled graph and its number of triples, Sec. 2.1.1	$\text{rank}_c(S, i)$	For a string S over alphabet $[1, \tau]$, the number of symbols equal to $c \in [1, \tau]$ in $S[1..i]$, Sec. 2.3.1
$\text{dom}(G), U$	The domain of graph G, i.e., the subset of $\mathcal{U}$ used as constants in G, and number of constants in G, Sec. 2.1.1	$\text{select}_c(S, j)$	For a string S over alphabet $[1, \tau]$, the position of the jth occurrence of symbol $c \in [1, \tau]$ in S, Sec. 2.3.1
$\mathcal{V}$	An infinite set of variables, disjoint from $\mathcal{U}$, Sec. 2.1.2	*range-next-value*	For a string S over alphabet $[1, \tau]$, a range $[s..e]$, and a threshold $c_x \in [1, \tau]$, the smallest symbol $c \geq c_x$ that occurs in $S[s..e]$, Sec. 2.3.2
$Q, m := \|Q\|$	A graph pattern $\{t_1, \ldots, t_m\}$, such that $t_i \in (\mathcal{U} \cup \mathcal{V})^3$ are triple patterns, and its size, Sec. 2.1.2	$\mathcal{R}, \mathcal{A}$	A relation $\mathcal{R}$ on a set of attributes $\mathcal{A}$, Sec. 3.1
$\text{vars}(Q), v$	The set of variables of query Q and its size, Sec. 2.1.2	$d := \|\mathcal{A}\|$	The number of attributes of a relation, Sec. 3.1
$Q(G)$	The evaluation of query Q over the graph G, Sec. 2.1.2	$\Pi, \Pi(j)$	An order and its jth attribute, Def. 3.1
$\mu(Q)$	The image of Q under μ, Sec. 2.1.2	C_j^T, C_j	The jth column of a table $T[1..n][1..d]$ representing a relation $\mathcal{R}$ of n tuples and d attributes, Def. 3.3
D	A relational database instance, Sec. 2.2.1	$A_j[1..U+1]$	For a column $C_j[1..n]$, $A_j[c]$ is the number of occurrences of symbols smaller than c in C_j, Def. 3.7
Q^*	The AGM bound of query Q, Sec. 2.2.1	F_j	For $i \in [1..n]$ and $c := C_j[i]$, $F_j(i)$ is the row corresponding to c in the column obtained by stably sorting the table by column j, Def. 3.7
$\text{leap}(t, x, c)$	For a variable $x \in \mathcal{V}$ and a constant $c \in \mathcal{U}$, yields the smallest constant $c_x \geq c$ from $\mathcal{U}$ such that the triple pattern t (which contains variable x) has solutions in G after replacing x by c_x, Def. 2.1	$\mathcal{G}(\mathcal{V}, \mathcal{E})$	An order graph, Def. 7.1
$\text{seek}(\mu, j, c)$	For a constant $c \in \mathcal{U}$ and a variable $x_j \in Q$, yields the smallest $c_{\min} \geq c$ from $\mathcal{U}$ that eliminates x_j in $\mu(Q_{\{x_j\}})$, Alg. 1		

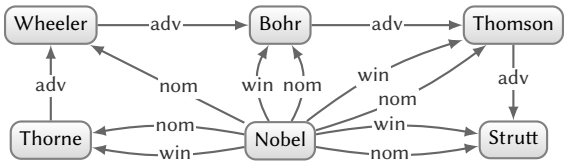
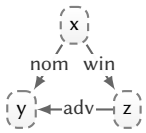


Fig. 3. Graph of Nobel winners, nominees and advisors



x	y	z
Nobel	Strutt	Thomson
Nobel	Thomson	Bohr
Nobel	Wheeler	Thorne

Fig. 4. Basic graph pattern (left) and its evaluation over the graph of Figure 3 (right)

each variable $x \in \text{vars}(Q)$ in Q by $\mu(x)$. Figure 4 illustrates a basic graph pattern and its evaluation over a graph, which yields three solutions. The central problem of interest to us in this article is to compute the complete set of solutions for $Q(G)$ in worst-case optimal time and using little space.

2.2 Worst-case optimal joins

2.2.1 AGM bound. The AGM bound [8] defines a limit on the number of solutions for natural join queries in a relational setting of the form $Q := r_1 \bowtie \ldots \bowtie r_m$, where $r_1, \ldots, r_m$ are (pairwise distinct) relation names. Given a natural join query Q and a relational instance D, the AGM bound of Q over D is the maximum number of tuples generated by evaluating Q over any instance D' of size not greater than D.[1] If we simply assume that the size of all relations is in $O(n)$, we can speak of the AGM bound of Q, denoted herein by Q^*, as a function of n.

[1]The size of an instance D' over schema $r_1, \ldots, r_m$ is said to be not greater than an instance D if for each relation r_i, the number of tuples of r_i in D' is not greater than the number of tuples of r_i in D.

When applying the AGM bound over graph patterns [33], there are three details requiring attention, since such patterns can be more complex than simple relational join queries. We can deal with them using the same techniques used by Gottlob et al. [28] to extend the AGM bound to conjunctive queries. First, graph patterns involve self-joins on a single ternary relation. But if we rewrite the query to make each relation name distinct, the AGM bounds differ only by a factor depending on the total number of self-joins. Second, graph patterns involve constants from the set $\mathcal{U}$; for example, the graph pattern Q of Figure 4 uses constants `win`, `nom` and `adv`. In this case we can recover the bound by transforming Q into a query Q' in which each pattern $t \in Q$ using k ($0 \leq k \leq 3$) constants is transformed into a relation of arity $3 - k$ in which we filter the base relation by the appropriate constants. Third, the same variable can appear multiple times in a triple pattern. Again, this case can be covered by creating a relation that uses one attribute to represent the variable and that stores all tuples that have the same value in the corresponding positions. Hence the same bound applies to graph patterns, within a factor that depends only on the query.

Our joins on general relations R also feature selections by equality predicates. The AGM bound can be similarly extended by replacing R by a relation R' where we have already performed the selections and projected on the remaining columns, analogously to what we discussed for triples.

2.2.2 Worst-case optimality. A *join algorithm* accepts a join query Q and a database instance D as input, and enumerates $Q(D)$ – the solutions for Q over D – as its output. A join algorithm is called *worst-case optimal* (wco) if it can run in time $O(Q^*)$ taking the number of terms in Q and attributes in D as constants (a.k.a. data complexity). The intuition is that in the worst case a join algorithm has to enumerate Q^* results, thus taking $\Omega(Q^*)$ time. Though join algorithms do exist that run within time $O(Q^*)$ [53, 64], a logarithmic factor $O(Q^* \log n)$ is often permitted to allow more flexibility (e.g., allowing binary search over sorted relations rather than hashing [64]).

Being wco is a non-trivial property of a join algorithm. Conventional algorithms convert join queries into binary join trees, where joins are evaluated pairwise using nested-loop joins, hash joins, merge joins, etc. Such approaches are not wco. Take, for example, the join query Q of Figure 1. If we join the pair $Q_1 := \mathsf{R} \bowtie \mathsf{S}$ in order to later evaluate $Q_2 := Q_1 \bowtie \mathsf{T}$, then Q_1^* is already in the order of n^2, while Q^* is in the order of $n^{3/2}$, and hence this plan is not wco [8]. Joining any pair of relations in Q_1 will be in the order of n^2 and thus no such plan can be wco. Ngo et al. [53] proposed the first wco join algorithm confirmed to run in time $O(Q^*)$ (later named NPRR). This algorithm was followed by Leapfrog TrieJoin (LTJ) [64], a simpler algorithm running in time $O(Q^* \log n)$.

2.2.3 Leapfrog TrieJoin. As illustrated in the introduction, LTJ evaluates join queries one attribute-at-a-time rather than one relation-at-a-time. For simplicity, we describe the LTJ algorithm in the context of graphs [33], where the idea extends naturally to higher-arity relations [64].

LTJ runs over an abstract trie data structure that represents a graph G. Concretely, it builds upon an abstraction for data access called a *trie-iterator*, which features one operation: `leap`.

Definition 2.1 (Trie-iterator). A *trie-iterator* for a graph G is an implementation of $\texttt{leap} : (\mathcal{U} \cup \mathcal{V})^3 \times \mathcal{V} \times \mathcal{U} \to \mathcal{U} \cup \{\perp\}$. Given a variable $x \in \mathcal{V}$, a triple pattern t with the variable x, and a constant $c \in \mathcal{U}$, $\texttt{leap}(t, x, c)$ returns the smallest constant $c_x \geq c$ from $\mathcal{U}$ such that t has solutions in G after replacing x by c_x. If there is no such value c_x, $\texttt{leap}(t, x, c)$ returns the special value $\perp$.

Veldhuizen [64] shows that for LTJ to run in $O(Q^* \log n)$ time, it suffices that the trie iterators support `leap` in $O(\log n)$ time (data complexity). Consider a graph pattern $Q := \{t_1, \ldots, t_m\}$, and let $v := |\text{vars}(Q)|$. For a subset $S \subseteq \text{vars}(Q)$ of variables, further let Q_S denote the set of triple patterns in Q that contain some variable in S. Algorithm 1 details how LTJ uses the `leap` operation to evaluate a basic graph pattern Q over a graph G. LTJ first defines an initial ordering $(x_1, \ldots, x_v)$ of $\text{vars}(Q)$; the specific ordering does not affect wco guarantees and will be discussed later.

Algorithm 1 LTJ for the evaluation of basic graph patterns

Input: A basic graph pattern Q, a trie-iterator $\mathcal{T}$ for a graph G,
and an ordering $(x_1, \ldots, x_v)$ of the variables in vars(Q)

`leapfrog_join()`:
Output: Reports all the tuples in $Q(G)$
1: **call** `leapfrog_search({}, 1)`

`leapfrog_search(`μ, j`)`:
Input: An index $1 \leq j \leq v+1$, and a mapping μ defined for the variables $\{x_k \mid k < j\}$
Output: Reports the solutions in $Q(G)$ that extend mapping μ by eliminating x_j onwards
1: **if** $j = v+1$ **then report** μ as an output solution
2: **else**
3: $c :=$ `seek(`$\mu, j, \min \mathcal{U}$`)`
4: **while** $c \neq \perp$ **do**
5: $\mu' := \mu \cup \{(x_j := c)\}$
6: **call** `leapfrog_search(`$\mu', j+1$`)`
7: $c :=$ `seek(`$\mu, j, c+1$`)`

`seek(`μ, j, c`)`:
Input: An index $1 \leq j \leq v$, a mapping μ defined for the variables $\{x_k \mid k < j\}$, and a value $c \in \mathcal{U}$
Output: The smallest value $c_{\texttt{min}} \geq c$ that eliminates x_j in $\mu(Q_{\{x_j\}})$
1: Let $t_1, \ldots, t_m$ be the triple patterns in $Q_{\{x_j\}}$
2: For $1 \leq i \leq m$, let $\mu(t_i)$ be the triple pattern t_i with its variables x_k, for $k < j$, replaced by $\mu(x_k)$
3: **while** true **do**
4: $c_{\texttt{min}} := c$
5: **for** $i \in [1\,..\,m]$ **do**
6: $c := \mathcal{T}$`.leap(`$\mu(t_i), x_j, c$`)`
7: **if** $c = \perp$ **then return** $\perp$
8: **if** $c_{\texttt{min}} = c$ **then return** $c_{\texttt{min}}$

Starting with x_1, LTJ finds each elimination $c \in \text{dom}(G)$ (also called a *binding* or *instantiation*) for x_1 such that, for every triple pattern $t \in Q_{\{x_1\}}$, if x_1 is replaced by c in t, then the evaluation of the modified t over G is non-empty. This is equivalent to intersecting the eliminations of x_1 over all the individual triple patterns $t \in Q_{\{x_1\}}$. LTJ uses `seek` to find each consecutive value c in that intersection. The `seek` procedure uses `leap` to iteratively find in each triple pattern $t \in Q_{\{x_1\}}$ the next possible candidate for the intersection, which corresponds to the smallest elimination for x_1 in t that is over some threshold c. When `seek` finally finds a value c that appears in all triple patterns $t \in Q_{\{x_1\}}$, LTJ eliminates x_1 with c, and keeps looking for the next eliminations for x_1.

Upon finding the first elimination c of x_1, the algorithm creates a mapping $\mu := \{(x_1 := c)\}$. Next LTJ finds values d that eliminate x_2 in $\mu(Q_{\{x_2\}})$ using the same form of intersection as before. When the first elimination d of x_2 is found, the current mapping is extended to $\mu := \{(x_1 := c), (x_2 := d)\}$. The process then continues to the next variable until all variables are eliminated, in which case μ is a solution. If no further elimination is found for a variable x_j and current mapping μ, the process backtracks to modify μ with the next elimination for x_{j-1}, and so on. LTJ terminates when all mappings for x_1 have been exhausted.

We are then left to consider the implementation of `leap`. Per the name "trie-iterator", the original implementation in a relational setting was based on (virtual) tries built for each relation, with levels of the trie corresponding to attributes of the relation, and each unique root-to-leaf path encoding a tuple of the relation [64]. In a graph context, a trie would be defined with one level for subjects, one for predicates, and one for objects, and with each root-to-leaf path encoding a triple of the graph. However trie-iterators based on traditional indexes (e.g., on B-trees) can only meet the $O(\log n)$-time requirement for $\texttt{leap}(t, x, c)$ if the positions of the constants in t form a prefix of the index order. In the case of graphs, supporting all possible triple patterns and query plans within the necessary time constraints implies indexing 6 different orders for all permutations of levels for subject, predicate and object (see "flat" in Figure 2). More generally, for an arity of d, we need a total of $d!$ orders (which we improve to $\Theta(2^d d^{1/2})$, see Section 6.2), that is, an exponential number of indexes. This makes LTJ space-demanding on graphs and impractical on most relational databases.

In Section 3 we introduce a read-only indexing scheme for a graph G that supports $\texttt{leap}(t, x, c)$ in $O(\log n)$ time, with no restrictions on the order of the constants in t, and using almost no extra space beyond that required to represent G.

2.2.4 Other wco algorithms. Recent years have seen various further proposals of wco algorithms [1, 2, 33, 34, 36, 37, 49, 52, 54, 62]. While many such algorithms are proposed in a relational context, they can be applied over graphs represented as ternary relations. However, most such works focus on improving time, or dealing with more complex queries, rather than reducing space requirements. As an exception, Navarro et al. [49] use space close to that of the raw data for any d, by using a particular order that eliminates one bit of every variable in each round. In exchange, the time complexity includes a factor that is exponential on the arity of the relation. Some recent papers have looked at ways to combine pairwise joins with wco joins to reduce space requirements. Freitag et al. [24] propose a hash-based indexing scheme that can be efficiently built on-the-fly at query time, and demonstrate how to integrate these algorithms into a functioning relational database; they also propose to use wco joins only when beneficial. Our proposal avoids on-the-fly indexing in the context of graphs. Graphflow [42] integrates wco joins with pairwise joins in order to generate hybrid plans for evaluating graph queries. While their work focuses on query planning, our focus is on space-efficient indexing techniques – inspired by indexes for text – that support wco joins.

2.3 Compact data structures

A compact data structure [48] stores the data plus the extra structures needed to efficiently query it, all within space close to that needed to store the raw data, in plain or in compressed form.

We describe the main compact data structures used in this work. As usual in this area, we assume the RAM computation model, where the typical arithmetic and logical operations on machine words of $\Theta(\log n)$ bits are carried out in constant time.

2.3.1 Bitvectors. A bitvector $B[1..n]$ is a sequence of bits that supports, apart from access to any bit $B[i]$, two key operations:

$\textbf{rank}_b(B, i)$ is the number of bits equal to $b \in \{0, 1\}$ in $B[1..i]$.
$\textbf{select}_b(B, j)$ is the position of the jth occurrence of bit $b \in \{0, 1\}$ in B.

A bitvector $B[1..n]$ can be stored in n bits, and with another $o(n)$ bits it can support access, `rank`, and `select` operations in $O(1)$ time [44]. A sparse bitvector, with m 1s, can be represented with $m \log_2(n/m) + O(m) + o(n)$ bits while again supporting these operations in $O(1)$ time [58].

2.3.2 Wavelet trees. A wavelet tree [30, 47] is a binary tree that represents a string $S[1..n]$ from an alphabet $[1, \tau]$ using $n \log_2 \tau + o(n \log \tau)$ bits of space (and can use compressed space).

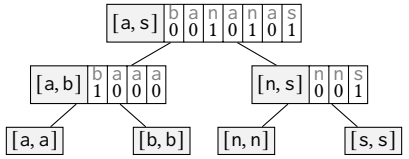


Fig. 5. Example wavelet tree for bananas

Structure. Each node represents a range of alphabet symbols. The root represents $[1, \tau]$ and the c^{th} left-to-right leaf represents $[c, c]$, that is, the interval containing only the c^{th} character, which we denote as c. If an internal node represents $[a, b]$, then its left child represents $[a, \lfloor (a+b)/2 \rfloor]$ and its right child represents $[\lfloor (a+b)/2 \rfloor + 1, b]$. Conceptually, a node representing $[a, b]$ stores the subsequence $S_{a,b}$ of S formed by the symbols that belong to $[a, b]$. Let $n_{a,b}$ be the length of $S_{a,b}$. In practice, this node only stores a bitvector $B_{a,b}[1 \,..\, n_{a,b}]$ with a 0 at position i if $S_{a,b}[i] \leq (a+b)/2$ and a 1 otherwise (marking whether $S_{a,b}[i]$ belongs to its left or right child). The leaves store nothing. Figure 5 depicts the wavelet tree for bananas; only bitvectors and pointers are stored, while other elements are illustrative.

Space. In a wavelet tree for a string $S[1 \,..\, n]$, the total number of bits stored at the nodes of each non-leaf level is also n. Since the tree has $\lceil \log_2 \tau \rceil$ non-leaf levels, storing all the bitvectors requires at most $n\lceil \log_2 \tau \rceil$ bits, just like a plain representation of S. To support operations efficiently (e.g., accessing $S[i]$), all the bitvectors must support fast `rank` and `select` operations, which requires $o(n \log \tau)$ further bits of space overall. The pointers of the tree add $O(\tau \log n)$ bits to its space usage, but this space can be saved by using a pointerless variant called a *wavelet matrix* [16], which offers the same functionality as wavelet trees. In what follows, we assume wavelet trees for simplicity.

Access, rank, and select. A wavelet tree on S supports access to any $S[i]$, as well as `rank` and `select` operations on S (which are defined just as for bitvectors). This is done by traversing one root-to-leaf path in the wavelet tree, in $O(\log \tau)$ time.

Let us show how to access $S[i]$: we start with $[a, b] := [1, \tau]$ and $i' := i$ at the root. If $B_{a,b}[i'] = 0$, we set $b := \lfloor (a+b)/2 \rfloor$, $i' := \mathsf{rank}_0(B_{a,b}, i')$, and continue by the left child. Otherwise, we set $a := \lfloor (a+b)/2 \rfloor + 1$, $i' := \mathsf{rank}_1(B_{a,b}, i')$, and continue by the right child. When we arrive at a leaf, it holds $a = b = S[i]$, and moreover $\mathsf{rank}_{S[i]}(S, i) = i'$.

Example. We can compute $S[5]$ on the wavelet tree of Figure 5 by reading $B_{\mathsf{a,s}}[i' = 5] = 1$ in the root bitvector $B_{\mathsf{a,s}} = 0010101$, thus going right with $i' := \mathsf{rank}_1(B_{\mathsf{a,s}}, 5) = 2$. On the right child of the root, we read $B_{\mathsf{n,s}}[i' = 2] = 0$, so we go left with $i' := \mathsf{rank}_0(B_{\mathsf{n,s}}, 2) = 2$. We then arrive at the leaf $[\mathsf{n}, \mathsf{n}]$, so we know that $S[5] = \mathsf{n}$, and moreover $\mathsf{rank}_{\mathsf{n}}(S, 5) = i' = 2$. □

Operation $\mathsf{rank}_c(S, i)$ is analogous. We do as for accessing $S[i]$, except that we go left if $c \leq (a+b)/2$ and right if not. When we arrive at the leaf $[c, c]$, it holds that $\mathsf{rank}_c(S, i) = i'$.

For $\mathsf{select}_c(S, i)$ we descend towards the leaf $[c, c]$, set $i' := i$, and return i' from the recursion. At an internal node $[a, b]$, if we receive i' from the left child, we return $\mathsf{select}_0(B_{a,b}, i')$ to the caller; otherwise we return $\mathsf{select}_1(B_{a,b}, i')$. When the root returns i', it holds that $i' = \mathsf{select}_c(S, i)$.

Advanced operations. The wavelet tree supports other operations [47]. Among these it can list all distinct values c in a range $S[s \,..\, e]$, also computing $[s_c \,..\, e_c] := [\mathsf{rank}_c(S, s-1) + 1 \,..\, \mathsf{rank}_c(S, e)]$ for each, taking time $O(k \log(\tau/k))$ to report the k distinct values in $S[s \,..\, e]$ [26]. We will use this operation to optimize for *lonely variables* appearing in only one triple pattern [33] in Section 4.2.

Wavelet trees also support the *range-next-value* operation in time $O(\log \tau)$ [26], which we will use to support operation `leap` in Section 3.4.2. Specifically, given a range $S[s\,..\,e]$ and a threshold $c_x \in [1, \tau]$, *range-next-value* finds the smallest symbol $c \geq c_x$ that occurs in $S[s\,..\,e]$. The algorithm also finds the range $[s_c\,..\,e_c]$ so that $S[s\,..\,e]$ contains from the s_c^{th} to the e_c^{th} occurrence of c in S (i.e., $s_c := \mathsf{rank}_c(S, s-1) + 1$ and $s_e := \mathsf{rank}_c(S, e)$).

3 One ring to index them all

We now present our ring index for a graph G, and describe how to implement the trie-iterator interface over it. This allows LTJ to evaluate basic graph patterns over G in wco time using almost no space beyond that of a raw, and even a compressed, representation of G. The ring index is inspired by text indexing techniques. Rather than store the tuples (rows) of a relation, we store each attribute (column) separately as a list of values. Crucially, the elements of each such column are stored in an order decided by all other columns; thus each column carries information about all other columns that can be used to retrieve the tuples of the original relation. We start with a more general description for relational tables that is later instantiated for the case of graphs.

3.1 Orders, tables, and columns

We map all the constants in the domain $\mathcal{U}$ to consecutive integers $[1\,..\,U]$. A relation $\mathcal{R}$ on attributes $\mathcal{A}$ is then seen as a set of n tuples from U^d, where $d := |\mathcal{A}|$. Different relations may be associated with different relation names; for simplicity, we focus on one relation. Our index builds on the concepts of *order*, *table*, and *column*. The first is just an ordering of attributes.

Definition 3.1. An *order* is any bijection from $[1\,..\,d]$ to $\mathcal{A}$. For technical convenience, we will assume that $\mathcal{A} := [1\,..\,d]$, and therefore an order is just a permutation Π of $[1\,..\,d]$.

We now define a table, which lexicographically sorts the tuples of a relation $\mathcal{R}$ in a given order.

Definition 3.2. The *table* corresponding to a relation $\mathcal{R}$ and an order Π is a matrix $T[1\,..\,n][1\,..\,d]$ of elements in $[1\,..\,U]$. Each table row $T[i]$ corresponds to a tuple in $\mathcal{R}$, with the cell $T[i][j]$ containing the value of the attribute $\Pi(j)$ in the tuple. The rows of T are sorted primarily by the values in the first column, then, upon ties, by the values in the second column, and so on. We say that T is sorted by the attributes $\Pi(1), \ldots, \Pi(d)$, that is, sorted by the order Π.

Finally, we define columns, which are simply the sequence of values in a given column of a table.

Definition 3.3. The jth *column* of a table T, C_j^T or just C_j, is the sequence of values $T[i][j]$, for increasing $i := 1, 2, \ldots, n$.

Our ring index for a relation $\mathcal{R}$ is a set of columns, one per attribute in $\mathcal{A}$ (we will relax this model in Section 7). Each column of the ring comes from a different table T corresponding to $\mathcal{R}$, each sorted with a particular order. The basic operation to build our index is to *re-sort* all the rows of a table by a column.

Definition 3.4. The *re-sort* of a table T (with order Π) by column j is a new table where column j is moved to the front of the first column. More precisely, the order of the re-sorted table is Π', with $\Pi'(1) := \Pi(j)$, $\Pi'(k) := \Pi(k-1)$ for $k \in [2\,..\,j]$, and $\Pi'(k) := \Pi(k)$ for $k \in [j+1\,..\,d]$.

