Worst-Case Optimal Graph Joins in Almost No Space

Diego Arroyuelo  
Universidad Técnica Federico Santa María & IMFD  
Santiago, Chile  
darroyue@inf.utfsm.cl

Aidan Hogan  
DCC, Universidad de Chile & IMFD  
Santiago, Chile  
ahogan@dcc.uchile.cl

Gonzalo Navarro  
DCC, Universidad de Chile & IMFD  
Santiago, Chile  
gnavarro@dcc.uchile.cl

Juan L. Reutter  
Pontificia Universidad Católica de Chile & IMFD  
Santiago, Chile  
juretter@ing.puc.cl

Javiel Rojas-Ledesma  
DCC, Universidad de Chile & IMFD  
Santiago, Chile  
jrojas@dcc.uchile.cl

Adrián Soto  
FIC, Universidad Adolfo Ibáñez & IMFD  
Santiago, Chile  
adrian.soto@uai.cl

Abstract

We present an indexing scheme that supports worst-case optimal (wco) joins over graphs within compact space. Supporting all possible wco joins using conventional data structures – based on B(+)Trees, tries, etc. – requires 6 index orders in the case of graphs represented as triples. We rather propose a form of index, which we call a ring, that indexes each triple as a set of cyclic bidirectional strings of length 3. Rather than maintaining 6 orderings, we can use one ring to index them all. This ring replaces the graph and uses only sublinear extra space on top of the graph; in order words, the ring supports worst-case optimal graph joins in almost no space beyond storing the graph itself. We perform experiments using our representation to index a large graph (Wikidata) in memory, over which wco join algorithms are implemented. Our experiments show that the ring offers the best overall performance for query times while using only a small fraction of the space when compared with several state-of-the-art approaches.

CCS Concepts

- Theory of computation → Database query processing and optimization (theory): Data structures and algorithms for data management.

Keywords

Worst-case optimal joins; graph patterns; graph databases; graph indexing; Burrows–Wheeler transform; wavelet trees

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’21, June 20–25, 2021, Virtual Event, China  
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-8345-1/21/06...

1 Introduction

A recent development in the efficient processing of database queries is that of worst-case optimal (wco) join algorithms, which process join queries in time proportional to the AGM bound [6]: the maximum possible output size produced by a join query over a relational database of a certain size. Such algorithms can be strictly better than any traditional query plan using pairwise joins [38, 44].

Leapfrog-TriJoin (LTJ) is a seminal wco join algorithm based on “eliminating” attributes from a query [44]. To illustrate the algorithm (defined in more detail in Section 2.2), consider the relations $R$, $S$, $T$ in Figure 1 along with the query $Q = R \bowtie S \bowtie T$ computing their natural join. In order to evaluate this query, LTJ assumes an ordering of the attributes in $Q$, say $(x, y, z)$ (details of this ordering are discussed later). For the first attribute $x$, LTJ finds all constants $a$ such that the query $\sigma_{x=a}(R \bowtie T)$ gives some solution, here joining all relations that mention $x$ in the join $(R$ and $T)$. In this case $\sigma_{x=1}(R \bowtie T)$ and $\sigma_{x=2}(R \bowtie T)$ give solutions, while $\sigma_{x=3}(R \bowtie T)$ 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 $\sigma_{x=a,y=b}(R \bowtie S)$ 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 in time proportional to the number of results. This is typically enabled by indexing each relation such that we can assign constants to any subset of attributes.
We prove that this implementation enables the evaluation of \( wco \) joins with only one index order, and thus with sublinear space. Inspired by text indexing, and characterise the space it requires.

The order shown, reading the first unbound attribute; in this case, the index order starts. In the (traditional) flat indexing scheme, we have several orders for higher-arity relations (though far fewer than traditional indexing schemes, as discussed later). 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 optimise such features by pushing them to lower-level operations over the index.

2 Related works and concepts

2.1 Graph joins and patterns

2.1.1 Graphs. We adopt a relational view of graphs, treating them as a single ternary relation \( G \). A triple \((s, p, o)\) in the graph encodes an edge \( s \xrightarrow{p} o \) from node \( s \) to node \( o \) labelled \( p \), as seen in Figure 3. We assume the domain of graphs to be drawn from a totally ordered, countably infinite set \( U \) of constants. Given a constant \( u \in U \), we denote by \( u + 1 \) the next element in the total order after \( u \) in \( U \). We denote by \( \text{dom}(G) \) the set of all constants in \( G \).