By the definition of table, if we move column j of a table T to the front to get T', we must sort the rows of T' by the order Π'. The next lemma better characterizes how to sort T to get T'.

Lemma 3.5. *The order of the rows in the re-sort of a table T by column j is obtained by stably sorting the rows of T by column j.*

Proof. The table T' is sorted by its order Π', so a stable sorting of T by column j correctly sorts the rows by the attribute $\Pi'(1) = \Pi(j)$. The rows of T' with the same value in the first column should be sorted by columns $\Pi'(2), \ldots, \Pi'(d) = \Pi(1), \ldots, \Pi(j-1), \Pi(j+1), \ldots \Pi(d)$. In this second list, we can insert $\Pi(j)$ between $\Pi(j-1)$ and $\Pi(j+1)$ without affecting the order, since we are focused on rows where $\Pi(j)$ has the same value. The order on those rows then does not change with respect to sorting by the full list of attributes $\Pi(1), \ldots, \Pi(d)$ we have in T. Thus a stable sorting of T by column j yields the correct order of T'. □

In other words, re-sorting leads us from a table that represents $\mathcal{R}$ with some order Π to another table that represents $\mathcal{R}$ with a new order Π'.

3.2 Navigation between a table and its re-sorting

Assume we re-sorted table T by column j to obtain T'. We are interested in tracking any row i of T to its corresponding row i' in T'.

Definition 3.6. If T' is a re-sort of T, we say that a row i of T and a row i' of T' *correspond* if they are identical, that is, they represent the same tuple of $\mathcal{R}$.

We now show that a simple formula maps any row i of T to its corresponding row i' of T'.

Definition 3.7. Given a column $C_j[1..n]$, let $A_j[1..U+1]$ store in $A_j[c]$ the number of occurrences of symbols smaller than c in C_j, that is,

$$A_j[c] \;:=\; |\{i \in [1..n],\ C_j[i] < c\}|. \tag{1}$$

Then, let us define the function $F_j : [1..n] \to [1..n]$, as follows:

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

where $c := C_j[i]$. Recall that $\mathsf{rank}_c(C_j, i)$ counts the number of times c appears in $C_j[1..i]$.

Here, $F_j(i)$ counts the occurrences of symbols less than c (the ith value of column j) in all of column j and the occurrences of c up to and including the ith position of column j. We now show that $F_j(i)$ gives us the new position of the ith value of column j after a stable sort of that column, and thus that we can use function F_j to navigate from a table T to the re-sort of T by column j.

Lemma 3.8. *If T' is the re-sort of T by column j, and $i' := F_j(i)$, then rows $T[i]$ and $T'[i']$ correspond.*

Proof. By Lemma 3.5, T' is obtained by stably sorting T by column j. Thus, every row k where $T[k][j] < T[i][j]$ (i.e., $C_j[k] < C_j[i]$) will appear before $T[i]$ in T'. The number of those rows k is $A_j[c]$ where $c := T[i][j] = C_j[i]$, by Eq. (1). By the stable sort, the other rows $T[k]$ that precede $T[i]$ in T' are those where $T[k][j] = T[i][j] = c$ (i.e., $C_j[k] = c$) and $k < i$. Their number is then $\mathsf{rank}_c(C_j, i-1)$. Therefore, the row $T[i]$ will appear in T' precisely at position $i' := A_j[c] + \mathsf{rank}_c(C_j, i-1) + 1$. Since $C_j[i] = c$, $\mathsf{rank}_c(C_j, i-1) + 1 = \mathsf{rank}_c(C_j, i)$ and therefore $i' = F_j(i)$, per Eq. (2). □

Therefore, if we represent C_j with wavelet trees (Section 2.3.2), we can map in time $O(\log U)$ from rows of T to rows of T' by using only sublinear space on top of the plain storage of C_j. We can similarly map from the row i' in T' to i in T by computing the inverse of function F_j:

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

where c is such that $A_j[c] < i' \le A_j[c+1]$. This is easily verified by noting that rank is the inverse of select, that is, $\mathsf{rank}_c(C_j, \mathsf{select}_c(C_j, p)) = p$.

Our function F_j is akin to the last-to-first mapping in the Burrows–Wheeler Transform [10, 15] and the FM-index [21, 22] for compressed text indexing [6]. This mapping can be extended to

the *restriction* operation (called the backward step in the FM-index): Let T' be the re-sort of T by column j. Given a value c and a range $T[s\,..\,e]$ of rows in T, the restriction maps the range $T[s\,..\,e]$ to the range $T'[s'\,..\,e']$ of all the rows in T' containing c in the first column and corresponding to a row in $T[s\,..\,e]$ (when there are no cs in $C_j[s\,..\,e]$, the result is an empty range, $s' > e'$).

Lemma 3.9. *Let T' be the re-sort of T by column j. Then the set of rows from the range $T[s\,..\,e]$ that contain a value c in column j correspond in T' to another range of rows, $T'[s'\,..\,e']$, with*

$$\begin{aligned} s' &:= A_j[c] + \mathsf{rank}_c(C_j, s-1) + 1, \\ e' &:= A_j[c] + \mathsf{rank}_c(C_j, e). \end{aligned}$$

Proof. All the qualifying rows contain the same value c in column j, thus they will be contiguous in T' by Lemma 3.5. Consider the first row $T[k]$ in $T[s\,..\,e]$ containing a c in column j. By Lemma 3.8, row k of T corresponds to row $s' := F_j(k)$ of T'. Since this is the first c in $C_j[s\,..\,k]$, we have $F_j(k) = A_j[c] + \mathsf{rank}_c(C_j, k) = A_j[c] + \mathsf{rank}_c(C_j, s-1) + 1$. Now consider the last row $T[k']$ in $T[s\,..\,e]$ with a c in column j. Since this is the last c in $C_j[k'\,..\,e]$, by Lemma 3.8 it corresponds in T' to row $e' := F_j(k') = A_j[c] + \mathsf{rank}_c(C_j, k') = A_j[c] + \mathsf{rank}_c(C_j, e)$. The qualifying rows then map to $T'[s'\,..\,e']$. □

3.3 The ring index for graphs

The relation we index is a set of n integer triples (s, p, o), that is, with attributes S, P, and O, each taking values in $[1\,..\,U]$. For legibility, we describe orders as permutations of the attribute names, so order SPO means the order that maps 1 to S, 2 to P, and 3 to O. We will also refer to those orders to denote the corresponding tables, that is, table SPO is the set of triples sorted by the order SPO.

Definition 3.10. Given a set of n integer triples (s, p, o), the *ring index* is formed by three columns (Definition 3.3) and their corresponding frequency arrays (Definition 3.7):

(1) The column C_{O} of table SPO and its corresponding array A_{O}.
(2) The column C_{P} of table OSP and its corresponding array A_{P}.
(3) The column C_{S} of table POS and its corresponding array A_{S}.

Note that, when we re-sort the table SPO by column O, we obtain the table OSP (since O is moved to the front), when we re-sort OSP by column P we obtain POS, and when we re-sort POS by S we obtain the table SPO again. From each re-sort, we only keep the *last* column. We now show that we can track a tuple across the three tables, and thus extract the contents of any triple, within only sublinear space on top of the space needed to store the n triples in raw form.

Lemma 3.11. *We can represent a ring on n triples in $3n \log_2 U + o(n \log U)$ bits and retrieve the content of any triple in time $O(\log U)$.*

Proof. We store the columns C_{O}, C_{P}, and C_{S} with wavelet trees. Assume we want to retrieve the content of the ith triple in table SPO. We first obtain $o := C_{\mathrm{O}}[i]$. By Lemma 3.8, we now map the row i to row i' in table OSP with $i' := F_{\mathrm{O}}(i)$, and obtain $p := C_{\mathrm{P}}[i']$. Finally, again using Lemma 3.8, we map the row i' to row i'' in table POS with $i'' := F_{\mathrm{P}}(i')$, and obtain $s := C_{\mathrm{S}}[i'']$. In total we carried out two rank operations and three accesses to sequences using wavelet trees, which takes time $O(\log U)$ as shown in Section 2.3.2.

Note that $i = F_{\mathrm{S}}(i'')$, and therefore we can cycle over the three columns (hence the name ring). Thus, we can equally start from the orders OSP or POS to extract a triple.

Each column requires $n \log_2 U + o(n \log U)$ bits of space, and the arrays A_* take $U \log_2 n$ further bits if represented in plain form. The latter term is also $o(n \log U)$ if $U \in o(n)$. Since there are at most $3n$ different values of s, p, and o, it holds that $U \leq 3n$. If U is not $o(n)$, we can exploit the fact

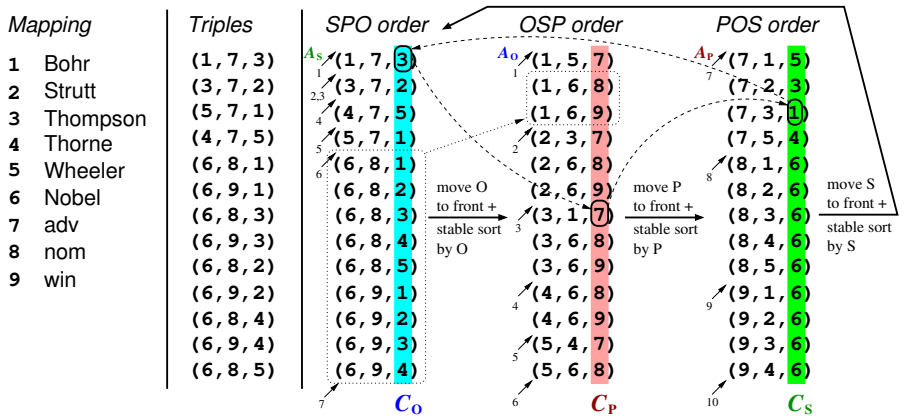


Fig. 6. Our ring index for the graph of Figure 3. The ring is formed by the sequences C_O, C_P, and C_S (the shaded columns), plus the arrays A_O, A_P, and A_S (the small diagonal arrows; $c\nearrow$ pointers between rows mean that $A[c]$ is the index of the upper row). The dashed curves show how we navigate the columns to retrieve the triple $(1, 7, 3)$. The dotted boxes and arrow illustrate a restriction from $C_O[5\,..\,13]$ to $C_P[2\,..\,3]$, by $o = 1$.

that the arrays A_* are nondecreasing to store them in $O(n)$ bits: We represent each array A_* as a bitvector D_* of size $n + U + 1 \in O(n) \subset o(n \log U)$ where we set the bits at positions $A_*[c] + c$, for $c := 1, \ldots, U + 1$. Then we retrieve $A_*[c] = \texttt{select}_1(D_*, c) - c$ in constant time. □

Example. Figure 6 shows the ring index corresponding to the graph of Figure 3. On the left, we map $\text{dom}(G)$ to the interval $[1\,..\,9]$. The second column shows the resulting set of $n = 13$ triples (e.g., (Bohr,adv,Thompson) becomes $(1, 7, 3)$). The next columns show the tables listing the triples sorted in SPO, OSP, and POS orders, highlighting the column we choose in each order to obtain the next one via re-sorting. The ring is formed by the three highlighted columns and their arrays A_*.

The figure shows with dashed curves how we recover the first triple in the SPO order, $(1, 7, 3)$, by starting from position $i := 1$. We know that the object is $C_O[i] = 3$. We then map the position i of column C_O to position i' of column C_P with the formula $i' := F_O(i) = 7$. Let us see this application of $F_O(i = 1)$ in detail. Since $C_O[1]$ is the first 3 in C_O (i.e., $\texttt{rank}_3(C_O, 1) = 1$) and there are $A_O[3] = 6$ values less than 3 in C_O, the triple at position $i = 1$ in table SPO becomes the triple at position $i' = F_O(1) = A_O[3] + \texttt{rank}_3(C_O, 1) = 6 + 1 = 7$ in table OSP, where we have stably sorted SPO by attribute O. We then know that the predicate of the corresponding triple is $C_P[i'] = 7$.

We similarly map the position i' of column C_P to position i'' of column C_S with $i'' := F_P(i') = 3$, to determine that the subject is $C_S[i''] = 1$. In order to map $C_P[i' = 7] = 7$ to C_S, we have $A_P[7] = 0$ and $\texttt{rank}_7(C_P, 7) = 3$, so $i'' = F_P(7) = 0 + 3 = 3$.

Finally, if we compute $i''' := F_S(i'')$ analogously, we obtain again $i''' = i = 1$. □

The ring index can be built from a raw representation of the triples (s, p, o), with a constant amount of stable sorting steps. Since $U \leq 3n$, the sorting can be done in linear time and space. The arrays A_* (or bitvectors D_*) are easily built in linear time as well. Finally, the wavelet trees of the columns C_* can be built in time $O(n \log U / \sqrt{\log n})$ [46], which is in $O(n\sqrt{\log U})$.

THEOREM 3.12. *Let G be a graph with n triples and U different constants. Then the ring index of G uses $3n \log_2 U + o(n \log U)$ bits of space and can retrieve any desired triple in $O(\log U)$ time. The ring index is built in $O(n\sqrt{\log U})$ time within $O(n \log U)$ bits of working space.*

This space is worst-case; Section 3.6 shows that we can make the ring index use space close to a *compressed* representation of G.

3.4 Processing joins

We now describe how to support `leap` over the ring index of a graph in $O(\log U)$ time using the algorithms of Section 3.2. This running time for `leap` implies the worst-case optimality of LTJ (Algorithm 1) over our representation [64]. Thus, in this section we prove the following theorem; working space refers to the space needed to solve Q apart from the space used by the index.

Theorem 3.13. *On a graph G, our ring index evaluates a basic graph pattern Q formed by m triple patterns in $O(Q^* \cdot m \log |\mathrm{dom}(G)|)$ time, where Q^* is the maximum possible output (AGM bound) of such a query on some graph of size $|G|$. The working space of the algorithm is $O(1 + |\mathrm{var}(Q)|)$ words.*

Let $Q := \{t_1, \ldots, t_m\}$ be a basic graph pattern. We assume the constants in each t_i have already been encoded as integers in $[1\,..\,U]$. We further assume that no variable appears more than once in a triple pattern t_i; the other case will be discussed later in Section 3.5. We use $t(G)$ as shorthand for the evaluation of the (singleton) basic graph pattern $\{t\}(G)$ (see Section 2.1). We denote by $G_t := \{\mu(t) \mid \mu \in t(G)\}$ the *occurrences* of t in G, that is, the set of triples in G matching t.

3.4.1 Computing the occurrences of a triple pattern. Because the constants in a triple pattern t are always consecutive when t is regarded as cyclic, there is some attribute order (SPO, OSP, or POS) where the rows matching the constants of t form a contiguous range $[s\,..\,e]$ because they appear in a prefix of the attributes. The following lemma shows how to find $[s\,..\,e]$ in $O(\log U)$ time.

Lemma 3.14. *Let $t := (\alpha, \beta, \gamma)$ be a triple pattern with $0 \le b \le 3$ constants. In $O(\log U)$ time, we can find values s, e such that:*

- *If $|G_t| = 0$, then $s = e = \perp$; otherwise*
- *$e = s + |G_t| - 1$, and $C_x[s\,..\,e]$ refers to the triples in G_t, $x \in \{S, P, O\}$ being the variable (cyclically) preceding a constant in t if $0 < b < 3$; x is arbitrary otherwise.*

Proof. The case for $b = 0$ constants is trivial because the triple pattern matches every triple in G and $[s\,..\,e] := [1\,..\,n]$ covers all the triples in any column.

If $b = 1$, assume w.l.o.g. that the lone constant in t is the bound value d for attribute S; the other two cases are analogous. Then, $[s\,..\,e] := [A_S[d] + 1\,..\,A_S[d+1]]$ because this is the range of all the triples with S $= d$ when they are sorted by attribute S; recall Eq. (1). Since attribute O (cyclically) precedes S, according to Definition 3.10, C_O lists the triples by order SPO, and thus $C_O[s\,..\,e]$ represents G_t (i.e., X = O).

When $b = 2$, let d' and d be the two (cyclically) consecutive constants in t, and assume w.l.o.g. that O and S are the corresponding attributes, that is, $d'd = \gamma\alpha$. In this case, X = P. Since C_O is sorted by order SPO and C_P by order OSP, F_O (Eq. (2)) maps from the order SPO of C_O to the order OSP of C_P.

We first obtain the range $[s\,..\,e]$ for the binding of S to d just as for $b = 1$. Then $C_O[s\,..\,e]$ refers to the triples, sorted by S, where S has value d. From those, we seek the triples where the value of O is d', or which is the same, where d' (cyclically) precedes d, that is, the triples with value d' in $C_O[s\,..\,e]$. Lemma 3.9 shows that these values d' form a range $C_P[s'\,..\,e']$, obtainable with a restriction from $C_O[s\,..\,e]$ in $O(\log U)$ time using the wavelet tree of C_O.

For the last case $b = 3$ we choose any two consecutive attributes, say OS as for $b = 2$. This yields a range $C_P[s'\,..\,e']$ of triples, sorted by O and upon ties by S, where O has value d' and S has value d. An additional restriction on C_P yields the range $C_S[s''\,..\,e'']$ of the triples matching t.

In all cases, we change to $s, e := \perp$ when $s > e$, as this means that t has no occurrences in G. □

Example. The triple pattern (Nobel,?,Bohr) finds out if Bohr won or was nominated for a Nobel prize. This is mapped to $(6, x, 1)$, with variable predicate P. Thus, $d', d = 1, 6$ in the proof of Lemma 3.14. We start from the second component, S, with value $d = 6$. The range $[s\,..\,e] := [A_S[6] + 1\,..\,A_S[6 + 1]] = [5\,..\,13]$ then contains the triples with subject value 6 in table SPO, corresponding to column C_O (see dotted lines in Figure 6). We now perform a restriction (Lemma 3.9) from $C_O[5\,..\,13]$ with value $d' = 1$, which yields $[s'\,..\,e'] := [2\,..\,3]$, now in table OSP and column C_P. That is, $[2\,..\,3]$ are the rows in table OSP where O has value 1 and S has value 6.

We can now obtain the bindings for the predicates p matching the triple. They are in $C_P[2\,..\,3] = \langle 8, 9\rangle$, corresponding to nom and win. That is, Bohr was nominated to, and won, a Nobel prize. □

3.4.2 Supporting leaps. We show how to support $\texttt{leap}(t_i, x, c)$, where t_i is either a triple pattern from Q or one of its progressively bound versions $\mu(\cdot)$ in Algorithm 1.

To evaluate $\texttt{leap}(t_i, x, c)$, we first obtain the values s_i, e_i of Lemma 3.14 for t_i. If $s_i, e_i = \perp$ we return $\perp$. Otherwise, let $t_i := (\alpha_i, \beta_i, \gamma_i)$. How we search for $c_x \geq c$ (see Definition 2.1) depends on where x and the constants in t_i are. If only α_i is a constant and $\beta_i = x$, or only β_i is a constant and $\gamma_i = x$, or only γ_i is a constant and $\alpha_i = x$, then we will find c_x by extending the constant *forwards*, because x (cyclically) follows the part of the triple pattern that is already bound. Otherwise, we will find c_x by extending *backwards* the bound attributes, that is, looking for x preceding the range of attributes that are already bound. We first describe the backward extension, which is simpler.

Backward extension. Let X be the attribute of variable x. When we process a triple pattern $t_i := (\alpha_i, \beta_i, \gamma_i)$ backwards, we have the range $C_X[s_i\,..\,e_i]$ of the triples matching the bound part of t_i, and want to find the smallest $c_x \geq c$ such that some of those bound parts are preceded by c_x, that is, $C_X[k] = c_x$ for some $k \in [s_i\,..\,e_i]$. Equivalently, we want to find the smallest $c_x \geq c$ in $C_X[s_i\,..\,e_i]$. This corresponds to the *range-next-value* operation discussed at the end of Section 2.3.2, which is supported in $O(\log U)$ time by the wavelet tree of C_X.

Forward extension. When only one of α_i, β_i, or γ_i is bound, and the attribute X of the variable x we seek is not reached in one step backwards, we rather process $(\alpha_i, \beta_i, \gamma_i)$ forwards. Let w.l.o.g. P be the bound attribute (say, to value p), which (cyclically) precedes X = O. By Lemma 3.14, the triples in G_{t_i} are in a range $C_S[s_i\,..\,e_i]$ (C_S is sorted by POS = PXS). Since only attribute P is bound, the range corresponds to the triples where P has value p. We must thus find the smallest $c_x \geq c$ that follows a p in one of the triples in the range $C_S[s_i\,..\,e_i]$.

To do this, we begin by finding the first occurrence of p in the range $C_P[A_X[c] + 1\,..]$ (C_P is sorted by OSP = XSP), corresponding to the triples where attribute X has a value of c or more.[2] We perform a restriction (Lemma 3.9) from this range by symbol p: letting $q := A_P[p] + \texttt{rank}_p(C_P, (A_X[c] + 1) - 1) + 1 = A_P[p] + \texttt{rank}_p(C_P, A_X[c]) + 1$, we obtain the range $C_S[q\,..\,e_i]$ of the triples where P has value p and X has a value $c_x \geq c$. We wish to find the first such triple, at $C_S[q]$, which has the smallest possible value for c_x. We now map the triple back to column C_P with $r := F_S^{-1}(q)$ (Eq. (3)).

This is the position in C_P of the first triple (in order OSP = XSP) where attribute X has a value $c_x \geq c$ and is preceded by p. To find the actual value of c_x, we apply a binary search for r in A_X, looking for $A_X[c_x] < r \leq A_X[c_x + 1]$. As mentioned in Lemma 3.11, A_X might be stored as a bitvector D_X to save space on large alphabets. In this case the binary search is replaced by the constant-time formula $c_x := \texttt{select}_0(D_X, r) - r$. See Figure 7.

Examples. To exemplify the backward extension, consider the basic graph pattern formed by the triple patterns (Bohr,adv,?x) and (Nobel,win,?x), to find Nobel prize winners advised by Bohr.

[2]This is indeed easily done as $r := \texttt{select}_p(C_P, \texttt{rank}_p(C_P, A_X[c]) + 1)$, which can be shown to be equivalent to the method we describe. This shortcut, however, cannot be generalized to the dimensions over 3 we consider from Section 6 onwards.

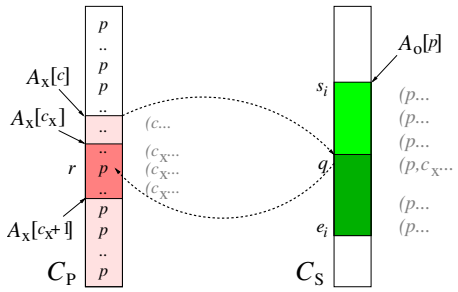


Fig. 7. Illustration of the forward extension algorithm. The strings in gray represent the way triple contents start in that area of the columns. The dashed arrows represent the computation of q and r

After mapping them to $(1, 7, x)$ and $(6, 9, x)$ and applying Lemma 3.14 to both, we obtain the ranges $C_O[1\,..\,1]$ for the former and $C_O[10\,..\,13]$ for the latter. To bind the variable x, we find the first value, $c := 3$, in $C_O[1\,..\,1]$, and now need to find the smallest value $c_x \geq c$ in $C_O[10\,..\,13]$. The wavelet tree yields the solution $c_x := 3$, which then is a binding for x (indeed, there are triples $(1, 7, 3)$ and $(6, 9, 3)$ in the graph). This corresponds to the object Thompson.

To exemplify the forward extension, assume that we extend our previous basic graph pattern with the triple pattern $(y, 8, x)$ in order to also find out all awards the advisee x was nominated to. This triple pattern is processed applying Lemma 3.14 to obtain the range $C_S[5\,..\,9]$ of the triples starting with 8 in table POS. The binding $x := 3$ of the previous paragraph must also be found in this range, so we must determine the smallest $c_x \geq 3$ that follows 8 in the rows $[5\,..\,9]$ of table POS. To do this, we find the first occurrence of 8 in $C_P[A_O[3] + 1\,..] = C_P[7\,..]$, with $q := A_P[8] + \mathsf{rank}_8(C_P, 7 - 1) + 1 = 4 + 2 + 1 = 7$ (the restriction formula) and $r := F_P^{-1}(7) = 8$. Finally, we find that $c_x = 3$ because $A_O[3] = 6 < r \leq 9 = A_O[4]$. Thus, we confirm the binding $x := 3$ (Thompson). The algorithm will later find the bindings for y using backward extension. □

Lemma 3.15. *Let G be a graph, t be a triple pattern, x be a variable that appears exactly once in t, and $c \in U$ be a constant. Then the ring index of G supports* `leap`(t, x, c) *in $O(\log U)$ time.*

Finally note that, except for μ, the working space used by `seek` in Algorithm 1 is constant; the same holds for `leap` and leapfrog_search. We can further maintain μ in constant space per recursive call by storing only the last assignment and pointing to the previous one in the stack. Thus, the working space of Algorithm 1 is $O(v + 1)$, since the maximum stack height is $v = |\mathrm{vars}(Q)|$.

For simplicity, we have focused on the resolution of basic graph patterns only. Our ring supports some additional enhancements, which we discuss in Appendix A.

3.5 Variables appearing more than once in a triple pattern

The case when a variable x has more than one occurrence in a triple pattern must be dealt with differently, as every binding of this variable implies working on two columns of the ring. Let us first note that when a variable x has more than one occurrence in a triple pattern t, these are (cyclically) consecutive. Thus, this means that we can support `leap`(t, x, c) on our unmodified index, by processing t backwards. First, we consider only the second occurrence (which is just behind the matched part of the triple pattern, if any). Every time we find a binding $x := c_x$, we perform a second restriction on the triple, with c_x. If the resulting range is nonempty, then the binding is valid and we recurse; otherwise we just set $c := c_x + 1$ and restart the search. The case of triple patterns formed by three occurrences of the same variable is analogous. The problem is that this algorithm does not run in $O(\log U)$ time, and would affect the optimality of LTJ.

To support `leap` in $O(\log U)$ time, we can instead split G into five graphs of n triples in total: G_{xyz} contains the triples (s, p, o) where s, p, and o are all different, G_{xxy}, G_{xyx}, and G_{yxx} contain the triples where $s = p$, $s = o$, and $p = o$, respectively, and the other component is different, and G_{xxx} contains the triples where $s = p = o$. We then create five rings, one per graph.

At query time, the occurrences of each partially bound triple pattern t_i span five intervals $C_*[s_i \,..\, e_i]$, one per ring. However, triple patterns t_i containing two copies of the same variable have two intervals only: one in the ring of G_{xxx} and the other in the ring of G_{xxy}, G_{xyx}, or G_{yxx}, depending on where the variable appears. Finally, a triple pattern t_i with three copies of the same variable has an interval in the ring of G_{xxx} only. This ensures that for the aforementioned algorithm handling multiple occurrences of a variable, every match $x := c_x$ we find is valid.

Triples that span several intervals $C_*[s_i \,..\, e_i]$ implement `leap` by searching in all of the intervals and taking the minimum $c_x \geq c$ found. This introduces a constant-time overhead factor but retains worst-case optimality. Furthermore, the space of the data structure is the same. On graphs where the edge labels are disjoint from the node labels, we need only consider G_{xyz} and G_{xyx}, and if they have no self-loops then no triple pattern with a variable appearing twice can match.

3.6 Rings in compressed space

We have shown that the ring index represents G using just $o(|G|)$ space on top of the space $|G|$ needed by its raw representation as an array of triples, that is, $3n \log_2 U + o(n \log U)$ bits. We now show that the space can indeed be made close to the size of a *compressed* representation of G. Precisely, if the sets of different subjects, predicates, and objects are S, P, and O, respectively, then the wavelet matrices for the columns C_{S}, C_{P}, and C_{O}, will use $n(\log_2 |S| + \log_2 |P| + \log_2 |O|)(1 + o(1))$ bits, and we will use $3n + o(n)$ further bits if we use bitvectors D_* instead of the arrays A_*. We obtain compression by representing the bitvectors of the wavelet matrices in compressed form.

Concretely, we will use a compressed bitvector representation [58] that, given a parameter b, cuts the bitvector into chunks of b bits and represents each chunk as a pair (c, o), where c is the number of 1s in the chunk and o is an identifier of that chunk among those having c 1s. The chunk is then represented using $\lceil \log_2(b+1) \rceil + \lceil \log_2 \binom{b}{c} \rceil$ bits. This is advantageous when c is near zero or near b (i.e., the chunk has many 0s or many 1s).

Consider our column C_{O}; the others are analogous. Since it contains the objects in SPO order, the objects o associated with the same subject-predicate pair sp will appear in increasing order. Those increasing ranges in $C_{\mathrm{O}}[s \,..\, e]$ induce runs of 0s and 1s in the bitvectors of the wavelet matrices; recall Section 2.3.2. In particular, in the first level, all the elements smaller than $|O|/2$ precede the elements larger than $|O|/2$, and therefore the root bitvector B has a run of 0s preceding a run of 1s in $B[s \,..\, e]$. Those runs are compressed well as pairs (c, o), as explained. When the range $B[s \,..\, e]$ is split in the left and right children, both new ranges also have a run of 0s preceding a run of 1s.

In Figure 6, for example, we can see an increasing range $C_{\mathrm{O}}[5 \,..\, 9]$, corresponding to objects associated with $s = 6$ and $p = 8$, and another range $C_{\mathrm{O}}[10 \,..\, 13]$ associated with $s = 6$ and $p = 9$. A second effect we can see in the figure is the run of values 6 in $C_{\mathrm{S}}[5 \,..\, 13]$, which owes to the predictability of the triples, associated with the regularity of the graph: if the predicate is 8 or 9 `–nom` or `won–` then the subject is 6 `–Nobel–`. Long ranges with the same symbol also induce runs of 0s and 1s in the wavelet tree bitvectors, which are in addition not split in subsequent levels.

In fact, our ring index can be regarded as a kind of *column store* [60]. Column stores encode each column of a table separately. Each column may reorder the rows to improve compression, but then must include an extra pointer per row in order to link the corresponding rows across columns. Our particular column order based on stable sorting, instead, boosts compression via grouping by the

other two components, but it does not require storing any pointer; the rows can be tracked using the F formula of Eq. (2) (we also need the bitvectors D_*, but those use just $O(1)$ bits per entry).

In this regard, our ring index is related to the proposal of Vo and Vo [65], who choose a column order $C_1, \ldots, C_d$ and, for each new column C_j, choose an ordered subset of previous columns, $C_{i_1}, \ldots, C_{i_k} \in \{C_1, \ldots, C_{j-1}\}$, sort the rows lexicographically by $C_{i_1} \cdots C_{i_k}$, and write C_j in that order. Our ring index sorts each column according to the sequence of the other $d-1$ columns (cyclically) preceding it. An important difference is that the scheme of Vo and Vo works only for compression; the whole table must be decompressed, column by column, completely reordering the rows to decode each new column, before the table can be put to use. Our ring, instead, can retrieve any desired triple, and even support wco joins, without ever having to decompress the data.

4 Engineering and implementation

We describe in this section the most relevant engineering and implementation aspects of our ring index. Our implementation is publicly available at https://github.com/adriangbrandon/ring.

4.1 Indexing

We implemented the ring index in C++11 over the succinct data structures library, SDSL (available at https://github.com/simongog/sdsl-lite) [27]. Because the alphabets are generally large, we represent the sequences C_* with wavelet matrices [16].

We provide two flavors of indexes, with the bitvectors of the wavelet matrices stored either in plain or in compressed form. Plain bitvectors are faster in practice. The compressed bitvectors use a parameter b in SDSL; larger values for b offer better compression but slower operations. Compressed bitvectors store no `select` data structures; they handle this operation via binary searches on `rank`.

In order to reduce the size of the universe, the sequence C_P of predicates uses its own alphabet, typically much smaller than $[1\,..\,U]$. The sequences C_S and C_O of subjects and objects, instead, share a common alphabet. Reducing the alphabet size in this way improves both space and operation time for the wavelet matrices at the cost of not directly supporting joins across the alphabets, that is, between predicates and subject/objects. If such joins were needed (they do not appear in our benchmark), a simple solution would be to have a common alphabet for all columns. To retain good time and space one could still use local alphabets and add three small mapping bitvectors of length U, M_S, M_P and M_O, each marking with 1s the symbols that do appear in their column. With `rank`/`select` on the bitvectors we would map between global and local alphabets.

In addition, only for C_P, which has a smaller alphabet, we store the `select` data structures for handling forward extensions as described in Section 3.4.2. On the other two sequences, we find c_x in a different way. Instead of computing $q := F_\mathrm{A}^{-1}(p)$ and then $c_x := \mathtt{select}_0(D_\mathrm{X}, q) - q$, which requires supporting `select` on C_S and C_O (Eq. (3)), we use the circularity of the triples to reach the same point with only `rank` and `access` operations, which is faster in practice on C_S and C_O: we compute $r := F_\mathrm{O}(p)$, which maps $C_\mathrm{O}[p]$ to $C_\mathrm{X}[r]$, and then $c_x := C_\mathrm{X}[r]$ (note it holds $q = F_\mathrm{X}(r)$).

The mapping from $\mathrm{dom}(G)$ to consecutive integers is done by sorting the triples by predicate, and then hashing subjects and objects to ensure uniqueness (the sorting aims to create longer runs of 0s and 1s in the bitvectors for the compressed version of the ring; recall Section 3.6). Finally, we use plain bitvectors D_* instead of arrays A_* to store cumulative symbol frequencies in C_*. We use `stl::sort` and `stl:stable_sort` standard C++ sorting algorithms to produce columns C_*. According to our experiments, the sorting steps represent only a small fraction (about 10%) of the whole construction process. The most expensive task is that of building the wavelet matrices, for which we use the standard `sdsl::construct_im` construction process.

4.2 Join algorithm

We implement Algorithm 1 with some improvements on the description of Section 3.4.

First, for each triple pattern t_i (in its original form or with further variables bound by a mapping), we maintain the values s_i, e_i instead of computing them from scratch during each `leap`. More precisely, we find s_i, e_i at the beginning of `leapfrog_search` as described in Lemma 3.14, and then update them after each further binding of t_i. The working space then rises to $O(mv)$, which is still very low. We update the range as follows. When extending backwards, the *range-next-value* operation not only yields c_x, but also the values $s := \mathsf{rank}_{c_x}(C_{\mathrm{X}}, s_i - 1) + 1$ and $e := \mathsf{rank}_{c_x}(C_{\mathrm{X}}, e_i)$; recall Section 2.3.2. With those values, we immediately have the new range $C_{\mathrm{Y}}[s'_i \,..\, e'_i]$ (where Y is the attribute that cyclically precedes X), with $s'_i := A_{\mathrm{X}}[c_x] + s$ and $e'_i := A_{\mathrm{X}}[c_x] + e$, which completes the restriction according to Lemma 3.9. When extending forwards from attribute, say, P bound to d (so the current range is $C_{\mathrm{S}}[s_i \,..\, e_i]$), once we obtain the value c_x, we perform a restriction by symbol d from $C_{\mathrm{P}}[A_{\mathrm{X}}[c_x] + 1 \,..\, A_{\mathrm{X}}[c_x + 1]]$ so as to obtain the final range $C_{\mathrm{S}}[s'_i \,..\, e'_i]$, using Lemma 3.14.

Example. Continuing our example of Section 3.4.2, in the backward extension we had obtained $c_x := 3$ for both $C_{\mathrm{O}}[1 \,..\, 1]$ and $C_{\mathrm{O}}[10 \,..\, 13]$. We can find the ranges in C_{P} for both, though in this case the ranges will have only one tuple. For example, for $C_{\mathrm{O}}[10 \,..\, 13]$, the *range-next-value* algorithm returns $c_x := 3$ and also $s := \mathsf{rank}_3(C_{\mathrm{O}}, 9) + 1 = 3$ and $e := \mathsf{rank}_3(C_{\mathrm{O}}, 13) = 3$. Thus the range is $C_{\mathrm{P}}[A_{\mathrm{O}}[3] + s \,..\, A_{\mathrm{O}}[3] + e] = C_{\mathrm{P}}[9 \,..\, 9]$.

In the forward extension, once we know that $c_x = 3$, we have the range $C_{\mathrm{P}}[A_{\mathrm{O}}[3] + 1 \,..\, A_{\mathrm{O}}[4]] = C_{\mathrm{P}}[7 \,..\, 9]$. With a restriction from this range by $d = 8$ we obtain $C_{\mathrm{S}}[7 \,..\, 7]$, the restriction of the original range $C_{\mathrm{S}}[5 \,..\, 8]$ that contains the rows of table POS that start with $p = 8$ and $o = 3$. □

The second optimization is to handle the *lonely variables* [33], that is, variables that appear in only one triple pattern t_i, in a different way: once the other variables of t_i have been bound, we report all the possible bindings of our ranges. From the current values s_i, e_i, we bind the remaining variables backwards, one by one. For each variable x, this corresponds to finding all the distinct values in $C_{\mathrm{X}}[s_i \,..\, e_i]$, which is done as described at the end of Section 2.3.2. Every returned value c is a valid binding $x := c$. The updated range is $s'_i := A_{\mathrm{X}}[c] + s_c$, $e'_i := A_{\mathrm{X}}[c] + e_c$, s_c and e_c being the values returned by the algorithm.

Example. Finishing our example, the variable y in the triple pattern $(y, 8, x)$ is a lonely variable. Once we have bound $x := 3$, we want to report all the values found in $C_{\mathrm{S}}[7 \,..\, 7]$. In this case we only find the value 6 (Nobel), indicating that Thompson "only" won the Nobel prize (in our database). □

Lonely variables actually permit storing a more compact representation of the output, by storing the entire ring interval instead of individually bounding the lonely variable for each element in this interval, as in a limited form of a factorized database [55]. For fairness with the other implementations, this feature is not included in the experimental section.

4.3 Variable elimination order

The running time of LTJ can often sharply depend on selecting a good order in which variables are eliminated [33]. It turns out that our ring index can also be used to provide relevant statistics on the fly, computed in logarithmic time, and without additional profiling.

Recall that given a triple pattern t_i, our index quickly computes the initial ranges $C_*[s_i \,..\, e_i]$, so that the number of triples matching the pattern is exactly $e_i - s_i + 1$. We use this information to construct the following elimination order for variables that appear in more than one triple pattern. First we compute the *selectivity* of each triple pattern t_i as $s(t_i) := e_i - s_i$. We then estimate the selectivity of each variable x as $s_{\min}(x) := \min_{t \in Q_{\{x\}}} s(t)$. The variables x are then bound by increasing order of $s_{\min}(x)$, that is, from most to least selective. We however modify this order to

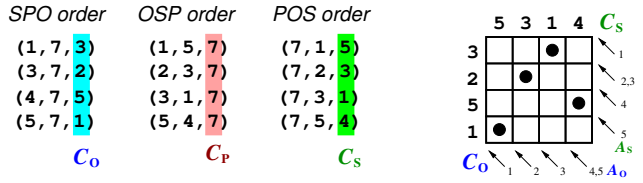


Fig. 8. The binary relation resulting from restricting the ring index of Figure 6 to $p = 7$.

ensure that, if possible, each new variable shares a triple pattern with some previous one. That is, the next variable to bind is the least selective one among those sharing a triple pattern with an already bound variable; if none exist, we just take the least selective variable. This rule is consistent with leaving the lonely variables to the end.

This scheme is similar to the one used in Jena LTJ [33], which also leaves lonely variables until the end, but both schemes feature some key differences. In the case of Jena LTJ, the standard Jena TDB optimizer is first used to induce an order among triple patterns. This optimizer uses relatively coarse-grained statistics precomputed from the dataset; more specifically, the statistics indicate how many triples each unique predicate is associated with. We rather use the size of the range, which is efficient to compute with the ring, which avoids the need to precompute statistics, and which provides much more fine-grained cardinality information.

4.4 Fixed predicates

A particular practical case where we can considerably improve our ring index is where there is a relatively small set $\mathcal{P}$ of predicates, and the triple patterns always have a constant predicate. Formally, the triples range over a set $\mathcal{U} \times \mathcal{P} \times \mathcal{U}$, where $|\mathcal{P}| := P \ll U = |\mathcal{U}|$, and the triple patterns range over $(\mathcal{V} \cup \mathcal{U}) \times \mathcal{P} \times (\mathcal{V} \cup \mathcal{U})$. Such triple patterns are very common in practice, forming the basis of more complex graph patterns used in real-world queries [12]. Furthermore, the fixed-predicate case is also a natural way to model property graph databases (see e.g. [4, 23]), another important graph database model that is widely used in industry.

In this case we can reduce the space of the ring while retaining its functionality and wco guarantees by storing a different sub-ring per predicate $p \in \mathcal{P}$ for all triples with predicate p.

Figure 8 shows the sub-ring corresponding to $p = 7$ obtained from our example of Figure 6. It is immediately obvious that we do not need to store C_P because it is a sequence of n copies of p. What is less obvious is that the remaining columns, C_S and C_O, form a sub-ring of length 2, and as such they are essentially the inverse permutation of each other, according to function F_j. Therefore, we need to store only *one* column, and can simulate the work of the other via the wavelet tree.

More precisely, note that $i' := F_O(i)$ maps the position i of C_O to position i' of C_P. Now, the re-sorting by P is immaterial, so C_S has the same ordering of C_P, meaning that $i' := F_O(i)$ maps from C_O to C_S. On the other hand, $i := F_S(i')$ maps back from C_S to C_O, thus F_O and F_S are inverse permutations. This relation can be laid on an $n \times n$ grid, where the rows represent C_O and the columns represent C_S. The grid, illustrated on the right of Figure 8, has exactly one point per row and per column: the points in format (row,column) are $(i, F_O(i))$ for $1 \le i \le n$, or equivalently, $(F_S(i'), i')$ for $1 \le i' \le n$. Such a grid can be represented with a wavelet tree (Section 2.3.2) in either direction; let us choose to represent the sequence of values $F_S(1) \cdots F_S(n)$ for concreteness.

Since the predicate is always fixed on the triple patterns, each one will be mapped to the appropriate sub-ring. During the process, the partially bound triple patterns will correspond to a range in C_O or in C_S, that is, a range of rows or columns in the grid. The operations we need to carry out Algorithm 1 on this grid can be interpreted in geometric terms. Fortunately enough, the

wavelet tree supports a number of geometric primitives in $O(\log n)$ time [47]. The following ones suffice to cover the required operations [9], where by $[\alpha, \beta] \times [x, y]$ we denote all (row,column) index pairs of the form (γ, z) such that $\alpha \leq \gamma \leq \beta$, and $x \leq z \leq y$, that is, the sub-grid formed by intersecting rows α to β and columns x to y:

- $\mathsf{rel\_num}(\alpha, \beta, x, y)$ counts the number of points in $[\alpha, \beta] \times [x, y]$.
- $\mathsf{rel\_min\_obj\_maj}(\alpha, \beta, x)$ returns the leftmost point in $[\alpha, \beta] \times [x, n]$.
- $\mathsf{rel\_min\_lab\_maj}(\alpha, x, y)$ returns the highest point in $[\alpha, n] \times [x, y]$.

With those primitives, we can carry out the operations required, as follows:

- For restrictions (Lemma 3.14) we need operations rank. We reduce $\mathsf{rank}_c(C_\mathrm{s}, i)$ to the operation $\mathsf{rel\_num}(A_\mathrm{s}[c] + 1, A_\mathrm{s}[c + 1], 1, i)$, and $\mathsf{rank}_c(C_\mathrm{o}, i)$ to $\mathsf{rel\_num}(1, i, A_\mathrm{o}[c] + 1, A_\mathrm{o}[c + 1])$.
- For backward extensions (Section 3.4.2), we need the *range-next-value* operation. Finding the smallest $c_x \geq c$ in $C_\mathrm{o}[s\,..\,e]$ reduces to computing $t := \mathsf{rel\_min\_obj\_maj}(s, e, A_\mathrm{o}[c] + 1)$ and then retrieving c_x such that $A_\mathrm{s}[c_x] < t \leq A_\mathrm{s}[c_x + 1]$. Similarly, finding the smallest $c_x \geq c$ in $C_\mathrm{s}[s\,..\,e]$ reduces to computing $t := \mathsf{rel\_min\_lab\_maj}(A_\mathrm{s}[c] + 1, s, e)$ and then retrieving c_x such that $A_\mathrm{o}[c_x] < t \leq A_\mathrm{o}[c_x + 1]$.
- We do not use forward extensions; we can always proceed backwards in our sub-rings.
- For lonely variables, we need to list all the distinct values in $C_\mathrm{o}[s\,..\,e]$ or $C_\mathrm{s}[s\,..\,e]$. This can be done with consecutive *range-next-value* operations, thus retrieving the bindings in order.

Our data structure then solves queries Q with fixed predicates in time $O(Q^* \cdot m \log n)$. Its space is $\sum_{p \in \mathcal{P}} \left(n_p \log_2 n_p + o(n_p \log n_p) + O(U \log n_p)\right)$ bits, where n_p is the number of triples in the sub-ring of p (the first two terms are for the wavelet trees and the third for the A_* arrays). This is at most $n \log_2 n + o(n \log n) + O(PU \log(n/P))$ bits. By using bitvectors D_* instead of arrays A_*, the third term becomes $O(PU)$, and by using compressed bitvectors for D_*, it becomes $O(n \log(PU/n)) \subseteq O(n \log P)$.

5 Experimental results

We now compare our system – running a modified version of LTJ over a ring index – versus state-of-the-art alternatives in terms of the space used for indexing and the time for evaluating basic graph patterns. We expect that our system will use less space than non-compact alternatives, and that it will use less time for evaluating queries than non-wco alternatives, while remaining competitive with wco alternatives. We further compare compressed and uncompressed variants of the ring, where we expect the compressed variant to use less space but to have slower query times.

We run two benchmarks over the Wikidata graph [66], which we choose for its scale, diversity, prominence, data model (it has labeled edges) and real-world query logs [11, 40]. The first benchmark is the Wikidata Graph Pattern Benchmark (WGPB) proposed by Hogan et al. [33] for a sub-graph of Wikidata, with diverse abstract graph patterns. The goal of this benchmark is to study how the indexes handle queries with different topologies. The second benchmark evaluates real-world graph patterns extracted from Wikidata query logs at full scale, and should be a good predictor of how they systems will perform in real applications.

Further details for reproducing our experiments are given in Appendix B.

5.1 Experimental setup

Our experiments compare various in-memory databases using wco algorithms, as follows.

Ring and C-Ring: LTJ running over our ring index using plain and compressed bitvectors, respectively. The latter uses parameter $b := 15$. The system operates in main memory.

EmptyHeaded: An implementation [1] of a more general algorithm than LTJ, which processes queries according to a *generalized hypertree decomposition (GHD)* [29], which produces a tree

where each node is a cyclic subquery, then uses the wco algorithm NPRR [53] to process each of these subqueries, and finally joins these intermediate results using Yannakakis' algorithm [68]. Triples are stored as 6 different tries (all orders) in main memory.

Graphflow: A graph query engine that indexes property graphs using in-memory sorted adjacency lists and supports hybrid plans blending wco and pairwise joins [42].

Qdag: The only previous succinct wco index [49], based on a quadtree representation of the graph that runs in main memory.

We disregard other compressed graph indexes [3, 13] that support only single triple patterns or pairwise joins. For reference we further include results for prominent graph database systems, one of which (Jena LTJ) implements a wco join algorithm:

Blazegraph: The graph database system [61] hosting the official Wikidata Query Service [40]. We run the system in triples mode wherein B+-trees index three orders: SPO, POS, and OSP. The system supports nested-loop joins and hash joins.

Jena: A reference implementation of the SPARQL standard. We use the TDB version, with B+-trees indexes in three orders: SPO, POS, and OSP. The system supports nested-loop joins.

Jena LTJ: An implementation [33] of LTJ on top of Jena TDB. All six different orders on triples are indexed in B+-trees.

RDF-3X: The reference scheme [51] that indexes a single table of triples in a compressed clustered B+-tree. The triples are sorted so that those in each B+-tree leaf can be differentially encoded. RDF3X also manages aggregated indexes SP, PS, SO, OS, PO, and OP, which store the number of occurrences of each pair in the dataset. RDF-3X handles triple patterns by scanning ranges of triples and uses a query optimizer based on pairwise joins.

Virtuoso: A widely used graph database hosting the public DBpedia endpoint, among others [19]. It provides a column-wise index of quads with an additional graph (G) attribute, with two full orders (PSOG, POSG) and three partial indexes (SO, OP, GS) optimized for patterns with constant predicates. The system supports nested loop joins and hash joins.

We also include two non-wco relational database systems (alongside EmptyHeaded, which is relational, but is included above as an in-memory wco system):

DuckDB: An in-memory relational database [57] focusing on OLAP workloads using vectorized execution. We load the graph into a single ternary relation using default indexes (experiments with further indexes did not improve performance) and multi-threaded execution.

Postgres: A popular relational database. We used version 13 and loaded the integer-encoded graph into a single ternary relation, adding a primary key B-tree index on SPO, and two auxiliary B-tree indexes on PO and OS.

We run our experiments on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores, 15 MB of cache, and 96 GB of RAM. Our code was compiled using `g++` with flags `-std=c++11`, `-O3`, and `-msse4.2`. Systems are configured per vendor recommendations. All queries are run with a timeout of 10 minutes and a limit of 1000 results (as originally proposed for WGPB [33]).

5.2 Graph patterns benchmark

We first run the Wikidata Graph Pattern Benchmark (WGPB) [33], which uses a Wikidata subgraph with $n = 81{,}426{,}573$ triples, 19,227,372 subjects, 2,101 predicates, and 37,641,486 objects. The benchmark provides 17 query patterns of different widths and shapes, including cyclic and acyclic queries, as shown in Figure 9. Each pattern is instantiated with 50 queries built using random walks such that the results are nonempty. All predicates are constant, all subjects and objects are variables, and each variable appears at most once in the same triple pattern. This benchmark is intended to understand the performance of the indexes on different abstract patterns, more than

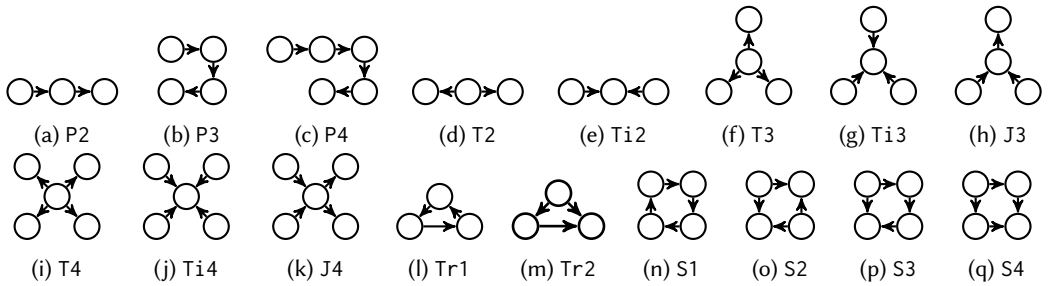


Fig. 9. Query patterns for the Wikidata benchmark

extracting general conclusions about their performance on a real-life scenario. Section 5.3 aims to answer this second question using a larger dataset and more general queries from a real query log.

5.2.1 Indexing and space. There are 4,869,562 identifiers that are both subjects and objects in the graph, so we use a common alphabet for both of size $SO := 51{,}999{,}296$. Our ring index, in either variant, was built in 6.7 minutes, that is, at a rate of 12.1 million triples per minute. The working space used for building the index was 2.2 GB, 2–3 times the size of the plain representation of the triples (which uses 932 MB) and of the largest variant of the resulting ring, 867 MB.

Regarding the space used by the ring, a simple representation of the dataset using 32-bit integer values requires 12 bytes per triple. A *packed* representation requires $\lceil \log_2 SO \rceil + \lceil \log_2 P \rceil + \lceil \log_2 SO \rceil = 26 + 12 + 26 = 64$ bits, or 8 bytes, per triple. Our ring index with plain bitvectors privileges time performance, and thus its `rank` and `select` structures pose a 30% space overhead, which adding the bitvectors D_* sums up to 11.16 bytes per triple, still less than the size of the simple representation. Our ring index using compressed bitvectors with $b := 15$ requires 6.68 bytes per triple, less than the packed representation. The reason is related, but not precisely the same, as those discussed in Section 3.6: the average number of triples per single combination of *sp*, *po*, or *os* is just 1.20 and the average run length is just 1.34. Instead, the identifier assignment technique mentioned in Section 4.1 leads to having nondecreasing runs of average length 4.23 in the sequences C_*.

As explained, our index replaces the representation of the triples, because one can obtain any desired triple from the index. The time to retrieve any arbitrary triple is 5 microseconds with plain bitvectors and 20 microseconds with compressed bitvectors.

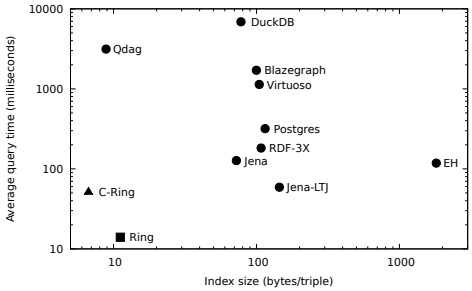
Table 2 compares the space of the indexes, showing that the uncompressed Ring uses 6.5–162 times less space than non-compact indexes, and 13–162 if we consider only the wco non-compact indexes. The only index using less space than Ring is Qdag, which is also succinct and wco. C-Ring, our compressed Ring variant, however, uses 60% of the space of Ring and 75% of the space of Qdag.

Graphflow failed to index the graph both on the experimental machine (with 96 GB of RAM), and another machine on which it was assigned 680 GiB (~730 GB) of Java heap space. Reviewing the source code, Graphflow loads in-memory adjacency lists with $P \cdot SO$ arrays of 32-bit integers, thus requiring $\Omega(P \cdot SO)$ space. Hence it was not feasible to load the Wikidata graph for which $P = 2{,}101$ and $SO = 51{,}999{,}296$. The system rather targets property graphs with few edge labels (unique predicates) and does not support queries with node identifiers (constant subjects and objects, as needed for our second set of experiments, described presently).[3] We conclude that Ring and C-Ring occupy much less space than Graphflow, but comparison of query runtimes was not possible for the selected experiments due to insufficient RAM.

[3]From personal communication with the first author of the Graphflow paper [42].

Table 2. Index space, counting data plus indexes in bytes per triple, on the Wikidata sub-graph, and average query time on WGPB data and queries, in milliseconds. Note the logscales in the plot. The asterisk for EmptyHeaded indicates it cannot limit the output to 1,000 results.

System	Space	Time
Ring	11.16	14
C-Ring	6.68	52



System	Space	Time
DuckDB	77.89	6,886
EmptyHeaded	1,809.84	118*
Graphflow	>8,966.90	—
Qdag	8.86	3,141
Blazegraph	99.86	1,709
Jena	72.32	127
Jena LTJ	144.64	59
Postgres	114.93	318
RDF-3X	107.65	182
Virtuoso	104.49	1,135

Given the reduced space obtained by C-Ring, one may regard it as a compressed representation of the graph, which additionally supports random access to the triples and even wco joins. A server may use it for, upon a client request, selecting a subgraph of G and send it in this format for further querying on the client's side, without any need of the client decompressing it before querying. We explored further compressing it by setting $b := 63$, where the space decreases to 5.35 bytes per triple but the time to extract a tuple raises to 73 microseconds.

5.2.2 Query times. Table 2 gives the average time taken by each index to sequentially evaluate all the queries. These global results are meant to serve as a rough guide only because the queries are synthetic and their frequency in real query logs are not uniform (we will consider real query logs in Section 5.3). With these warnings, we still see that Ring features an excellent performance, both compared to other algorithms and to prominent database systems.[4] C-Ring is almost 4 times slower than Ring, but it is the smallest index, using 60% of the Ring space while still running faster than most other indexes. In particular, it is 60 times faster than Qdag, the next smallest index.[5]

This query log is intended for an analysis of query time distributions by query shape, as shown in Figure 10. For visibility, the plots omit Jena, which was always slower than Jena LTJ, showing the benefits of wco joins (per Table 2, Jena LTJ is about twice as fast overall as Jena). Postgres and DuckDB are also omitted as they fall out of the plots in the current scale (despite the average of Postgres being comparable to the remaining systems).

Ring is the best, or near the best, in all the acyclic queries. Qdag is the best, or near the best, in all the queries with just three variables and some cyclic ones of four (`S1` and `S2`). Yet, it has serious problems with the queries of five variables and some acyclic ones of four (`J3` and `Ti3`). EmptyHeaded is the best, or near the best, in all the cyclic queries and in one acyclic query (`Ti2`),

[4]EmptyHeaded does not support limiting the number of results to 1,000, so it is not fair to compare its 118 milliseconds per query in Table 2 with the others, which report only up to 1,000 results (and still EmptyHeaded outperforms most of them). When running Ring without limiting the number of results, we obtain 26 milliseconds per query, still 4.5 times faster than EmptyHeaded. The absence of limits for EmptyHeaded is not noticeable in Figure 10, because it shows only quantiles and very few queries produce more than 1,000 results.

[5]Qdag does not handle constants in the triple patterns. Since these queries have constant predicates, we use a Qdag to index one binary relation per predicate. This is done in order to obtain query times; the space we report refers to indexing the triples. In Section 5.2.3, instead, we report the space they use when indexing each predicate separately.

but it has very bad cases for long paths (P4) and some tree-like queries (T3, T4, and J4). While Ring is clearly slower than the best performers on the cyclic queries, it offers much more stable times than the bad cases of Qdag and EmptyHeaded, never exceeding 0.05 seconds.

Comparing Ring with Qdag, the latter uses an encoding of the output that grows exponentially with the number of nodes in the patterns [49], so its decreasing performance on larger queries is expected. Comparing Ring with EmptyHeaded, we speculate that the advantage of Ring in acyclic queries is mostly due to the lonely variables optimization. To be more precise, consider queries in T4, Ti4, and J4. In the case of Ring, once the central variable connecting the other four variables is eliminated, the remaining variables are subsequently eliminated using the lonely variables optimization that enumerates their values faster using the wavelet tree functionality (see Section 4.2). EmptyHeaded rather processes acyclic queries using a version of the traditional Yannakakis' algorithm [68], which we speculate is not so well optimized for simple tree-like queries or long paths that may give rise to multiple lonely variables at the end.[6]

DuckDB featured two errors and six additional timeouts of more than 600 seconds; in the results of Table 2, we exclude the errors and count the timeouts as 600 seconds, giving a lower bound for its times. When unable to fit intermediate results in main memory; the system spills data to disk. In two cases, the spill exceeded the free space available on the disk (approx. 600GB). Revising these queries in more detail, we found that the query planner sometimes chose a join order that immediately led to massive intermediate results by choosing to first compute object–object joins on high-degree nodes, such as countries, languages, etc., when much better join orders were available.

C-Ring uses the least space among all the indexes and, like Ring, offers quite consistent times, being roughly a constant factor slower than Ring for each pattern. Compared with Qdag, the closest index in terms of space, C-Ring is faster in 8 patterns and slower in 9. As Table 2 shows, however, C-Ring is much more stable than Qdag in terms of query times across all patterns.

5.2.3 Performance with fixed predicates Since all queries in this benchmark have fixed predicates, we test the simplified ring of Section 4.4 on the same data and queries. This is intended to show how the Ring can be optimized for this case; other indexes are not included because we do not know how they can be similarly simplified. The exception is the Qdag; recall footnote 5.

Table 3 shows the space and average query time of these variants, for the compact indexes. The Ring for fixed predicates uses about half the space of the general Ring, but it is also about twice as slow. In fact, it uses about the same space as the C-Ring, though the latter is four times slower than the Ring. The C-Ring for fixed predicates is even smaller, reaching just 4.90 bytes per triple, which coincides with the space Qdags obtain when they are built as sets of two-dimensional grids (one per predicate) instead of a single three-dimensional grid. Qdags, however, are 50 times slower on average. Figure 15, in Appendix C, shows the detailed times.

5.3 Real-world benchmark at full scale

In order to test these systems for a realistic workload at scale, we perform experiments evaluating basic graph patterns taken from real-world queries over the full Wikidata graph of $n = 958{,}844{,}164$ triples, which occupies 10.7 GB in plain form and 7.9 GB in packed form. In search of challenging examples, we downloaded queries that gave timeouts from the Wikidata query logs [40], and selected queries with a single basic graph pattern, obtaining 1,300 unique queries. The minimum, mean and maximum number of triple patterns and variables per query were $(1, 2.4, 22)$ and $(1, 2.6, 16)$, respectively. Unlike the queries used previously, this set contains constant subjects and objects, variable predicates, etc. We provide more statistics in Appendix B.3.

[6]Just leaving lonely variables to the end, which helps LTJ, would probably harm Yanakakis' algorithm, as they would not be used to filter tuples in the parent GHD node.

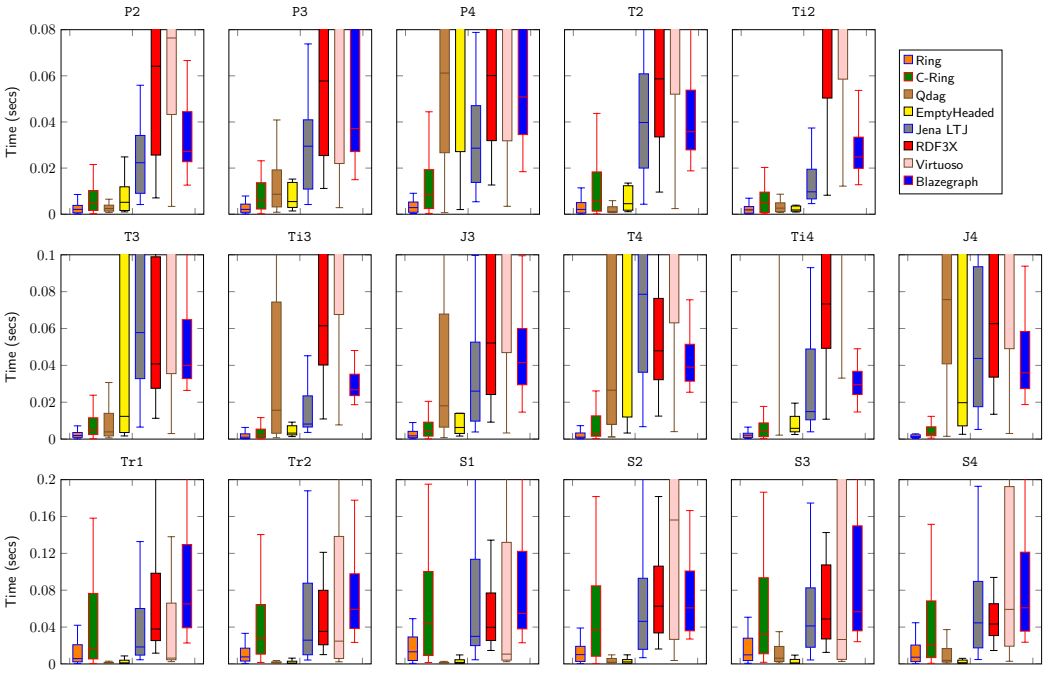
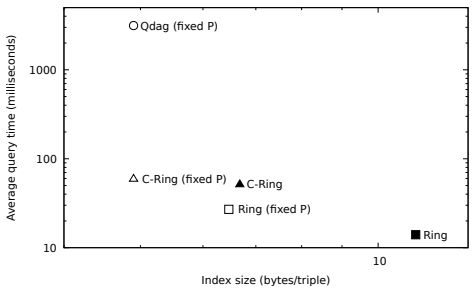


Fig. 10. Comparison of query times (in seconds). The boxes span from the 25% to the 75% percentile, with the median marked inside. The lines extend from minima to maxima, removing outliers

Table 3. Space and average time of the compact indexes, including variants that only support fixed predicates.

System	Space	Time
Ring	11.16	14
C-Ring	6.68	52
Ring (fixed P)	6.47	27
C-Ring (fixed P)	4.90	60
Qdag (fixed P)	4.90	3,141



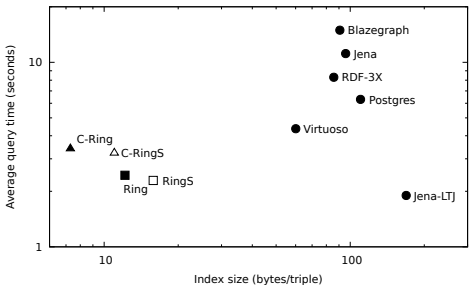
We exclude DuckDB because the larger graph does not fit in memory[7], EmptyHeaded because its index requires 900 GB and cannot be loaded into our main memory, and Graphflow as we estimate that it would require terabytes of main memory and does not support constants in arbitrary positions. We also exclude Qdag because the index does not handle triple patterns with constants in arbitrary positions as occur in this benchmark.

The graph database systems – Blazegraph, Jena, Jena LTJ, RDF-3X and Virtuoso – may use secondary storage and work with strings as constants in the queries, graphs, and results (though

[7]We also tried DuckDB on a machine with 350GB of RAM, where it succeeded in loading the larger graph, but exhibited highly variable and unstable behaviour, being killed by the Operating System on the tenth query of the benchmark. We believe this to be due to choosing join orders that generate massive intermediate results.

Table 4. Index space (in bytes per triple) and some statistics on the query times (average and median, in seconds) on the full Wikidata graph. The last column (TimeOuts) counts the queries taking over 10 minutes.

System	Space	Avg	Med	TO
Ring	12.15	2.44	0.011	4
RingS	15.83	2.29	0.018	4
C-Ring	7.29	3.43	0.038	5
C-RingS	11.00	3.25	0.052	5
Blazegraph	90.79	14.93	0.070	24
Jena	95.83	11.16	0.035	18
Jena LTJ	168.84	1.90	0.162	1
Postgres	110.12	6.30	0.139	4
RDF-3X	85.73	8.30	0.126	13
Virtuoso	60.07	4.37	0.050	7



internally they may use dictionary-encoded numeric constants). The other systems – Ring, C-Ring, DuckDB, EmptyHeaded, Graphflow, and Qdag – work in memory with dictionary-encoded numeric constants in the queries, graphs, and results. The Postgres system may use secondary storage but uses dictionary-encoded constants in the queries, data and results.

This difference can be unfair with the systems managing strings, as it may induce additional time and space overhead. To ensure a fair comparison in all cases, we introduce two new variants of our structure, **RingS** and **C-RingS**, which include a succinct dictionary [41] and thus receives strings in the queries and returns strings in the results. We use the variant `HTFC-rp` with sampling 64 [41], building separate dictionaries for SO and for P. Since this is a 32-bit structure, we cut the dictionary into lexicographic slices of a fixed number of strings and binary search the right dictionary to encode a query string into an integer; for decoding we find the dictionary directly with a division. Note that in these versions the mapping from strings to integers follows the lexicographic order.

The Ring index is built over this graph in 1.43 hours (11.2 million triples per minute) using 23.6 GB of RAM. It occupies 11.6 GB (12.15 bytes per triple) with plain bitvectors, just 8.4% more than the plain integer data. The dictionary of strings indexes nearly 296 million strings of total size 12.4 GB (more than the triples in plain integer form), in 13.4 minutes using 23.8 GB of working space. The string space is compressed to 27%, so as to use 3.68 additional bytes per triple, using 144 slices for SO. Considering the integer triples plus the strings, RingS compresses the raw data to 65%.

Table 4 shows statistics on the index space and query times obtained for our benchmark; using strings maintains essentially the same times on the Ring and C-Ring (with a somewhat lower average and larger median than the integer versions).

The systems supporting wco joins (Jena LTJ and the Ring variants) clearly outperform the others, showing that worst-case optimality makes a noticeable difference on real-life queries.

RingS is smaller than all classical indexes by a factor of 3.8–10.7, has the lowest median by a factor over 2, and outperforms the non-wco systems, on average, by a factor of 1.9–6.5. Only Jena LTJ, with the largest index, is 20% faster than RingS on average and has more stable times.

The comparison with Jena LTJ is interesting since it shares with RingS a similar algorithm (LTJ) and a similar heuristic to choose the variable ordering and treatment of lonely variables. The results show then that we can reduce the space usage of the structures that support LTJ by a factor over 10, at the price of a modest increase of 20% in the average query time. In turn, Jena LTJ outperforms Jena, its non-wco version that shares the same data representation, by an average factor of almost 6, at the price of increasing the space by a factor of 1.8 in order to store the six tries. RingS, then, is wco, using 6 times less space than Jena, and outperforms it by an average factor of almost 5.

C-RingS, while using 70% of the space of RingS (60% if we disregard the string dictionaries) and 5.5–15.5 times less space than the other indexes, is 40% slower than Ring and 70% slower than Jena LTJ (which is more than 15 times larger), but still faster than all the other non-wco systems.

6 Quads and higher dimensions

We now look at how our ring index could be extended in the future to handle relations of higher arity. We first focus on *quads*, or relations of dimension four, which are commonly used to store graph databases [18, 31, 32]. As it turns out, we can build rings for quads, but we pay a cost in terms of space, as now we need to work with two different rings for wco joins (classical schemes would need 4! = 24 orders). We then describe how our machinery can be extended to deal with arbitrary dimensions, at the cost of requiring more rings as the dimension grows. We show, however, that the space cost incurred by ring indexes is orders of magnitude lower than with classical indexes.

6.1 Indexing quads

Just as for triples, let us regard quads (s, p, o, g) as cyclic, and index them using four columns C_{S}, C_{P}, C_{O}, and C_{G}, each represented with wavelet trees. More precisely, we start with column C_{G} of the table SPOG, then column C_{O} of table GSPO, then column C_{P} of table OGSP, and finally column C_{S} of table POGS. As with triples, we can use functions F_i to track a tuple across all 4 tables, which allows us to retrieve the content of any tuple in the original graph.

A problem with this ring is that it is possible that a cyclic quad-pattern has, at a certain moment, two non-contiguous areas with constants. Precisely, this happens when (only) S and O, or P and G, are bound. This configuration defeats our strategy to implement the LTJ algorithm in wco time: we cannot apply Lemma 3.14 directly, which means that the leap operation takes more time because we need to iterate over elements that may not contribute to the overall answer.

This problem can be circumvented by adding a second ring (thus doubling the space) that indexes another order, like SOPG, where S and O, as well as P and G, are contiguous. This ring then starts with order SOPG and builds new columns C'_{G}, C'_{P}, C'_{O}, and C'_{S}. Given the order in which the variables will be bound, one can determine which of the two rings must be used for each quad-pattern so that it never has non-contiguous areas of constants. Note that our technique allows that different quad-patterns of the query use different rings.[8]

A second point is that, while our backward extension algorithm (Section 3.4.2) is general and can be used as-is on quads, our forward extension was described for the case where exactly one attribute is bound. Over quads, however, we may need to process forwards a quad-pattern where the bound part has two consecutive elements, for example when binding x in $(s, p, ?x, ?y)$. Consider that case; all the others are analogous because the quad is cyclic. Say that we want the smallest $c_x \geq c$ such that $(s, p, c_x, ?y)$ occurs in G. The solution must then be extended to performing two restrictions (by p and s) from the interval $C_{\text{O}}[c\,..\,U]$, and then returning to C_{O} using F_*^{-1} twice.

Concretely, we start from the interval $[s_o\,..] := [A_{\text{O}}[c] + 1\,..]$, so that $C_{\text{P}}[s_o\,..]$ represents the quads with an object in $[c\,..\,U]$. Since P precedes O in our ring with order SPOG, we restrict for p on $C_{\text{P}}[s_o\,..]$ using Lemma 3.9 to obtain $C_{\text{S}}[s_p\,..]$, which represents the quads with predicate p followed by an object in $[c\,..\,U]$. Since S precedes P, we now perform a second restriction, for s on $C_{\text{S}}[s_p\,..]$, to obtain $C_{\text{G}}[s_s\,..]$, which represents the quads with subject s and predicate p followed by an object in $[c\,..\,U]$. Since C_{G} is ordered by SPOG, the quad represented by $C_{\text{G}}[s_s]$ contains the smallest object in $[c\,..\,U]$ associated with subject s and predicate p. We find it with $q := F_{\text{P}}^{-1}(F_{\text{S}}^{-1}(s_s))$ (Eq. (3)), which leads us to the position of c_x in table OGSP, thereafter binary searching for c_x such that

[8] As seen later, we can even choose the variable binding order on the fly, and switch from one ring to the other if necessary.

$A_{\mathrm{O}}[c_x] < q \leq A_{\mathrm{O}}[c_x + 1]$.[9] If we want to obtain the range $C_{\mathrm{G}}[s'_i \,..\, e'_i]$ for spc_x, we must again apply two restrictions from the range $[A_{\mathrm{O}}[c_x] + 1 \,..\, A_{\mathrm{O}}[c_x + 1]]$, just as in Lemma 3.14.

6.2 Indexing higher dimensions

The discussion on quads can be generalized to relations with d attributes: provided we can always choose a ring where the bound variables are contiguous in its order, we can solve joins in wco time. We can also use any of the rings to extract a tuple in $O(d \log U)$ time exactly as in Lemma 3.11, now iterating over the d attributes using F_j. Since we can start at any position of the ring, and moreover we can choose any interpretation of $\mathcal{A}$ as $[1 \,..\, d]$ in Definition 3.1, our algorithm lists the elements of the ith tuple of a table sorted by any desired attribute order.

To answer basic graph patterns using Lemma 3.14, at each stage of the computation we maintain the range of the (cyclically) leftmost bound attribute. We can extend the range backwards to include the preceding column in $O(\log U)$ time per intersection step, but extending the range forwards takes us $O(d \log U)$ time, because we have to navigate from the rightmost to the leftmost bound attribute for each intersection step, as described for the quads. As there can be dm variable positions to instantiate in the tuple patterns, the time that multiplies Q^* (the AGM bound) is now $O(d^2 m \log U)$.

This works for queries that do not repeat variables in the same tuples (i.e., have no equality selections on attributes of the same relation). If we extend the idea of Section 3.5 to handle variables appearing multiple times in a tuple, we incur a super-exponential penalty factor on d in the query time (more precisely, a $\Theta(B_d)$ penalty factor, where B_d is the dth Bell number). Such queries are not typically considered by wco join algorithms in a relational setting. This super-exponential penalty may be mitigated in practice by covering only those equalities that occur in some data tuple with repeated elements, and/or more generally accepting non-wco joins in such cases.

We then obtain the following result; the number of rings needed is bounded next.

Theorem 6.1. *On a set of n tuples in $[1 \,..\, U]^d$, the ring index solves a basic graph pattern query of m tuple patterns with no variables repeated in the same tuple in time $O(Q^* \cdot d^2 m \log U)$, where Q^* is the maximum possible output of such a query on some set of n tuples in $[1 \,..\, U]^d$ (the AGM bound). The working space of the query algorithm is $O(v+1)$ words, where v is the number of distinct variables in the query. The size of the ring index is $dn \log_2 U + o(dn \log U)$ bits times the number of orders it has to index, which is $O(2^d)$, and it can retrieve the ith tuple under any desired order in time $O(d \log U)$.*

For simplicity, we are focusing on indexing a single relation of dimension d. A database contains in general various relations with different numbers d of attributes. Our scheme extends naturally to indexing each relation separately, with the number of rings needed for each, and to combine them freely in join queries. At query time, instead of maintaining a column range per triple in the basic graph pattern as in Section 3.3, we maintain a column range per relation involved in the join (one per mention if the relation is mentioned several times).

6.2.1 On the number of rings needed. The remaining question is how many orders, or rings, must we build so that, independently of the variable elimination order, all attributes that have already been bound in the tuple patterns are contiguous in some order. We have shown that one order suffices for $d = 3$, but we need two for $d = 4$. How many indexes are needed to support LTJ in general dimension d? We answer this question in three parts. First, we discuss the number needed when using traditional indexes supporting prefix-lookups, which we denote as *flat* indexes. We define the important concept of *trie switching*, which reduces the number of traditional indexes from $d!$ to $O(2^d d^{1/2})$ (and to $O(2^d)$ in Section 7.2.1), and is thus of independent interest. We then

[9] As discussed in Section 4.1, we can also find q in backward direction using Eq. (2), $q := F_{\mathrm{O}}(F_{\mathrm{G}}(s_s))$. This can be faster because F_*^{-1} uses operation `select`, which is slower than `rank` in practice. In general, one can choose the cheapest path.

turn to cyclic indexes, starting with the unidirectional version and then continuing with our ring index. We show that the number of cyclic indexes needed to support LTJ are between $\Omega(2^d d^{-1/2})$ and $O(2^d)$, and that the actual numbers are much lower than the number of flat indexes needed.

Flat indexes and trie switching, variants W and TW. A classical index (called flat in Figure 2) on d columns needs to store, in principle, all the $w(d) := d!$ possible orders to support LTJ-like wco algorithms. In our discussion, we refer to these indexes as class W (for Worst-case-optimal).

However, for $d \geq 4$, we can reduce the number of classic indexes by reordering the variables we have already bound. For example, when indexing quads (s, p, o, g), this avoids storing a trie[10] with the order GSPO if we have tries for GSOP and SGPO: to process a tuple in the order GSPO, we descend by some g in the trie of GSOP to retrieve s, and then by permuting gs to sg we can now switch to the trie of SGPO in order to retrieve p, and then continue onto o, thus emulating GSPO order.

We call TW the class of indexes using trie switching. TW indexes need to store at least $(d-l)\binom{d}{l}$ orders for any $0 \leq l < d$, because we may have any subset of l variables already bound to constants and need to intersect using any of the $(d-l)$ remaining variables. Each such arrangement requires storing a different order. This formula is maximized for $l = \lfloor d/2 \rfloor$, yielding a lower bound of

$$tw(d) := \left\lceil \frac{d}{2} \right\rceil \binom{d}{\lfloor d/2 \rfloor}$$

orders. We now prove that $tw(d)$ is also an upper bound by building a sufficient set of $tw(d)$ orders. The key idea is that the strings that suffice to handle all the cases in level l can be built in a such a way that their prefixes also handle the lower levels. This motivates the next definition and lemma.

Definition 6.2. A set of strings $S[1\,..\,l+1]$ is (l, d)-complete if, for every $0 \leq m \leq l$, every possible subset of m values in $[1\,..\,d]$, in some order, followed by any other final number in $[1\,..\,d]$, is a prefix $S[1\,..\,m+1]$ of some string S in the set. A $(d-1, d)$-complete set is called simply d-complete.

Lemma 6.3. *There exists a d-complete set of $tw(d)$ strings.*

Proof. We proceed by induction on l, building for every $0 \leq l < d$ an (l, d)-complete set. For $l \leq d/2$, this set will have $(d-l)\binom{d}{l} \leq tw(d)$ strings, and $tw(d)$ strings will suffice for larger l.

For $l = 0$, we have the d distinct strings of length 1, one per final number. Now, assume we have an (l, d)-complete set of size $(d-l)\binom{d}{l}$. Those strings list all the possible $\binom{d}{l+1}$ subsets of $l+1$ values, each appearing by symmetry $(d-l)\binom{d}{l}/\binom{d}{l+1} = l+1$ times. To produce an $(l+1, d)$-complete set, we extend each of the $\binom{d}{l+1}$ different subsets by each of the $d-l-1$ possible final numbers. Since we already have $l+1$ strings for each such subset, we can extend those with $l+1$ of the $d-l-1$ possible final numbers. If $d-l-1 > l+1$, however, we will have to create new copies of some of those $l+1$ strings to extend them with the remaining possible final numbers. At the end, we have a set of $(d-l-1)\binom{d}{l+1}$ strings $S[1\,..\,l+2]$. This set is $(l+1, d)$-complete because it satisfies the definition for $m = l-1$ and was built by extending a set of prefixes that was already (l, d)-complete.

Once $d-l-1 \leq l+1$, that is, $l+1 \geq d/2$, we create no new strings since there will be enough of length $l+1$ to extend them to length $l+2$ in all the possible ways. The maximum size then occurs when $l = \lceil d/2 \rceil - 1$ and our set has $(d - \lceil d/2 \rceil + 1)\binom{d}{\lceil d/2 \rceil - 1} = tw(d)$ strings. □

Trie switching can also be used with the ring without needing extra space: if we find a range for some contiguously bound variables in one ring, we search another ring with the same variable values contiguously bound in another desired order using Lemma 3.14. Since we have to bind the (up to d) variables again in the new index, each such change of index costs $O(d \log U)$ time, which

[10] We speak of concrete tries for simplicity, though the actual (classic) implementation may use other structures, like B-trees.

is within the time complexity given in Theorem 6.1. (In fact, our description in Section 3.4 assumes that we search again for the column range of the bound part of the triple patterns upon each new binding; only in the engineering done in Section 4.2 we remember the column range resulting from the current bindings in order to reduce the time in practice.) In the following discussion we analyze both versions of each new index scheme: with and without trie switching.

Cyclic indexes, variants CW and CTW. On indexes supporting cyclic tuples but not trie switching (which we call CW indexes), exactly $cw(d) := (d-1)!$ orders are needed. This is because the $d!$ permutations can be divided into $(d-1)!$ equivalence classes of size d, where two permutations Π and Π' are equivalent if one is a cyclic rotation of the other, that is, $\Pi[1\,..\,d] = \Pi'[i\,..\,d]\cdot\Pi'[1\,..\,i-1]$ for some i. Exactly one index per equivalence class is then needed.

As explained, we can enable trie switching on cyclic indexes, leading to what we call CTW indexes. We call $ctw(d)$ the number of CTW indexes needed to implement LTJ. Seen as a lower bound for trie switching, $tw(d)$ is the number of prefixes of length $\lceil d/2\rceil$ that cover every possible subset of $\lceil d/2\rceil - 1$ positions followed by any other position. Since each position in the cycle is the starting point of a sequence of $\lceil d/2\rceil$ elements, we need at least $\lceil tw(d)/d\rceil \leq ctw(d)$ cycles to cover all the needed prefixes. For example, with SPOG we obtain $\{s,p\}$ with variable o, $\{p,o\}$ with variable g, $\{o,g\}$ with variable s, and $\{g,s\}$ with variable p.

We now prove two upper bounds for $ctw(d)$. The first one, useful for small d values, shows that CTW cuts the number of orders required for TW at least by half, because $tw(d-1) \leq tw(d)/2$.

LEMMA 6.4. *It holds that* $ctw(d) \leq tw(d-1) = \lfloor \frac{d}{2} \rfloor \binom{d-1}{\lfloor d/2 \rfloor}$.

PROOF. Consider a $(d-1)$-complete set of strings TW (Lemma 6.3) forming a TW index for the symbols $[1\,..\,d-1]$. Let $\mathrm{dom}(S)$ be the set of symbols in string S, and let $TW_S \subseteq TW$ be the set of strings in TW that are prefixed with any S' such that $\mathrm{dom}(S') = \mathrm{dom}(S)$.

We build a (possibly suboptimal) CTW index for the values $[1\,..\,d]$ by simply appending d to each of the strings in TW; this immediately implies the lemma. To see that this index is valid, consider a variable instantiation order $X \cdot d \cdot Y$ (i.e., a permutation of the dimensions), with $x = |X|$ and $y = |Y|$. Processing the partial queries X (resp., Y) on the set TW yields some string $S_X \in TW_X$ (resp., $S_Y \in TW_Y$) such that $\mathrm{dom}(S_X[1\,..\,x]) = \mathrm{dom}(X)$ (resp., $\mathrm{dom}(S_Y[1\,..\,y]) = \mathrm{dom}(Y)$). Thus, we can process X in the CWT index in the same way, ending on $(S_X \cdot d)[1\,..\,x]$. Since $\mathrm{dom}(S_Y[1\,..\,y]) = \mathrm{dom}(Y)$, it follows that $\mathrm{dom}(S_Y[y+1, d-1]) = \mathrm{dom}(X)$. In the CWT index, we can then, after processing X, switch from $(S_X \cdot d)[1\,..\,x]$ to $(S_Y \cdot d)[y+1\,..\,d-1]$. We can now extend that match with variable $d = (S_Y \cdot d)[d]$ and finally, by circularity, process Y at $(S_Y \cdot d)[1\,..\,y]$. □

We now derive an upper bound for $ctw(d)$ that, though weaker than the preceding one for $d \leq 13$, is asymptotically stronger.

Definition 6.5. Let a set of strings $S[1\,..\,l]$ be (l,d)-sufficient if, for every $0 \leq m \leq l$, every possible subset of m values in $[1\,..\,d]$, in some order, is a suffix $S[l-m+1\,..\,l]$ of some string S in the set. A (d,d)-sufficient set is simply called d-sufficient.

LEMMA 6.6. *There exists a d-sufficient set of $sw(d)$ strings, where*

$$sw(d) := \binom{d}{\lfloor d/2 \rfloor}.$$

PROOF. We proceed by induction on l, building for every $0 \leq l \leq d$ an (l,d)-sufficient set. For $l \leq d/2$, this set will have $\binom{d}{l} \leq sw(d)$ strings, and $sw(d)$ strings will suffice for larger l.

For $l = 0$, we just have the empty string. Now, assume we have an (l,d)-sufficient set of size $\binom{d}{l}$. To produce an $(l+1,d)$-sufficient set, we need to create $\binom{d}{l+1}$ strings, which is more than those

we have as long as $d - l > l + 1$. Each string of our (l, d)-sufficient set is extended by prepending some element it does not contain, and the remaining $\binom{d}{l+1} - \binom{d}{l}$ subsets are obtained by duplicating some string of the (l, d)-sufficient set and prepending a new element to it. When $l = \lfloor d/2 \rfloor$, our set stops growing because we always have enough strings in our (l, d)-sufficient set to prepend all the distinct elements needed to form an $(l + 1, d)$-sufficient set. □

Lemma 6.7. *It holds that* $ctw(d) \leq sw(\lfloor d/2 \rfloor) \cdot tw(\lceil d/2 \rceil) + sw(\lceil d/2 \rceil) \cdot tw(\lfloor d/2 \rfloor)$.

Proof. Let $c := \lfloor d/2 \rfloor$, SW_s (resp., TW_s) be a c-sufficient (resp., c-complete) set of strings, and SW_l (resp., TW_l) be a $(d - c)$-sufficient (resp., $(d - c)$-complete) set of strings where we have summed c to every symbol (the subscripts s and l stand for small and large symbols). We then build a CTW index by concatenating every string in SW_l with every string in TW_s, and every string in SW_s with every string in TW_l. The size of the CTW index is then as stated.

To see this is a valid index, consider any instantiation order $X[1\,..\,d]$ (i.e., a permutation in $[1\,..\,d]$) we process left to right. Let $X_s(p)$ (resp., $X_l(p)$) be the subsequence of $X[1\,..\,p]$ formed by symbols in $[1\,..\,c]$ (resp., $[c + 1\,..\,d]$). Let $\mathrm{dom}(S)$ be the set of symbols in S. A consequence of SW_s being c-sufficient and SW_l being analogously obtained from a $(d-c)$-sufficient set is that, if we have processed $X[1\,..\,p]$, we always have some string $S_s \in SW_s$ such that $\mathrm{dom}(S_s[c-|X_s(p)|+1\,..\,c]) = \mathrm{dom}(X_s(p))$, and some string $S_l \in SW_l$ such that $\mathrm{dom}(S_l[d - c - |X_l(p)| + 1\,..\,d - c]) = \mathrm{dom}(X_l(p))$.

We start, for $p = 0$, with any $S_s \in SW_s$ and $S_l \in SW_l$. After processing $X[1\,..\,p]$, we consider $X[p + 1]$. If $X[p + 1] \in [1\,..\,c]$, then, because TW_s is c-complete, there is a string $S \in TW_s$ such that $\mathrm{dom}(S[1\,..\,|X_s(p)|]) = \mathrm{dom}(X_s(p))$ and $S[p + 1] = X[p + 1]$. Also, by construction, $S_l \cdot S$ is in our CWT index. If, instead, $X[p + 1] \in [c + 1\,..\,d]$, then, because TW_l is built from a $(d - c)$-complete set, there is a string $S \in TW_l$ such that $\mathrm{dom}(S[1\,..\,|X_l(p)|]) = \mathrm{dom}(X_l(p))$ and $S[p + 1] = X[p + 1]$, and by construction $S_s \cdot S$ is in the CWT index. Thus, we can always maintain a contiguous range for $X[1\,..\,p + 1]$ in some of the CWT strings by using trie switching and circularity. □

By Stirling's approximation, $sw(d) = 2^{d+1/2}/\sqrt{\pi d}\,(1 + O(d^{-1/2}))$ and $tw(d) = 2^{d-1/2}\sqrt{d/\pi}\,(1 + O(d^{-1/2}))$; thus $tw(d) \in \Theta(2^d d^{1/2})$ and $cwt(d) \leq 2^d/\pi\,(1 + O(d^{-1/2})) \in O(2^d)$. This shows that circular indexes are asymptotically smaller than flat indexes, even when trie switching is used. On the other hand, the lower bound $ctw(d) \geq \lceil tw(d)/d \rceil \in \Omega(2^d d^{-1/2})$, yields a $\Theta(d^{1/2})$ factor of uncertainty about the size of circular indexes with trie switching.

Bidirectional indexes, variants CBW and CBTW. Consider a cyclic index with no trie switching, which we call CBW; let $cbw(d)$ be the number of cycles of this kind we need. In the unidirectional case (CW), recall that $cw(d) = (d-1)!$. In the bidirectional case, we can remove CW indexes that are the reverse of others (e.g., SPOG and GOPS). If $d > 2$, this yields the upper bound $cbw(d) \leq cw(d)/2$, because every cycle in the CW index appears in reverse order as well, and that is different from the original order and from any other cycle. Thus, bidirectionality cuts the number of indexes by at least half (for $d > 2$) if trie switching is not enabled. In fact, we may be able to cut the number of indexes further. From the starting point in the cycle, under CBW we can extend the range of bound values left or right, in any of the 2^{d-2} sequences of choices, until a single position is left. The sequence of values included in the range, for each combination, covers a new permutation. For example, if from SPOG we start at position 1, we obtain SGOP with left-left, SGPO with left-right, SPGO with right-left, and SPOG with right-right. Thus, each index can, at best, cover $d2^{d-2}$ different permutations, and thus a lower bound is $cbw(d) \geq \lceil (d-1)!/2^{d-2} \rceil$ orders.

Adding bidirectionality to a cyclic index with trie switching (which we call CBTW), an immediate lower bound is $cbtw(d) \geq \lceil ctw(d)/2 \rceil$, because any CBTW index storing r orders can be converted into a CTW index storing $2r$ orders, by storing each order and its reverse. We can always switch to

the reverse order whenever the CBTW index extends the range backwards, so that the CTW index always extends it forwards. We can, however, derive a slightly better lower bound due to rounding.

Lemma 6.8. *A CBTW index must index at least the following number of orders:*

$$cbtw(d) \;\geq\; \max\left\{\left\lceil \binom{d}{l} \frac{\lceil (d-l)/2 \rceil}{d} \right\rceil,\; 1 \leq l < d \right\}.$$

Proof. Every position of every cycle is the starting point of a segment of length l. The segment can be extended in both directions by a new variable, thus each subset of l variables needs to appear $\lceil (d-l)/2 \rceil$ times. Since there are $\binom{d}{l}$ distinct subsets, the index needs to cover $\binom{d}{l}\lceil (d-l)/2 \rceil$ starting points. Since each cycle provides d starting points, the lower bound for each value of l follows. For example, with spog we cover $\{s, p\}$ with variables o or g, $\{p, o\}$ with variables g or s, $\{o, g\}$ with variables s or p, and $\{g, s\}$ with variables o or p; so we cover 8 of the $\binom{4}{2} \cdot 2 = 12$ needed combinations and thus we need at least two cycles. (Due to rounding effects, the formula is not always maximized when $l = \lfloor d/2 \rfloor$.) □

Since $\lceil ctw(d)/2 \rceil \leq cbtw(d) \leq ctw(d)$, $cbtw(d)$ is asymptotically equal to $ctw(d)$. This concludes the proof of the following theorem.

Theorem 6.9. *The following bounds hold on the number of orders that must be indexed by various classes of indexes.*

- $w(d) = d!$ *and* $cw(d) = (d-1)!$.
- $\lceil cw(d)/2^{d-2} \rceil \leq cbw(d) \leq cw(d)/2$ *for* $d > 2$.
- $tw(d) = \lceil \frac{d}{2} \rceil \cdot sw(d) \in \Theta(2^d d^{1/2})$.
- $\lceil tw(d)/d \rceil \leq ctw(d) \leq tw(d-1)$, *so* $ctw(d) \in \Omega(2^d d^{-1/2})$.
- $ctw(d) \leq sw(\lfloor d/2 \rfloor) \cdot tw(\lceil d/2 \rceil) + sw(\lceil d/2 \rceil) \cdot tw(\lfloor d/2 \rfloor) \in O(2^d)$.
- $cbtw(d) \geq \max\{\lceil \binom{d}{l} \frac{\lceil (d-l)/2 \rceil}{d},\; 1 \leq l < d \rceil\}$.
- $\lceil ctw(d)/2 \rceil \leq cbtw(d) \leq ctw(d)$, *so* $cbtw(d) \in \Theta(ctw(d))$.

Thus, the ring index with trie switching must store between $\Omega(2^d d^{-1/2})$ *and* $O(2^d)$ *orders. A traditional (i.e., non-cyclic) index, even with trie switching, must store* $\Theta(2^d d^{1/2})$ *orders.*

The only structure supporting wco joins while using fewer orders are Qdags [49], which need to index only one order. In exchange, their query time is $O(Q^* \cdot 2^d m \log U)$ instead of the $O(Q^* \cdot d^2 m \log U)$ time we obtain with rings.

The actual numbers. Although we have constructive proofs to build indexes of every kind within guaranteed upper bounds, we ran exhaustive searches to find the exact number of orders that suffice for running wco algorithms in each case, for $d \leq 8$. When the search space was too large, we resorted to approximation algorithms for set cover and gradient descent. Table 5 shows the number of orders that must be stored to implement wco algorithms (W), if we support cyclic tuples (C), bidirectionality (B), and trie switching (T). Our ring index then corresponds to CBW, and to CBTW if we use trie switching. A cyclic unidirectional index corresponds to CW and CTW. A classical flat index corresponds to W, or to TW with trie switching. The numbers verified our exact formulas for W, TW, and CW. Instead, there is a gap between our lower bounds for CTW, CBW, and CBTW, and the approximations we obtained for the larger values of d. The approximations are always much smaller than our upper bounds, and closer to the lower bounds.

Per Table 5, trie switching slashes the number of required orders in flat indexes by orders of magnitude, but the number of indexes is still unfeasible for, say, $d > 4$. Adding circularity and bidirectionality further reduces the number of trie-switching indexes by another order of magnitude.

Table 5. Number of orders that must be indexed to support wco algorithms depending on the index capabilities

d	Flat		Cyclic		Ring	
	W	TW	CW	CTW	CBW	CBTW
2	2	2	1	1	1	1
3	6	6	2	2	1	1
4	24	12	6	4	2	2
5	120	30	24	8	5	5
6	720	60	120	[10,12]	10	7
7	5040	140	720	[20,24]	[23,37]	[10,12]
8	40320	280	5040	[35,50]	[79,168]	[21,25]

This enables the use of wco algorithms on dimensions that would be unfeasible with classical approaches, even with trie switching. We provide the rings for these cases in Appendix D.

6.2.2 Finding the right index. With an exponential number of orders indexed, how to find the proper ring to instantiate the next variable is an issue. We can do this in constant time by building a table of $2^d \cdot d$ cells which, given a subset of tuple positions already instantiated and a new position to instantiate, gives the indexed order that must be used next. To build this table for CBTW, for example, we take every indexed order S and, for every $S[i\,..\,j]$ with $i \neq j$ (regarded as cyclic, i.e., i can be larger than j), makes the table point to $S[i\,..\,j]$ at the cells with subset $\mathrm{dom}(S[i\,..\,j])$ and next positions to instantiate $S[i-1]$ or $S[j+1]$. The space of this structure is at most $O(d^{3/2})$ per order indexed (because $cbtw(d) \in \Omega(2^d d^{-1/2})$) and is built in time $O(d^2)$ per order indexed, that is, at most $O(2^d d^2)$. This space and construction time are negligible compared to the rings themselves.

7 From rings to order graphs

Our rings for d dimensions are obtained in Section 3 by choosing an attribute order Π and re-sorting it repeatedly by the dth attribute until returning to Π. The set of last columns of all the re-sorted tables forms the ring index. In Section 6, when one ring was not sufficient (i.e., for $d > 3$), we used various orders, creating one ring per order. The re-sorting concept, however, is not limited to choosing the last attribute. In the general case, we can define the concept of an *order graph*.

Definition 7.1. An *order graph* of dimension d is a labeled directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a subset of all the $d!$ possible orders, and an edge from node Π to node Π', labeled $j \in [1\,..\,d]$, can exist only if Π' is obtained from Π by moving value j to the front.

As an example for $d = 4$, if $\Pi = 2143$, $\Pi' = 1243$, then an order graph may have a labeled edge $2143 \xrightarrow{1} 1243$. In this view, a ring corresponds to an order graph with d edges forming a directed cycle, where the label of the edge leaving from each node Π is labeled $\Pi(d)$. An index with several rings corresponds to several disjoint cycles of length d. We are interested in *complete* order graphs, which are those that suffice to run LTJ in wco time.

Definition 7.2. An order graph $\mathcal{G}$ of dimension d is *complete* if, for any subset $S \subsetneq \{1, \ldots, d\}$ and any $x \in [1\,..\,d] \setminus S$, there exists a directed path of length $|S| + 1$ in $\mathcal{G}$ whose labels contain exactly the elements of $S \cup \{x\}$ (in some order), with the one labeled x starting or ending the path.

From the discussion of the previous section, it follows that trie switching makes it possible to run LTJ in wco time on the columns corresponding to complete order graphs.

Definition 7.3. Consider a complete order graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of dimension d. Then, the *order index* of $\mathcal{G}$ is formed by $|\mathcal{E}|$ columns, with a column C_j^{Π} for each edge $\Pi \xrightarrow{j} \Pi'$ in $\mathcal{G}$. This corresponds to

the column of the attribute j in Π, that is, the $\Pi^{-1}(j)$th column of the table with order Π (where Π^{-1} is the inverse of the permutation Π). The order index also includes the global cumulative frequency arrays A_j for each attribute $j \in [1\,..\,d]$ (the arrays A_j depend only on j and not on Π). We remark that, given Π, j determines Π' and vice versa.

We can extract tuples from order indexes in time $O(d \log U)$, like with the ring. By definition, any complete order graph must have a path of d edges labeled with a permutation of $[1\,..\,d]$ since any subset $S \subsetneq \{1, \ldots, d\}$ of size $d-1$ must extend to the remaining variable, forwards or backwards. Let this path go from nodes Π to Π' in $\mathcal{G}$. We can then retrieve the values of the ith tuple in the order of Π as follows. We start from node Π, and use $F_j(\cdot)$ to track the position i across the successive columns towards Π', while retrieving the values from the corresponding column positions.

We can also use order indexes to answer pattern queries, thereby generalizing Theorem 6.1.

Theorem 7.4. *An order index built on the order graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of a table $\mathcal{R}$ with n tuples and d attributes in $[1\,..\,U]$ can use $|\mathcal{E}|n \log_2 U + o(|\mathcal{E}|n \log U)$ bits of space and solve queries Q of m tuple patterns in time $O(Q^* \cdot d^2 m \log U)$, where Q^* is the AGM bound of query Q on $\mathcal{R}$.*

Proof. We will mimic Algorithm 1. Each time we have a partially bound tuple pattern t_i and want to further bind another variable x to the smallest value $c_x \geq c$, we will take a path from Π to Π' in the order graph $\mathcal{G}$ of our index, whose labels $\ell_1, \ldots, \ell_k$ correspond to the k bound attributes in t_i in some order, and that is preceded or followed by x. Let $C_1, \ldots, C_k$ be the index columns corresponding to the path from Π to Π'. We use the natural generalization of the backward and forward extensions given in Sections 3.4.2 and 6. If the edge labeled x follows the path, we start from the full range $C_1[1\,..\,n]$, perform k restrictions on columns $C_1, \ldots, C_k$, and end with a *range-next-value* with value c on the resulting range of column $C_x^{\Pi'}$, as in the backward extension. We then obtain the smallest value $c_x \geq c$ with which t_i can bind x. Instead, if the edge labeled x precedes the path, we use the forward extension. We start from $C_1[A_x[c]+1\,..]$ (note the order of Π starts with x), perform the restrictions on $C_2, \ldots, C_k$, choose the first value in the resulting range of C_k, and trace it back to $C_{k-1}, \ldots, C_1$ using F_*^{-1} (Eq. (3)), to finally obtain the desired value $c_x \geq c$ by computing the range $A_x[c_x+1\,..\,c_x+1]$ that contains the retrieved value in C_1. This takes $O(d \log U)$ time per occurrence of each bound variable in the query, which adds up to $O(d^2 m \log U)$ over all the triple patterns, and to $O(Q^* \cdot d^2 m \log U)$ overall, counting all the possible instantiations. □

We have assumed we know the path from Π to Π' corresponding to each subset S and new variable x. Analogously to Section 6.2.2, this can be precomputed in a table of size $O(2^d d)$ that, for each S and x, stores one suitable path, for a total space of $O(2^d d^2)$. This table is precomputed by traversing all the paths of $\mathcal{G}$ of length up to d and filling the corresponding cells.

We are also assuming, for simplicity, that every attribute of the relation is *queryable*, that is, it can be bound to constants in queries. In practice, there may only be $d' < d$ queryable attributes. In this case we build the order index for just those attributes, which has exponential size on d' only. In order to recover the complete tuples, we choose a path $v_0, \ldots, v_{d'}$ of the order graph labeled by the d' queryable attributes, and create an additional path from $v_{d'}$ with the remaining $d - d'$ attributes, so that their values can be retrieved from the columns associated with the $d - d'$ path edges.

7.1 Types and sizes of order indexes

An order index from graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ requires $|\mathcal{E}|n \log_2 U + o(|\mathcal{E}|n \log U)$ bits, that is, proportional to the number of edges in the graph because it stores one column per edge[11]. Such an order index is

[11]We must add $O(dU \log n)$ for the arrays A_* or $O(d(U+n))$ for the bitvectors D_*, of which we store one per attribute. Since $U \leq dn$, however, this is $O(d^2 n)$, and since we will soon see that $|\mathcal{E}| \in \Omega(2^d d^{1/2})$, this term is in $o(|\mathcal{E}|n \log U)$.

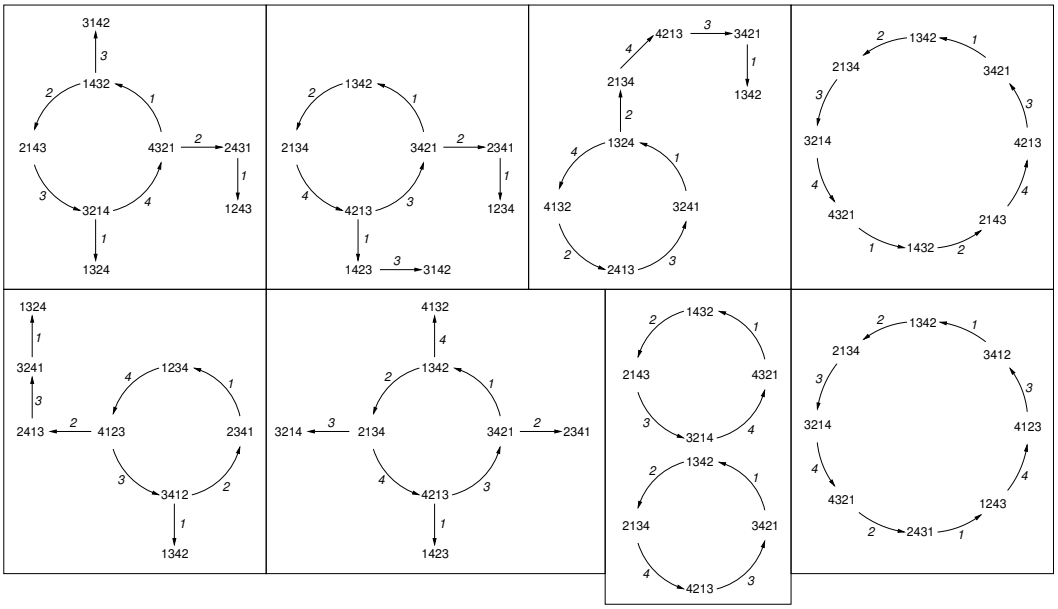


Fig. 11. The eight non-isomorphic complete order graphs of minimum size 8 for quads (i.e., $d = 4$)

essentially $|\mathcal{E}|/d$ times larger than the size of the raw data (and can recover any tuple). But what is the number of edges needed in complete order graphs? To analyze this question, let us simplify and say that an order index for $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is of size $|\mathcal{E}|$. Our ring indexes of Section 6.2 requiring k rings are then of size kd. A natural question is whether the smallest possible order index for dimension d consists of k rings, that is, k disjoint cycles, or if there are smaller order indexes of other shapes.

Indeed, for $d = 2$ and $d = 3$, the smallest order indexes, of size 2 and 3 respectively, are simple cycles when seen as order graphs. For $d = 4$, the use of two rings yields an order index of optimal size 8, yet there are other 7 non-isomorphic order indexes of the same size, as shown in Figure 11 (our notion of isomorphism on order graphs permits consistent renaming of the edge labels). Most other solutions consist of a "hairy" ring, that is, a single ring of length 4 with paths sprouting from some nodes. Two solutions, instead, are formed by a unique larger cycle, of length 8.

We can use an analogue to the idea of order graphs for unidirectional indexes, where we only extend backwards. More precisely, the edge labeled by the new variable x must always follow the path of S in Definition 7.2. In this case, already for $d = 3$ there are various optimal order indexes apart from two cyclic triangles; we show them in Appendix D.

The solutions using a single cycle are interesting because of their simplicity. Note that they depart from the concept that a ring must be of size d and represent a reordering of the attributes of a relation. When using a single cycle that contains all the needed paths, the result is a kind of ring containing several columns associated with the same attributes (though sorted in different ways), and where we do not always choose the last column of an order to obtain the next one.

We do not know if it is always optimal to use a single cycle, though we have found no counterexamples where an optimally-sized single cycle can be replaced by a (strictly) smaller order index. Instead, we next show that minimum-size order indexes must consist of a set of (possibly hairy) cycles, ruling out what we define next as confluential graphs.

Definition 7.5. An order index is *confluential* if its graph has nodes with in-degree larger than 1.

When proving next that confluential order indexes are non-optimal, we will focus on all the distinct label sequences of paths of length up to d. If we reduce the size of the graph without modifying this set of label sequences, then the smaller graph is still complete, both in the bidirectional and the unidirectional case. This assumes, as we have done up to now, that we are only interested in paths of $\mathcal{G}$ with no repeating labels. Paths with such repetitions can still be used: if we find a repeated label along a path we can find again the range of the already bound variable in the new column. This brings obvious inefficiencies in time, though it could lead to smaller order indexes in principle. We have not found, in our exhaustive searches, any solution of this kind that is not matched by another solution without repetitions, however. Our next lemma is general enough to consider paths with repeated labels as well.

Lemma 7.6. *No confluential order index can be of minimum size.*

Proof. Consider the graph of a confluential index of dimension d, and a node w with two edges $u \to w \leftarrow v$ leading to it. Our aim is to show that any path with $l \leq d$ distinct labels that uses one of those edges can be replaced by a similar path that uses the other.

Let us first assume that we are only interested in paths with no repeated labels. We then fix some $1 \leq k \leq l$ and consider two paths, $u_{-k} \to \cdots \to u_{-1} := u \to w := w_0 \to w_1 \to \cdots \to w_{l-k}$ and $v_{-k} \to \cdots \to v_{-1} := v \to w \to w_1 \to \cdots \to w_{l-k}$.

Note that the label of both edges $u \to w$ and $v \to w$ must be $\Pi_w(1)$, Π_w being the order of node w. (Indeed, the orders Π_u and Π_v must be identical if we remove $\Pi_w(1)$ from them, because they both become Π_w when we add $\Pi_w(1)$ at their beginning.) Because of our assumption, $\Pi_w(1)$ is different from both $\Pi_u(1)$ and $\Pi_v(1)$, since otherwise the labels of $u_{-2} \to u_{-1}$ or $v_{-2} \to v_{-1}$ would be $\Pi_w(1)$ and the subpaths we are considering would repeat the label $\Pi_w(1)$. Therefore, it must be that $\Pi_u(1) = \Pi_v(1) = \Pi_w(2)$, and thus $\Pi_w(2)$ is the label of both $u_{-2} \to u_{-1}$ and $v_{-2} \to v_{-1}$. The reasoning can be repeated to show that, in general, the label of both edges $u_{-t} \to u_{-t+1}$ and $v_{-t} \to v_{-t+1}$ must be $\Pi_w(t)$, for all $1 \leq t \leq k$. Therefore, the sequences of labels of both paths $u_{-k} \to \cdots w_{l-k}$ and $v_{-k} \to \cdots w_{l-k}$ are identical.

As this holds for any path of any length $1 \leq l \leq d$ starting at any distance $1 \leq k \leq l$ from w, we can safely remove either $u \to w$ or $v \to w$, and hence obtain a smaller graph, without altering the set of paths of lengths in $[1\,..\,d]$ existing in the graph. Indeed, all the paths of length up to d ending at w must be labeled with a reversed prefix of Π_w. If there are several edges leading to w, we can safely leave only the one ending a longest path to w; all the others are redundant.

Now assume we accept paths containing repeated labels. In this case, the paths are not necessarily of length l, but as long as necessary to contain l distinct labels. In this case, we reason identically while ignoring the repeated labels that appear in the paths. It can be equally shown that both paths leading to w end with $\Pi_w(1)$, that their preceding label different from $\Pi_w(1)$ must be $\Pi_w(2)$, that the preceding label not in $\{\Pi_w(1), \Pi_w(2)\}$ must be $\Pi_w(3)$, and so on. The result is still that both paths arriving at w by u or v are similar and thus we can safely remove $u \to w$ or $v \to w$. □

Lemma 7.7. *For every d, there is an optimal order index where all the nodes have indegree exactly* 1.

Proof. By Lemma 7.6, optimal order graphs are either trees or (possibly hairy) cycles. Yet, every tree in an order graph can be converted into a hairy cycle of the same size by identifying any node u of outdegree zero with the only node of indegree zero (i.e., make one arbitrary leaf be the same node as the tree root). This works because only the first k attributes of Π_v are relevant if v is at distance k from the tree root, so we redefine those orders Π_v as the corresponding re-orders from u. The sets of hairy cycles are precisely the graphs where all nodes have indegree 1. □

We next show some techniques for proving lower and upper bounds on order indexes.

Table 6. Minimum size of order indexes to support wco algorithms depending on the index capabilities. Upper bounds with asterisks were not obtained with exhaustive searches, so there could be smaller ones.

d	Unidirectional				Bidirectional		
	TW	CTW	CTWO	L. Bound	CBTW	CBTWO	L. Bound
2	4	2	2	2	2	2	2
3	12	6	6	6	3	3	3
4	32	16	15	12	8	8	8
5	80	40	33	30	25	20	20
6	192	78*	68*	60	42	42	42
7	448	168*	162*	140	84*	84*	70
8	1024	400*	360*	280	200*	209*	168

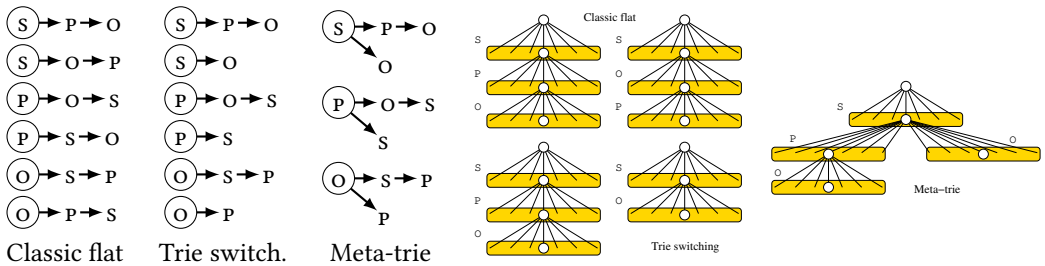


Fig. 12. On the left, the different trie schemes. On the right, the corresponding tries SPO and SO(P); each rectangle leads to storing up to n trie nodes. Note the meta-trie describes the structure of the actual tries.

Table 6 shows the upper and lower bounds we could obtain. The lower bounds are formal results that hold for every possible order index. The upper bounds are obtained allowing only sets of size-d rings (CTW and CBTW, as in Section 6.2.1), improved trie switching (TW, see Section 7.2.1), and a single cycle (CTWO and CBTWO). Those are produced by combining formal bounds with exhaustive and gradient search. The bounds obtained with exhaustive search can still be larger than the lower bounds since the lower bounds are not tight, or the upper bounds explore only a specific kind of order index. The best order indexes we found are listed in Appendix D.

As can be seen, the use of a single cycle yields a unidirectional order index of size 15 for $d = 4$, where the smallest unidirectional index in Table 5 is of size 16 (i.e., 4 cycles). The difference is more clear on $d = 5$, where single-cycle order indexes of sizes 20 and 33 exist for the bidirectional and the unidirectional case, respectively; this is significantly smaller than the best possible solutions using multiple rings, which are of sizes 25 and 40, respectively. The gain is not monotonic on d, however: the smallest bidirectional indexes of both kinds for $d = 6$ are of size 42, and there is no smaller order index. The situation for the unidirectional case and for larger d is unclear, although in most cases we have found better solutions for single-cycle indexes than for multiple rings. This shows that in many cases we can do better by using order indexes other than sets of rings.

7.2 Lower and upper bounds on index sizes

We now generalize the lower and upper bounds of Section 6.2.1 to order indexes.

7.2.1 Trie switching strikes back. Since we are now considering indexes that are not just a set of rings, we can also consider tries that are not complete, and that share their first levels with others. This noticeably reduces the size of the TW representation, both in practice and asymptotically.

Consider the case of $d = 3$, where 6 tries are necessary for TW. Since each level of each trie stores up to one node per edge of G, its size can be computed as the total number of levels so as to meaningfully compare it to the size of order indexes (actually, this can be pessimistic for TW). Since we require 6 tries, the total size of TW is 18 (6 tries of 3 levels each). In fact, trie switching did not apply for $d = 3$. However, consider the following set of tries: SPO, PS, OSP, SO, POS, OP, with total size 15. For example, we do not need to store the full trie PSO because, once we reach a leaf in the trie PS, we can switch to the corresponding internal node in SPO. Further, the tries SPO and SO can *share* their first level, that is, each node in the first level, corresponding to a subject, is the root of two subtries, that of PO and that of O. To distinguish them we can, for example, put in S all the children by P followed by all the children by O. With such level sharing, the size of TW drops to 12, that is, equivalent to 4 tries instead of 6. A good way to see this is to consider the *meta-trie* $\mathcal{T}$ of the sequences {SPO, PS, OSP, SO, POS, OP} and count its number of (meta-)nodes, since now we may store up to one trie node per edge of G for each meta-node of $\mathcal{T}$.Figure 12 illustrates this scheme.

In order to generalize the computation to any dimension d, we rename the attributes to $\{1, \ldots, d\}$ and define that the only meta-trie paths we will store are those where the attributes are decreasing from the root to every internal path node. Thus, for any particular subset of attributes and any additional attribute, there is a trie in our collection where the subset of attributes is read in decreasing order, and the additional attribute is the leaf. This makes the index complete to implement LTJ in wco time. With this notation, our TW set of meta-trie paths for $d = 1$ is $\{1\}$, for $d = 2$ is $\{21, 12\}$, and for $d = 3$ is $\{321, 312, 23, 213, 13, 12\}$. The size of the TW scheme is then the number of non-root meta-nodes in the meta-trie $\mathcal{T}_d$ storing all those paths, 1 for $d = 1$, 4 for $d = 2$, and 12 for $d = 3$.

To obtain the general formula for the size of TW, assume by induction we have the meta-trie $\mathcal{T}_{d-1}$ and want to compute $\mathcal{T}_d$. Then, (1) for every path of $\mathcal{T}_{d-1}$ we create a new path prepending d to it; (2) we add a leaf child d to every internal meta-node of $\mathcal{T}_{d-1}$; and (3) we add a leaf child d to the only full decreasing path $(d-1)(d-2)\cdots 1$ of $\mathcal{T}_{d-1}$. For example, from the only path $\{1\}$ of $\mathcal{T}_1$, rule (1) creates 21, rule (2) does not apply because $\mathcal{T}_1$ has no internal nodes, and rule (3) creates 12. Now, from the paths $\{21, 12\}$ of $\mathcal{T}_2$ we create $\mathcal{T}_3$: rule (1) creates 321 and 312, rule (2) creates 23 and 13, and rule (3) creates 213; we also maintain the path 12 from $\mathcal{T}_2$, which was not extended by rule (3). This model paves the way to the exact calculation of the TW index size; note that the resulting size corresponds to using just 2^{d-1} classic tries (since each of those is of size d).

Lemma 7.8. *The size of the TW index in dimension d using meta-tries is $2^{d-1}d \in \Theta(2^d d)$.*

Proof. Let $i(d)$ and $h(d)$ be the number of internal and leaf nodes in $\mathcal{T}_d$, respectively. To build $\mathcal{T}_d$, we start with the $i(d-1)$ internal nodes and the $h(d-1)$ leaves of $\mathcal{T}_{d-1}$. Then, rule (1) adds $1 + i(d-1)$ new internal nodes and $h(d-1)$ new leaves (i.e., a copy of $\mathcal{T}_{d-1}$ hanging from node d), rule (2) adds $i(d-1)$ new leaves, and rule (3) extends one path of $\mathcal{T}_{d-1}$, thus incrementing $i(d)$. Overall, $i(d) = 2 + 2i(d-1)$ and $h(d) = 2h(d-1) + i(d-1)$, which given $i(1) = 0$ and $h(1) = 1$ solves for $i(d) = 2^d - 2$ and $h(d) = (d-2)2^{d-1} + 2$. The size of $\mathcal{T}_d$ is then the sum $i(d) + h(d) = 2^{d-1}d$. □

As we see next, the size of our order indexes lies between $\Omega(2^d d^{1/2})$ and $O(2^d d)$, although the precise values in Table 6 show that order index sizes are closer to our lower bounds.

7.2.2 Lower bounds. A lower bound on the number of nodes in a single-cycle order index follows from the observation in Lemma 6.8 that the cycle must contain at least $\binom{d}{l}\lceil (d-l)/2 \rceil$ positions for any l. We now show that this lower bound holds indeed for any order index, thereby leveraging our asymptotic lower bound for those more general indexes as well.

Note that reinterpreting "paths of length l" as "paths with l distinct elements" does not change these lower bounds, which count the amount of starting positions of those paths.

Theorem 7.9. *Any order index on dimension d must be of size at least*

$$\max\left\{d \cdot \left\lceil \binom{d}{l} \frac{\lceil (d-l)/2 \rceil}{d} \right\rceil,\ 1 \le l < d\right\},$$

which when choosing $l = \lfloor d/2 \rfloor$ is $\Omega(2^d d^{1/2})$.

Proof. We first consider single-cycle order indexes. Let P_l be the set of distinct subsets of l attributes out of $[1\,..\,d]$. From the proof of Lemma 6.8, we have that any element of P_l must occur at least $\lceil (d-l)/2 \rceil$ times in the cycle in order to cover all the possible choices of the $d-l$ variables to instantiate next. Now take a particular attribute $\text{A} \in [1\,..\,d]$. Let $P_{\text{A},l} \subseteq P_l$ be the elements of P_l where A occurs. Then $|P_{\text{A},l}| = \binom{d-1}{l-1}$, which counts all the ways to choose the other $l-1$ attributes from the $d-1$ remaining choices. The elements of $P_{\text{A},l}$, together, occur $\binom{d-1}{l-1}\lceil (d-l)/2 \rceil$ times in the cycle. Each occurrence of A in the cycle can belong to at most l occurrences of $P_{\text{A},l}$. Therefore, A must occur at least $\left\lceil \binom{d-1}{l-1} \cdot \lceil (d-l)/2 \rceil / l \right\rceil = \left\lceil \binom{d}{l} \cdot \lceil (d-l)/2 \rceil / d \right\rceil$ times. Adding over all attributes A gives the lower bound, which holds for every l. The bound is simply d times the bound of Lemma 6.8.

We now show that the lower bound holds for general order graphs. Per Lemma 7.7, we can focus on graphs where all the indegrees are 1. Let δ be the difference between the number of edges in the graph and the number of distinct strings labeling subpaths of length l that do not repeat attributes. Our lower bound above for cycles applies to the second value, and serves to lower-bound the number of nodes because $\delta \ge 0$ in sets of cycles (δ is exactly zero if all the length-l subpaths in the cycles contain l different attributes). We now show that $\delta \ge 0$ in general, which proves the theorem, that is, general graphs with the same number of edges do not bring more distinct subpaths to cover the $\binom{d}{l}\lceil (d-l)/2 \rceil$ needed combinations.

Graphs where all the indegrees are 1 are sets of hairy cycles, that is, cycles with trees possibly sprouting from each node, as in Figure 11. Let us consider the process of building such graphs by starting from the cycles and adding one new edge at a time from a node v towards a new leaf u (which may later become an internal node if another edge from u is added). Note that there is exactly one path of length $l-1$ leading to v. Thus, the new edge we add from v to u creates at most one new path of length l, namely the one of length $l-1$ leading to v and extended with the new edge to u. So the value of δ for the original cycle is maintained upon edge insertions. □

The lower bound of Theorem 7.9 is tight for all $d \in [2\,..\,6]$, per Table 6, but our exhaustive search shows that the lower bound of 70 given by the theorem for $d = 7$ is not reachable with single cycles.

The lower bound for the unidirectional case is the same, replacing $\lceil (d-l)/2 \rceil$ by just $(d-l)$. The formula then becomes $d \cdot \binom{d-1}{l}$, which is maximized for $l = \lfloor d/2 \rfloor$, yielding the lower bound $tw(d)$ for the size of unidirectional order indexes. Note that this corresponds exactly to the lower bound $ctw(d) \ge \lceil tw(d)/d \rceil$ we had obtained in Section 6.2.1 for unidirectional cycles, with the difference that an order index does not need to consist of an integral number of cycles of length d.

Note that, although our lower bound holds equally for pure-cycles and general order indexes, there could still be cases where all the best order indexes do not include just sets of cycles. We have not found such a case by brute force and believe it does not happen, though our computation power is limited to the upper bounds for CBTWO and CTWO shown in Table 6.

7.2.3 Upper bounds. Each upper bound obtained in Section 6.2.1 multiplied by d holds for the size of an order index as particular cases. Per Theorem 6.9, the smallest unidirectional and bidirectional cyclic indexes are of size $O(2^d d)$. In Appendix D.3 we give another upper bound that, although of this same order, is stricter than those directly derived from the results of Section 6.2.1.

8 Supporting updates

Our index has been described and implemented as a static structure: it must be rebuilt from scratch in order to reflect changes in the database. This is acceptable in many scenarios, for example, we can maintain a small classic dynamic index with the recent updates and periodically rebuild the static compressed index, possibly in the background or in another server. There are other cases, however, where we need to support fine-grained mixes of queries and updates. In this section we show how to address this case, enabling insertion and deletion of tuples and elements of the universe in time $O(\log U \log n)$ per column and with a slowdown factor of $O(\log n)$ in query times. Our first result is a dynamic version of Theorem 7.4.

THEOREM 8.1. *A dynamic order index built on the order graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of a table $\mathcal{R}$ with n tuples and d attributes, with U the maximum universe size seen so far, can use $|\mathcal{E}|n \log U + o(|\mathcal{E}|n \log U) + O(U)$ bits of space and solve queries Q of m tuple patterns in time $O(Q^* \cdot d^2 m \log U \log n)$, where Q^* is the AGM bound of query Q on $\mathcal{R}$. It can also insert and delete tuples in time $O(|\mathcal{E}| \log U \log n)$.*

If the universe $\mathcal{U}$ grows and shrinks considerably along time, using space proportional to its maximum size so far can be inconvenient. For such a case, we also prove the following result.

THEOREM 8.2. *A dynamic order index built on the order graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of a table $\mathcal{R}$ with n tuples and d attributes can use $|\mathcal{E}|n \log n + o(|\mathcal{E}|n \log n)$ bits of space and solve queries Q of m tuple patterns in time $O(Q^* \cdot d^2 m \log^2 n)$, where Q^* is the AGM bound of query Q on $\mathcal{R}$. It can also insert and delete tuples in amortized time $O(|\mathcal{E}| \log^2 n)$.*

For lack of space, we defer all the details to Appendix E.

9 Conclusions

We have introduced the *ring*: an index that regards the triples of a graph database as cyclic and bidirectional, so that it can simulate the 6 triple orders as one. The ring supports the worst-case-optimal (wco) Leapfrog TrieJoin algorithm (LTJ) for solving basic graph patterns using almost no space on top of the raw triple data, and even in compressed space. Our ring further offers fast on-the-fly statistics for query optimization. Our experiments show that the ring uses a fraction of the space of traditional indexes while ranking amongst the best in terms of query times.

We then generalized the concept of ring to relations with $d > 3$ attributes, where a single ring cannot cover all the $d!$ orders needed to support worst-case-optimal LTJ. We showed, however, that the number of rings that need to be stored is $O(2^d)$, and that it quickly becomes orders of magnitude smaller than traditional prefix-based indexes like B-trees or tries. This enables the use of worst-case optimal algorithms for solving multijoin queries on dimensions that were totally impractical (e.g., only 7 rings, instead of $6! = 720$ classical indexes, are needed for $d = 6$).

We further introduced *order indexes*, which generalize sets of rings to general "order" graphs, where we store columns of the tables in various orders (one column per order graph edge), in a way that LTJ can be implemented in worst-case-optimal time. We have shown that order indexes can be more space-efficient than sets of rings, reducing space by up to 20%.

Our findings demonstrate that spending a lot of space is not necessary to achieve, and even improve, the best current performance in wco indexing of graph databases. Further, we open up many interesting lines of research regarding time versus space trade-offs in the context of wco join algorithms. We finish with some concrete lines of future work regarding the ring and order indexes.

Future work. We have found the optimal-size rings only for $d \leq 6$, and as such the problem of how to find the minimal indexes for larger d is interesting, as well as better understanding the nature and possibilities of order graphs. However, our lower bounds show that even the smallest

order graphs become impractical for $d \geq 8$. A more practical line of research is to explore the trade-off between maintaining high-arity relations, which necessitate fewer joins but larger indexes, and decomposing those relations into several lower-arity relations, which need smaller indexes but more joins, and whose AGM bound is higher. As such, our new indexes enable trade-offs using higher-arity relations, and thus faster queries, given an allowed memory footprint.

Our focus has been on indexing for wco joins, where our query planning strategy is currently based on simple methods. Future work could explore further techniques from the literature, such as tree decompositions for variable ordering [1], low-level caching techniques to reuse intermediate results [34], adaptive plans that use statistics collected during query evaluation [42], and hybrid plans that combine wco and non-wco join algorithms for higher-arity relations [24, 42], among others. Our index may further support custom optimizations. For example, a useful statistic would be to find how many different elements are associated with a column range, which can be computed, at least for backward extensions, in logarithmic time by roughly doubling the space [25]. Supporting further query operators, such as projection, regular path queries, aggregation, etc., would also be of interest, particularly regarding the possibilities of pushing such operators to low-level operations on the index. A recent spin-off of our results, for example, shows that the ring can efficiently support regular path queries [5]; combining those with wco joins is a formidable challenge.

Acknowledgments

We thank Amine Mhedhbi for help with Graphflow and Daniela Campos for help with CompactLTJ. This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002. Gómez-Brandón was supported in part by MCIN/AEI/10.13039/5011000-11033: grants PID2020-114635RB-I00; by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe": PID2022-141027NB-C21; by MCIN/AEI/10.13039/501100011033 and "Next-GenerationEU"/ PRTR: grants TED2021-129245B-C21, PDC2021-120917-C21 and by GAIN/ Xunta de Galicia: GRC: grants ED431C 2021/53, and CIGUS 2023-2026. Hogan was supported by FONDECYT Grant No. 1221926. Navarro was supported by FONDECYT Grant No. 1230755. Reutter was supported by FONDECYT Grant No. 1221799.

References

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. 2017. EmptyHeaded: A relational engine for graph processing. *ACM Transactions on Database Systems* 42, 4, Article 20 (2017), 44 pages.

[2] M. Abo Khamis, H. Q. Ngo, D. Olteanu, and D. Suciu. 2019. Boolean tensor decomposition for conjunctive queries with negation. In *Proc. 22nd International Conference on Database Theory (ICDT)*. 21:1–21:19.

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

[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys* 50, 5 (2017), 68:1–68:40.

[5] D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, and J. Rojas-Ledesma. 2023. Optimizing RPQs over a compact graph representation. *The VLDB Journal* (2023). To appear.

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

[7] Me. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. 2010. Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In *Proc. 19th International Conference on World Wide Web (WWW)*. 41–50.

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

[9] J. Barbay, F. Claude, and G. Navarro. 2013. Compact binary relation representations with rich functionality. *Information and Computation* 232 (2013), 19–37.

[10] M. J. Bauer, A. J. Cox, and G. Rosone. 2013. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science* 483 (2013), 134–148.

[11] A. Bonifati, W. Martens, and T. Timm. 2019. Navigating the maze of Wikidata query logs. In *Proc. 19th World Wide Web Conference (WWW)*. 127–138.
[12] A. Bonifati, W. Martens, and T. Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2-3 (2020), 655–679.
[13] N. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, A. Fariña, and G. Navarro. 2023. Space/time-efficient RDF stores based on circular suffix sorting. *The Journal of Supercomputing* 79 (2023), 5643–5683.
[14] N. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and G. Navarro. 2017. Compressed representation of dynamic binary relations with applications. *Information Systems* 69 (2017), 106–123.
[15] M. Burrows and D. Wheeler. 1994. *A block sorting lossless data compression algorithm.* Technical Report 124. Digital Equipment Corporation.
[16] F. Claude, G. Navarro, and A. Ordóñez. 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems* 47 (2015), 15–32.
[17] O. Curé, G. Blin, D. Revuz, and D. C. Faye. 2014. WaterFowl: A compact, self-indexed and inference-enabled immutable RDF store. In *Proc. 11th European Semantic Web Conference (ESWC)*. 302–316.
[18] O. Erling. 2012. Virtuoso, a hybrid RDBMS/graph column store. *Data Engineering Bulletin* 35, 1 (2012), 3–8.
[19] O. Erling and I. Mikhailov. 2009. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*. Springer, 7–24.
[20] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. 2013. Binary RDF representation for publication and exchange (HDT). *Journal of Web Semantics* 19 (2013), 22–41.
[21] P. Ferragina and G. Manzini. 2005. Indexing compressed texts. *Journal of the ACM* 52, 4 (2005), 552–581.
[22] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms* 3, 2 (2007), 20.
[23] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. 2018. Cypher: An evolving query language for Property Graphs. In *Proc. International Conference on Management of Data (SIGMOD)*. 1433–1445.
[24] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endowment* 13, 11 (2020), 1891–1904.
[25] T. Gagie, J. Kärkkäinen, G. Navarro, and S. J. Puglisi. 2013. Colored range queries and document retrieval. *Theoretical Computer Science* 483 (2013), 36–50.
[26] T. Gagie, G. Navarro, and S. J. Puglisi. 2012. New algorithms on wavelet trees and applications to Information Retrieval. *Theoretical Computer Science* 426-427 (2012), 25–41.
[27] S. Gog, T. Beller, A. Moffat, and M. Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms, (SEA)*. 326–337.
[28] G. Gottlob, S. T. Lee, G. Valiant, and P. Valiant. 2012. Size and treewidth bounds for conjunctive queries. *Journal of the ACM* 59, 3 (2012), article 16.
[29] G. Gottlob, N. Leone, and F. Scarcello. 1999. Hypertree decompositions and tractable queries. In *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 21–32.
[30] R. Grossi, A. Gupta, and J. S. Vitter. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*. 841–850.
[31] S. Harris, A. Seaborne, and E. Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/.
[32] A. Harth and S. Decker. 2005. Optimized index structures for querying RDF from the web. In *Proc. 3rd Latin American Web Congress (LA-Web)*. 71–80.
[33] A. Hogan, C. Riveros, C. Rojas, and A. Soto. 2019. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*. 258–275.
[34] O. Kalinsky, Y. Etsion, and B. Kimelfeld. 2017. Flexible caching in trie joins. In *Proc. 20th International Conference on Extending Database Technology (EDBT)*. 282–293.
[35] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. 2020. Functional aggregate queries with additive inequalities. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–41.
[36] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. 2016. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems* 41, 4 (2016), 22.
[37] M. A. Khamis, H. Q. Ngo, and D. Suciu. 2017. What do Shannon-type inequalities, submodular width, and disjunctive Datalog have to do with one another?. In *Proc. 36th Symposium on Principles of Database Systems (PODS)*. 429–444.
[38] P. Koutris, T. Milo, S. Roy, and D. Suciu. 2017. Answering conjunctive queries with inequalities. *Theory of Computing Systems* 61 (2017), 2–30.
[39] V. Mäkinen and G. Navarro. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms* 4, 3, Article 32 (2008).

[40] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. 2018. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*. 376–394.
[41] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. 2016. Practical compressed string dictionaries. *Information Systems* 56 (2016), 73–108.
[42] A. Mhedhbi and S. Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment* 12, 11 (2019), 1692–1704.
[43] MonetDB. 2013. `http://www.monetdb.org`.
[44] J. I. Munro. 1996. Tables. In *Proc. Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 37–42.
[45] J. I. Munro and Y. Nekrich. 2015. Compressed data structures for dynamic sequences. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*. 891–902.
[46] J. I. Munro, Y. Nekrich, and J. Scott Vitter. 2016. Fast construction of wavelet trees. *Theoretical Computer Science* 638 (2016), 91–97.
[47] G. Navarro. 2014. Wavelet trees for all. *Journal of Discrete Algorithms* 25 (2014), 2–20.
[48] G. Navarro. 2016. *Compact Data Structures – A practical approach.* Cambridge University Press.
[49] G. Navarro, J. Reutter, and J. Rojas. 2020. Optimal joins using compact data structures. In *Proc. 23rd International Conference on Database Theory (ICDT)*. 21:1–21:21.
[50] G. Navarro and K. Sadakane. 2014. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms* 10, 3 (2014), article 16.
[51] T. Neumann and G. Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19 (2010), 91–113.
[52] H. Q. Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th Symposium on Principles of Database Systems (PODS)*. 111–124.
[53] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. 2012. Worst-case optimal join algorithms. In *Proc. 31st Symposium on Principles of Database Systems (PODS)*. 37–48.
[54] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES)*. 2:1–2:8.
[55] D. Olteanu and M. Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45, 2 (2016), 5–16.
[56] G. Ottaviano and R. Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282.
[57] M. Raasveldt and H. Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proc. International Conference on Management of Data (SIGMOD)*. 1981–1984.
[58] R. Raman, V. Raman, and S. S. Rao. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms* 3, 4, Article 43 (2007).
[59] J. Salas and A. Hogan. 2018. Canonicalisation of monotone SPARQL queries. In *Proc. 17th International Semantic Web Conference (ISWC)*. 600–616.
[60] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. 2018. C-store: A column-oriented DBMS. In *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*. ACM, 491–518.
[61] B. B. Thompson, M. Personick, and M. Cutcher. 2014. The Bigdata®RDF Graph Database. In *Linked Data Management*. Chapman and Hall/CRC, 193–237.
[62] N. Tziavelis, W. Gatterbauer, and M. Riedewald. 2020. Optimal join algorithms meet top-k. In *Proc. International Conference on Management of Data (SIGMOD)*. 2659–2665.
[63] N. Tziavelis, W. Gatterbauer, and M. Riedewald. 2021. Beyond equi-joins: Ranking, enumeration and factorization. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2599–2612.
[64] T. L. Veldhuizen. 2014. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*. 96–106.
[65] B. D. Vo and K.-P. Vo. 2007. Compressing table data with column dependency. *Theoretical Computer Science* 387 (2007), 273–283.
[66] D. Vrandecic and M. Krötzsch. 2014. Wikidata: A free collaborative knowledgebase. *Communications of the ACM* 57, 10 (2014), 78–85.
[67] C. Weiss, P. Karras, and A. Bernstein. 2008. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endowment* 1, 1 (2008), 1008–1019.
[68] M. Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*. 82–94.

A Extensions to basic graph patterns

Our Ring structure can easily handle some extensions to the basic graph patterns we study.

Equality selections $x = a$ (which are analogous to constants in triple patterns) can be extended to range selections by leaving x as a variable and allowing the queries to specify a range $a \leq x \leq b$ for any variable x, where a and b are constants. When it turns to bind the variable x in `leapfrog_search`, we use a instead of $\min \mathcal{U}$ in line 3 and add the condition $c \leq b$ in line 4 (see Algorithm 1). To handle lonely variables with a restricted range, the wavelet tree can also obtain all the k distinct values $c \in [a\,..\,b]$ appearing in $S[s\,..\,e]$ in time $O(k \log(U/k))$ [26].

This can be further extended to ranges over different variables. For example, a condition like $x \leq y$ can be handled by adding the range constraint $x \leq y_0$ if x is instantiated after we bind $y := y_0$, or by adding $y \geq x_0$ if y is instantiated after we bind $x := x_0$. More complex constraints like $x \leq y + z$ can be handled similarly, by adding a range on the last variable that is instantiated.

While likely to be practical, the worst-case optimality of such an approach is far from obvious. Khamis et. al argue that variants of LTJ algorithms are not able to achieve optimal running times for range inequalities, even if they support enumerating only those elements that satisfy them [35], and that one must use instead join algorithms capable of exploring multiple query plans at once, such as PANDA [37]. However, it may be also be possible to use the ring, and its range functionalities, to enable hybrid approaches for queries with range inequalities, such as those in [38, 63].

B Detailed experimental setup

In the following we give further details on the experimental setup. Materials including scripts, code, queries and data can be found on the webpage http://68.183.136.91/.

B.1 The Wikidata graphs

We take the same Wikidata graphs as proposed for the Wikidata Graph Pattern Benchmark (WGPB) [33], which can be downloaded from https://zenodo.org/record/4035223. The graph is based on the 2018/11/18 truthy version of Wikidata, where the raw dump contains 3,303,288,386 triples. Multilingual labels, aliases and descriptions were removed, leaving only English labels. The result is a graph of 969,496,651 triples with 5,419 unique predicates; this was the graph used in our paper for the Wikidata real-world experiments. In the graph recommended for the WGPB experiments [33], triples whose predicates appear in fewer than 1,000 triples or more than 1,000,000 triples were also removed.

B.2 System details

In the following we provide additional details about the alternative indexes and systems that we compare with Ring and C-Ring.

Blazegraph: We use version 2.1.6 with the HTTP interface, obtaining the code from https://github.com/wikimedia/wikidata-query-rdf/blob/master/docs/getting-started.md.

DuckDB: We use version 0.8.1, obtaining the code from https://duckdb.org/#quickinstall and sending queries via the Python client library.

EmptyHeaded: We obtained the code from https://github.com/HazyResearch/EmptyHeaded. Data were indexed in memory.

Graphflow: We obtained the code from https://github.com/queryproc/optimizing-subgraph-queries-combining-binary-and-worst-case-optimal-joins/. Data were indexed in memory.

Qdag: We choose the version that uses BFS enumeration of the graph nodes and threading. We obtained the code from the authors.

Jena: We use Jena TDB version 3.10.0, from https://github.com/apache/jena. We use the HTTP interface.

Jena LTJ: We obtained it from https://github.com/cirojas/jena-leapfrog and used in the same way as Jena. It is based on Jena TDB version 3.10.0.

Postgres: We use version 13.3, obtaining the code via an Ubuntu package and sending queries via the command line interface.

RDF-3X: We use version 0.3.7 with the command-line interface (to the best of our knowledge, HTTP is not supported), obtaining the code from https://code.google.com/archive/p/rdf3x/.

Virtuoso: We use version 7.2.5.1 with the HTTP interface. The code was downloaded from https://github.com/openlink/virtuoso-opensource.

In the case of the systems with HTTP interfaces we also tried running queries using command-line interfaces to eliminate HTTP overhead, but found that the HTTP interfaces in general offered better performance (particularly in the case of Jena and Jena LTJ). The systems were configured per vendor recommendations for the machine used. We refer to http://68.183.136.91/ for further details on how we configured and ran these systems.

B.3 Query sets

We use the standard WGPB queries — which consist of 50 instances of 17 abstract query patterns, each generating at least one result and limited to 1000 results — without modification. We refer to Hogan et al. [33] for further details on the generation of these queries; we downloaded them from https://zenodo.org/record/4035223.

For real-world experiments, in search of challenging queries, we download the 122,980 queries that gave timeouts from the Wikidata query logs [40] spanning from June 2017 to March 2018. We remove Wikidata-specific features (e.g., `SERVICE` clauses for labels) and extract basic graph patterns from queries with precisely one basic graph pattern. We subsequently filter the patterns to ensure that they are weakly connected (avoiding Cartesian products), that they have at least one variable, and that their constants appear in the dataset. We chose not to filter queries with empty results as these often occur in practice. We also canonically label the variables of the patterns and de-duplicate them modulo isomorphism [59]. We project all variables and limit the results to 1000 (per WGPB). This process yielded 1,300 queries for testing. We provide an example of one of the more challenging cases in Figure 13, which looks for information about people who died on the same date. The average number of triple patterns and variables per query was 2.07 and 3.49, respectively; in Figure 14 we present box-plots for these numbers where although most queries have only 1 or 2 triple patterns and variables, more complex queries have up to 15 triple patterns and 30 variables. In Table 7 we show the most common types of triple patterns; we find all possible types (aside from all constants), where 221 triple patterns (8.1%) have variable predicates and 5 of those involve joins on those variable predicates.

C Time distributions for fixed-predicate indexes

Figure 15 shows the distributions of the original Ring and CRing, as well as their variants optimized for fixed predicates and the optimized variant of Qdags.

D Minimal order graphs

Figure 16 shows the five non-isomorphic order graphs of minimal size 6 for triples, in the unidirectional case (i.e., we can only move backwards). It includes using two rings, a cycle of length 6, and various single hairy rings.

```
SELECT * WHERE {
  ?v1 wdt:P21 ?v3 .   # sex or gender
  ?v1 wdt:P31 wd:Q5 . # instance of human
  ?v1 wdt:P570 ?v2 .  # date of death
  ?v1 wdt:P734 ?v0 .  # family name
  ?v4 wdt:P21 ?v3 .   # sex or gender
  ?v4 wdt:P31 wd:Q5 . # instance of human
  ?v4 wd:P570 ?v2 .   # date of death
  ?v4 wd:P734 ?v0 .   # family name
} LIMIT 1000
```

Fig. 13. A difficult real-world query in SPARQL syntax

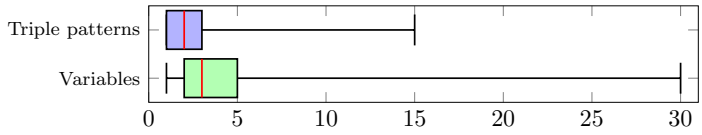


Fig. 14. Box-plots of number of variables and triple patterns in real-world queries

Table 7. Most common types of triple patterns where ? indicates a variable and s, p, o indicate constants

Type	Count	Perc.
?p?	1,449	53.1%
?po	1,012	37.1%
???	146	5.4%
sp?	45	1.7%
s??	40	1.5%
??o	33	1.2%
s?o	2	0.07%
Total	2,727	

The rest of the section shows the smallest bidirectional order indexes we found, both in the form of rings and cycles. For the lower dimensions, where we found all the minimal solutions via exhaustive search, we show all the nonredundant solutions. We consider two solutions redundant if they are isomorphic as order graphs, that is, if one can be obtained from the other by renaming the indices (and taking into account circularity and bidirectionality); rings $1,2,3$ and $2,1,3$ are the same upon exchanging 1 with 3 and shifting, for example.

D.1 Rings

For $d = 1$, $d = 2$, and $d = 3$, the only minimal solution is a single ring: 1; $1,2$; and $1,2,3$. For $d \geq 4$ we need more than one ring. For $d = 4$, in particular, two rings suffice, and exhaustive search shows that there is only one solution if we remove redundant ones:

$$1,2,3,4$$
$$1,2,4,3$$

For example, by exchanging 1 with 2, this solution matches the one listed in Figure 11, where $2,1,3,4$ appears in the lower ring and $2,1,4,3$ appears in the upper ring.

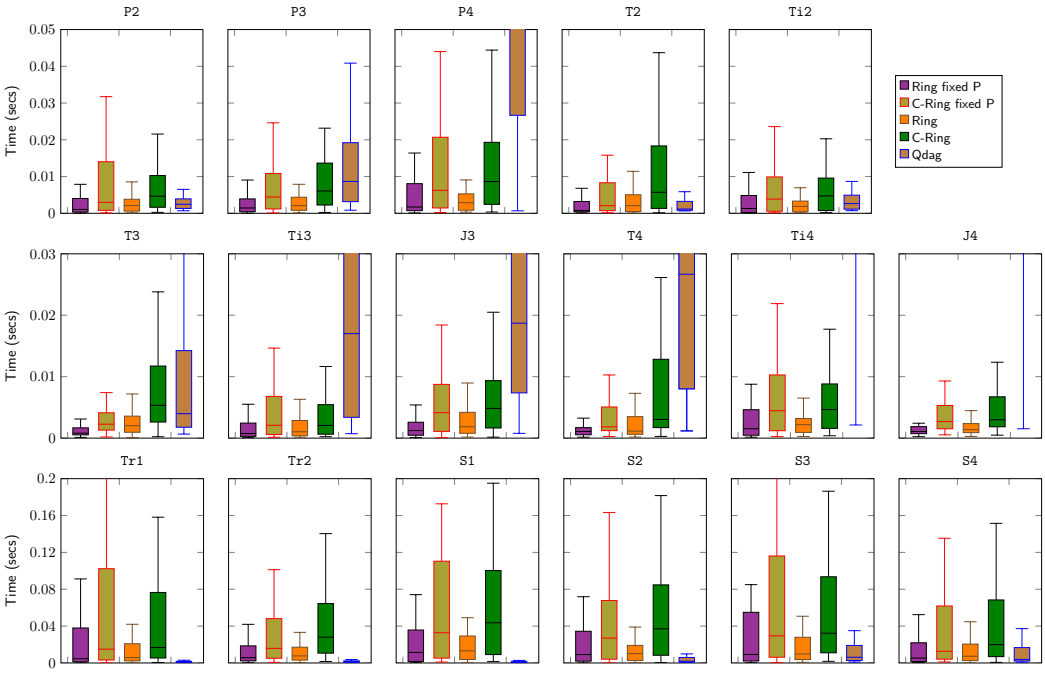


Fig. 15. Comparison of query times (in seconds), now including fixed-predicate structures

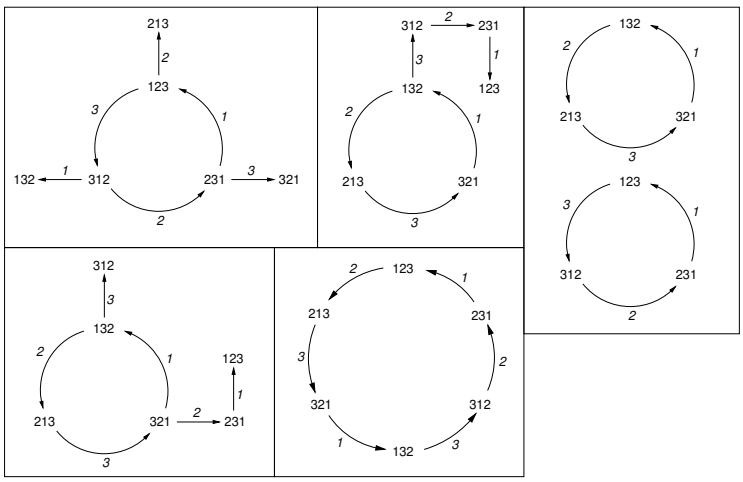


Fig. 16. The five non-isomorphic complete unidirectional order graphs of minimum size 6 for triples ($d = 3$).

Exhaustive search also shows that we need five rings for $d = 5$; the following are the (only) 3 nonredundant solutions of this size:

```
1,2,3,4,5   1,2,3,4,5   1,2,3,4,5
1,2,3,5,4   1,2,3,5,4   1,2,4,5,3
1,2,4,3,5   1,2,4,3,5   1,2,5,3,4
1,2,5,4,3   1,3,2,5,4   1,3,2,5,4
1,3,5,2,4   1,3,4,2,5   1,3,4,2,5
```

The 5 rings of each solution amount to 25 elements, whereas our lower bound is 20. Still, no set of 4 rings (with total size 20) suffices for $d = 5$. The lower bound is instead matched with single cycles, as we see in Section D.2.

We need seven rings for $d = 6$. Our exhaustive search finds 7 nonredundant solutions:

```
1,2,3,4,5,6 1,2,3,4,5,6 1,2,3,4,5,6 1,2,3,4,5,6 1,2,3,4,5,6 1,2,3,4,5,6 1,2,3,4,5,6
1,2,3,5,6,4 1,2,3,5,6,4 1,2,3,6,5,4 1,2,4,3,6,5 1,2,4,3,6,5 1,2,4,3,6,5 1,2,4,3,6,5
1,2,4,5,6,3 1,2,4,6,3,5 1,2,4,6,3,5 1,2,5,4,6,3 1,2,5,6,4,3 1,2,5,6,4,3 1,2,5,6,4,3
1,2,6,4,3,5 1,3,2,6,5,4 1,3,2,5,6,4 1,3,5,6,2,4 1,2,6,5,3,4 1,2,6,5,3,4 1,2,6,5,3,4
1,3,4,2,6,5 1,3,6,2,4,5 1,3,6,2,4,5 1,4,3,5,2,6 1,3,2,5,4,6 1,3,2,6,4,5 1,3,5,2,4,6
1,4,3,6,2,5 1,4,3,5,2,6 1,4,3,5,2,6 1,4,6,2,3,5 1,3,6,2,4,5 1,3,5,2,4,6 1,3,6,2,4,5
1,4,5,2,3,6 1,5,2,4,3,6 1,5,2,4,3,6 1,5,4,2,3,6 1,4,6,2,3,5 1,4,5,2,3,6 1,4,5,2,3,6
```

This number of rings amounts to 42 elements in total, which matches our lower bound that holds for every order index.

We were unable to run an exhaustive search for the rings on $d = 7$. Instead, we used an approximation of the set cover problem to find the following solution, which uses 12 rings:

```
1,2,7,3,6,5,4  1,2,6,5,4,7,3  1,2,7,3,5,4,6  1,3,6,4,2,5,7  1,4,7,2,6,3,5
1,2,5,4,3,6,7  1,3,5,7,6,2,4  1,5,4,3,2,7,6  1,3,5,2,7,6,4  1,4,3,2,5,6,7
1,5,7,4,2,3,6  1,2,3,4,7,6,5
```

This solution, of size 84, does not match our lower bound of 70.

The largest dimension we have explored is $d = 8$, where our approximation found a solution with 25 rings, for a total size of 200:

```
1,3,6,7,8,4,2,5  1,2,3,7,5,4,6,8  1,4,3,8,5,6,2,7  1,4,7,8,2,3,5,6  1,5,8,2,6,4,3,7
1,6,2,3,4,5,8,7  1,4,2,7,5,6,3,8  1,2,6,8,7,3,5,4  1,5,6,7,4,2,3,8  1,3,2,5,8,4,6,7
1,3,8,7,5,2,4,6  1,2,8,4,6,3,7,5  1,4,5,6,3,2,7,8  1,3,5,8,7,6,2,4  1,6,5,2,3,7,4,8
1,2,3,4,8,6,5,7  1,2,6,8,3,7,4,5  1,3,2,5,4,7,8,6  1,3,4,6,5,2,8,7  1,2,7,6,3,4,5,8
1,4,6,8,3,2,5,7  1,5,3,8,2,4,7,6  1,5,4,3,8,7,2,6  1,3,7,6,4,8,2,5  1,4,7,3,2,6,8,5
```

This is far from our lower bound of 168.

D.2 Cycles

For $d = 1$, $d = 2$, and $d = 3$, the only minimal solution is a single cycle of length d: 1; 1, 2; and 1, 2, 3. The minimal cycles for $d = 4$ require 8 elements, just as with rings. The following are the only two minimal nonredundant cycles for $d = 4$, found by exhaustive search (cf. Figure 11):

```
1,2,3,4,1,2,4,3
1,2,3,4,2,1,4,3
```

Exhaustive search finds 3 minimal nonredundant cycles of length 20 for $d = 5$:

```
1,2,3,4,5,1,3,4,1,2,5,3,2,4,1,5,2,4,5,3
1,2,3,4,5,1,4,2,3,5,2,1,3,5,4,2,5,1,3,4
1,2,3,4,5,1,4,2,3,5,2,1,4,3,1,5,2,4,5,3
```

Each cycle is smaller than the size 25 of the smallest possible set of rings and matches our lower bound, which holds for every order index.

We could not run an exhaustive search for the minimum cycles in dimension $d = 6$. Instead, we used a gradient descent technique to find the following cycle:

```
1,2,3,4,5,6,3,4,1,6,2,5,3,1,4,2,6,3,5,1,4,6,5,2,1,3,6,4,2,5,1,6,3,2,4,5,6,1,2,3,5,4
```

The cycle is of size 42, which matches our lower bound, and thus our gradient descent found an optimal solution. We remark that, for all $d \leq 6$, we have found cycles whose size matches the lower bound that holds for every order index.

The smallest cycle we found for $d = 7$ using gradient descent is of size 84, just like the smallest set of rings we found and larger than our lower bound of 70:

```
1,2,3,4,5,6,7,2,3,4,7,5,1,2,6,4,3,7,1,5,6,4,2,7,3,1,6,5,2,7,4,1,3,6,2,5,4,1,7,6,3,2,
4,5,7,3,2,1,6,5,4,7,6,2,1,3,5,7,2,1,4,6,3,5,1,4,6,7,3,5,2,1,4,3,5,7,6,1,4,2,3,5,6,7
```

We conjecture that smaller cycles exist closer to the lower bound of 70, but our exhaustive search has shown that no cycle of size 70 exists for $d = 7$.

The largest dimension we have explored is $d = 8$, where we found a cycle of length 209:

```
2,3,6,7,1,8,3,2,6,4,5,8,5,1,3,2,6,1,4,5,6,3,7,1,5,4,8,3,7,1,5,2,8,4,6,7,3,2,5,8,1,4,
6,3,8,4,2,1,5,7,2,8,3,4,1,5,3,8,6,7,1,4,3,5,7,2,6,1,4,7,5,2,8,5,3,4,8,6,7,3,5,2,1,6,
8,4,7,3,2,1,8,5,6,3,4,2,1,7,5,6,4,8,1,2,7,3,8,6,2,4,1,3,6,5,2,4,7,3,1,8,6,7,2,4,5,1,
3,6,8,7,5,4,2,3,1,8,5,7,4,3,6,1,7,2,8,6,5,4,3,2,8,5,6,7,1,3,2,5,4,8,7,3,5,1,6,4,7,2,
3,5,6,7,4,8,1,3,6,2,7,8,5,3,6,2,8,1,4,7,5,6,2,8,1,7,4,2,6,5,1,7,8,2,4,5,1,6,8,7,4
```

This is larger than the best solution based on rings we found, of size 200, and far from our lower bound of 168. We again conjecture that smaller cycles exist closer to the lower bound.

D.3 A refined upper bound for cyclic indexes

The upper bound of Lemma 6.7 translates into the following stricter bound for single-cycle indexes.

Lemma D.1. *There exists a single-cycle unidirectional order index of size $O(2^d d)$, more precisely, $f(\lfloor d/2 \rfloor, \lceil d/2 \rceil) + f(\lceil d/2 \rceil, \lfloor d/2 \rfloor)$, where*

$$f(x,y) = \left(sw(x) \cdot \lfloor x/2 \rfloor + \sum_{l=\lfloor x/2 \rfloor + 1}^{x} \binom{x}{l} \right) \cdot tw(y) + \left(tw(y) \cdot (\lceil y/2 \rceil - 1) + \sum_{l=\lceil y/2 \rceil}^{y} \binom{y}{l} \cdot (y-l) \right) \cdot sw(x)$$

Proof. In Lemma 6.7 we build $sw(\lfloor d/2 \rfloor) \cdot tw(\lceil d/2 \rceil) + sw(\lceil d/2 \rceil) \cdot tw(\lfloor d/2 \rfloor)$ unidirectional rings of length d, which is shown to be $O(2^d)$. Those rings are formed by pairing, for $c := \lfloor d/2 \rfloor$, all the $sw(c)$ elements of a c-sufficient set for the first half of symbols with all the $tw(d-c)$ elements of a $(d-c)$-complete set for the second half of symbols, and vice versa. Since we do not use those rings in cyclic form to support LTJ, they can be concatenated to form a single unidirectional cycle of length $d(sw(\lfloor d/2 \rfloor) \cdot tw(\lceil d/2 \rceil) + sw(\lceil d/2 \rceil) \cdot tw(\lfloor d/2 \rfloor)) \in O(2^d d)$.

This calculation can be tightened by noting that we do not need all those rings to be of length d. Let $s_1, \ldots, s_{sw(c)}$ be the lengths of the strings in the (c,c)-sufficient set and $t_1, \ldots, t_{tw(d-c)}$ be the lengths of the strings in the $(d-c)$-complete set. Then the total length of the rings we must produce is $\sum_{i,j}(s_i + t_j) = tw(d-c) \cdot (\sum_i s_i) + sw(c) \cdot (\sum_j t_j)$.

As per Lemma 6.6, we need to produce only $\binom{c}{l}$ suffixes of length $l = \lfloor c/2 \rfloor + 1, \ldots, c$ in order to build all the needed strings in the c-sufficient sets. Therefore, to contain all the needed suffixes, we can build all the $\binom{c}{\lfloor c/2 \rfloor}$ necessary strings of length $\lfloor c/2 \rfloor$, and extend only $\binom{c}{l}$ of those to length l, for each $l = \lfloor c/2 \rfloor + 1, \ldots, c$. The total length obtained is then $\sum_i s_i = sw(c) \cdot \lfloor c/2 \rfloor + \sum_{l=\lfloor c/2 \rfloor + 1}^{c} \binom{c}{l}$, which, multiplied by $tw(d-c)$, corresponds to the first part of the formula we are proving.

Similarly, Lemma 6.3 shows that we need to produce only $\binom{d-c}{l}(d-c-l)$ prefixes of lengths $l = \lceil (d-c)/2 \rceil, \ldots, d-c$, to build all the needed strings in the $(d-c)$-complete sets. The analogous

calculation then yields $\sum_j t_j = tw(d-c) \cdot (\lceil (d-c)/2 \rceil - 1) + \sum_{l=\lceil (d-c)/2 \rceil}^{d-c} \binom{d-c}{l}(d-c-l)$, which, multiplied by $sw(c)$, leads to the second part of the formula in the lemma.

The formula $f(\lfloor d/2 \rfloor, \lceil d/2 \rceil)$ corresponds to combining a c-sufficient set with a $(d-c)$-complete set; $f(\lceil d/2 \rceil, \lfloor d/2 \rfloor)$ corresponds to a $(d-c)$-sufficient set combined with a c-complete set. □

The lemma yields constructive upper bounds for CTWO indexes that are about twice the size we had obtained in Table 6 via exhaustive search and approximations.

Another upper bound for the size of the smallest unidirectional cyclic index is $2s + 2d - 2$, where s is the size of any bidirectional cyclic index: Given such a bidirectional cyclic index $v_1, \ldots, v_s$, cut it at any point v_s, and build a unidirectional cyclic index by concatenating $v_1, \ldots, v_s, v_1, \ldots, v_{d-1}$ with its reverse. For any segment that the bidirectional index had to extend in backward direction, there is an equivalent segment in the copy that can be followed in forward direction.

E Supporting updates – the details

We show how to obtain the results claimed in Section 8. The bitvectors of length n we described in Section 2.3.1 can be implemented so that they support bit insertions and deletions, in addition to supporting access, `rank`, and `select`, all in time $O(\log n)$. The space stays $n + o(n)$ bits, and even $m \log_2(n/m) + O(m) + o(n)$ bits for sparse bitvectors with m 1s [39]. A wavelet tree (Section 2.3.2) implemented on those dynamic bitvectors supports insertion and deletion of symbols in a sequence $S[1..n]$ over alphabet $[1..U]$, as well as access, `rank`, and `select` queries, all in time $O(\log U \log n)$ [39]. The other algorithms we describe in Section 2.3.2, like those to solve *range-next-value* or to extract all the distinct values in a range [26], can also be implemented directly on the dynamic bitvectors, and thus with the same $O(\log n)$ slowdown factor.

Using those dynamic wavelet trees, we can then obtain the same results of Theorems 3.13, 6.1, and 7.4, with an $O(\log n)$ slowdown factor in the time complexities.[12] We next show how to support insertions and deletions of tuples and elements of the universe in the most general context of the order indexes of Section 7.

E.1 Inserting and deleting tuples

The first part of inserting a new tuple t in a table is to update the d arrays A_* so as to reflect that there is one more occurrence of each of the values in t. This is better done with the representation as bitvectors D_*, which will now be dynamic: if the value of the jth attribute of t is c, then we must insert a bit 0 at position $\texttt{select}_1(D_j, c+1)$ of D_j. This takes time $O(d \log n)$ in total.

Once the bitvectors D_* are updated, we must determine the position where (the corresponding attribute of) t should be inserted in the column of *some* node of the order graph; this will be called the *anchor node*. For this sake we make use of the same property referred to in Section 7: there must exist a path of d edges in the order graph containing all the labels in $[1..d]$. Let $v_0, \ldots, v_d$ be the nodes in this path, representing orders $\Pi_0, \ldots, \Pi_d$, and $\ell_j = \Pi_j(1)$ be the attribute labeling the edge $v_{j-1} \to v_j$, for $1 \le j \le d$. Further let $C_1, \ldots, C_d$ be the columns corresponding to the edges, $C_j := C_{\ell_j}^{\Pi_{j-1}}$. We now perform d restrictions, exactly as described in the proof of Theorem 7.4 (and as in Lemma 3.14, if $d = 3$), for the values of t, on the columns $C_1, \ldots, C_d$.

[12] It is possible to reduce the times to $O(\log U \log n/(\log\log n)^2)$ using slightly faster access, `rank`, and `select` on wavelet trees [50]. Operation *range-next-value* can be done with $O(\log U/\log\log n + \log\log n)$ bitvector operations [9, Lem. 11], which on dynamic bitvectors becomes $O(\log U \log n/(\log\log n)^2 + \log n)$ time. We prefer to stick to the simpler and more practical results [39], however. Other more efficient dynamic sequence representations [45] do not support *range-next-value*.

If t does not exist in the table, there will be some column C_j where the restricted range becomes empty, $C_j[s\,..\,e]$ with $s > e$. The anchor node is then v_j, in whose order Π_j the tuple t should be inserted at position s. If t exists, on the other hand, it should not be inserted.[13]

We insert the values of t in all the columns C_* starting from the anchor node v_j, by propagating the position s across the order graph edges starting from v_j. If we know that t should be at position s in the order Π_v of a node v and there is an edge from v to u, with order Π_u, labeled with attribute $i = \Pi_u(1)$, then we insert the value of attribute i of t at position $C_i^{\Pi_v}[s]$, and after this is done the position of t in Π_u is $s' := F_i^{\Pi_v}(s)$ (Eq. (2)). This process is analogous to that of inserting a circular string in the Burrows-Wheeler Transform of a dynamic set of circular strings [39, Sec. 3.6].

If the order graph is a single cycle, this process reaches all the edges of the graph. If the graph is formed by a set of rings, as in Section 6.2, we must find an anchor node in each ring and then propagate the position of t inside each ring. Other order graph shapes can be handled as well, for example if each connected component is a cycle containing a path of length d with the d distinct attributes, and there are trees sprouting from the cycle nodes. Those include all the order graphs we have found to be of interest in Section 7. We can then update those order indexes upon the insertion of t in time $O(|\mathcal{E}| \log U \log n)$, where $|\mathcal{E}|$ is the number of edges in the order graph.

Deleting an existing tuple t follows the inverse process. We perform the d restrictions on the edges that connect $v_0, \ldots, v_d$, reaching the position s of t in the order Π_{v_d}. From v_d, which acts as our anchor node, we propagate the deletion. This time, we compute the position s' before removing t: if we know the position s of t in Π_v and there is an edge $v \longrightarrow u$ labeled by attribute i, we compute $s' := F_i^{\Pi_v}(s)$ and only then remove $C_i^{\Pi_v}[s]$ (if v is connected to several nodes, we delete the position after computing s' on all its out-neighbors). Once we have updated all the columns C_*, we remove the 0 at position $D_j[\texttt{select}_1(D_j, c{+}1) - 1]$ for every value c at every attribute j of t. Thus, we delete a tuple from the index in time $O(|\mathcal{E}| \log U \log n)$. Since $d \leq |\mathcal{E}|$, this is the total time complexity.

E.2 Modifying the universe

In graph databases seen as sets of tuples, inserting/deleting tuples corresponds to inserting/deleting graph edges but also, indirectly, to inserting/deleting graph nodes and edge labels. In the general case, this operation may then involve modifying the universe U.

Modifying U is complicated in our representation because the wavelet trees are structured according to a binary partition of $[1\,..\,U]$. We will support the following update operations:

- Doubling the universe range, from $[1\,..\,U]$ to $[1\,..\,2U]$, where U will always be a power of 2.
- Allocating a new element of the universe, within the current range $[1\,..\,U]$.
- Removing an element from the universe, if it does not appear in any tuple.
- Halving the universe range, from $[1\,..\,U]$ to $[1\,..\,U/2]$.

The requirement that U is a power of 2 affects the space usage only sublinearly: our original terms $\log_2 U$ may become now as large as $\lceil \log_2 U \rceil < 1 + \log_2 U$. Using powers of 2 brings important simplifications for expanding the universe because the wavelet trees are perfectly balanced. Doubling U corresponds to prepending a 0-bit to every symbol in each represented sequence S. If we have the wavelet tree W for S, the new wavelet tree of S on alphabet $[1\,..\,2U]$ has a new root node, with a bitvector formed by n 0s, whose left child is W and whose right child is a perfectly balanced wavelet tree with U leaves and all empty bitvectors.

Though simple, this expansion requires $O(n + U)$ time in principle. The $O(U)$ cost, coming from creating the wavelet subtree with empty bitvectors, is easily removed by deamortization: we do not represent the children of a wavelet tree node whose bitvector is empty; we do that only when

[13] If we want to allow for repeated tuples, we can insert the new copy of t anywhere in $[s\,..\,e+1]$ of order Π_d.

some symbol must be represented in that node. If we use wavelet matrices [16] instead of wavelet trees, doubling U is even easier: we just create a new bitvector of n 0s preceding the others.

To eliminate the $O(n)$ time needed to create a bitvector of all 0s, we resort to a format that gap-encodes the number of 0s between consecutive 1s [39, Sec. 3.4]. In this format, a sequence of n 0s is created in constant time and uses only $O(\log n)$ bits. Such gap-encoded bitvectors can also be operated in $O(\log n)$ time and within the same compressed space we use [39, Sec. 5].[14]

As a result, we support doubling the universe range in constant time per (wavelet tree/matrix of) index column, and within the same asymptotic space we have been using.

We maintain the maximum universe element in use, $U' \leq U$. To allocate a new universe element, we just set $U' := U' + 1$ and return U', if this does not exceed U. If U' exceeds U, however, we must double the universe range as explained, so that U becomes $2U$.

For removing an element of $[1\,..\,U]$ that is no longer in use, we simply leave the symbol unused and maintain a gap-encoded dynamic bitvector $R[1\,..\,U]$ marking with 1s the removed symbols. Setting some position $R[p] := 1$ then takes time $O(\log n)$, and we incur $O(U)$ extra bits of space. When a new symbol has to be allocated, we first try to reuse removed symbols: if $p := \mathtt{select}_1(R, 1)$ returns a result, we return p and set $R[p] := 0$. Otherwise, we increase U' and possibly double the universe range, as explained. When we have to double the universe, R is always a sequence of U 0s, so we can easily convert it into a gap-encoded sequence of $2U$ 0s in constant time.

Thus, overall, we allocate and remove universe elements in $O(|\mathcal{E}| + \log n)$ time, using $O(U)$ extra bits of space. Note, however, that U must now be interpreted as the maximum size ever reached by the universe, because it never decreases. We then obtain Theorem 8.1.

If using space and time related to the maximum universe size seen so far is a problem, we must implement the last operation in our list. Concretely, we halve U when R contains less than $U/2$ 0s, that is, less than half the elements in $[1\,..\,U]$ is in use. We then guarantee that $\log_2 U$ is at most 1 more than the logarithm of the current universe size.

Halving the universe range is a costly operation, however. We map every universe element i to $\mathtt{rank}_0(R, i)$, reconstruct all the wavelet trees over alphabet $[1\,..\,U/2]$, and reset R to $U/2$ 0s. If we remove the symbols of the wavelet tree of $S[1\,..\,n]$ one by one, and insert their remapped values into a new wavelet tree, we incur no extra space and the cost is $O(|\mathcal{E}| n \log U \log n)$. Since we do this only after $U/2$ element removals, the amortized cost to remove a universe element is $O(|\mathcal{E}|(n/U) \log U \log n)$. This is too high if U is much smaller than n. An alternative is to reduce the universe from U to $U/(n/U)$ only when the fraction of used elements falls below U/n. The amortized time is then $O(|\mathcal{E}| \log U \log n)$, but in exchange the universe range can be n/U times larger than necessary, and thus all the $\log U$ terms in our space and time complexities become $\log n$. We thus obtain Theorem 8.2.

[14] Their gap encoding result [39, Thm. 6] can create a long bitvector of 0s efficiently, as explained, but only their so-called block-identifier encoding [39, Thm. 7] achieves the desired space when the number of 1s becomes significant. We can obtain the best of both worlds by switching from gap to block-identifier encoding on the bit chunks that become sufficiently dense of 1s. Those chunks are encoded independently and can be re-encoded in time $O(\log n)$, retaining our complexities.