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 \( U \). A triple pattern is then a triple \((s, p, o)\) in \( (U \cup \mathcal{V})^3 \), and a basic graph pattern is a set \( Q \subseteq 2^{(U \cup \mathcal{V})^3} \) of triple patterns. Each triple pattern represents an atomic query over the graph, and thus a basic graph pattern corresponds to a conjunctive query (aka. join query) over the relational representation of the graph. Let \( \text{vars}(Q) \) 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 \( \rho(Q) = \{ \mu : \text{vars}(Q) \rightarrow \text{dom}(G) | \mu(Q) \subseteq G \} \) called solutions, where \( \mu(Q) \) denotes the image of \( Q \) under \( \rho \); that is, the result of replacing 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 paper is to compute the complete set of solutions for \( Q(G) \) in optimal time and using little space.
for example, the join query \( Q \) of Figure 1. If we first join the pair \( Q_1 = R \bowtie S \) in order to later evaluate \( Q_2 = Q_1 \bowtie T \), 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. Thus no plan involving pairwise joins can be wco. Ngo et al. [38] proposed the first wco join algorithm confirmed to run in time \( O(Q^* \log n) \), called LEAPFROG TRIEJOIN (LTJ) [44].

2.2.3 Leapfrog TrieJoin. As illustrated in the introduction, LTJ evaluates join queries one variable-at-a-time (i.e., one attribute-at-a-time) rather than one relation-at-a-time. Here we provide more details on the LTJ algorithm in the context of graphs [23], referring to the original LTJ paper for the full relational setup [44].

Given a graph \( G \), LTJ is built upon an abstraction for data access called a trie-iterator, which features one operation: 1eat.

Definition 2.1 (Trie-iterator). A trie-iterator for a graph \( G \) is an implementation of \( 1eat : (U \cup \forall)^3 \times V \times U \rightarrow U \cup \{\perp\} \). Given a variable \( x \in V \), a triple pattern \( t \) with the variable \( x \), and a constant \( c \in U \), \( 1eat(t, x, c) \) returns the smallest constant \( c_x \geq c \) from \( U \) such that \( t \) has solutions in \( G \) after replacing \( x \) by \( c_x \). If there is no such value \( c_x \), then \( 1eat(t, x, c) \) returns the special value \( \perp \).

Veldhuizen [44] shows that for LTJ to run in \( O(Q^* \log n) \) time, it suffices for the trie iterators to support 1eat in \( O((log n)^3) \) time (data complexity). Consider a graph pattern \( Q = \{t_1, \ldots, t_m\} \), and let \( v = |vars(Q)| \). For a subset \( S \subseteq vars(Q) \) of variables, further let \( Q_S \) denote the set of triple patterns in \( Q \) that contain at least one variable in \( S \). Algorithm 1 details how LTJ uses the 1eat operation to evaluate a basic graph pattern \( Q \) over a graph \( G \). LTJ first defines an initial ordering \( (x_1, \ldots, x_0) \) of \( vars(Q) \); the specific ordering does not affect wco guarantees, and will be discussed later.

Starting with \( x_1 \), LTJ finds each elimination \( c \in \text{dom}(G) \) for \( x_1 \) such that, for each 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 \( x_1 \) for each individual triple pattern \( t \in Q_{\{x_1\}} \), for which LTJ uses 1eat to recursively seek each triple pattern in \( Q_{\{x_1\}} \) forward to the maximum current value for all triple patterns, returning values for which all triple patterns agree, until the eliminations for triple patterns are exhausted.

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 \( \mu(Q_{\{x_1\}}) \) using the same form of intersection as before based on 1eat. 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 \( \mu \) is then a solution. If no elimination is found for a variable \( x_j \) and current mapping \( \mu \), the process then tries to eliminate \( x_j \) for the next mapping generated up to \( x_{j-1} \). The process terminates when all mappings for \( x_j \) have been exhausted.

We are then left to consider the implementation of 1eat. 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 a relation, and each unique root-to-leaf path encoding a tuple of the relation [44]. 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
Algorithm 1 LEAPFROG for the evaluation of basic graph patterns

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

**Output:** Reports the tuples in $Q(G)$

```plaintext
def leapfrog_join():
    call leapfrog_search((1), 1)

leapfrog_search($\mu$, $j$):
    **Input:** An index $1 \leq j \leq v + 1$, and a mapping $\mu$ defined for the variables $\{x_k | k < j\}$
    **Output:** An index $1 \leq j \leq v + 1$, and a mapping $\mu$ as an output solution
    1: if $j = v + 1$ then return $\mu$ as an output solution
    else
    2: $c := \text{seek}(\mu, j, \text{min } U)$
    3: while $c \neq \bot$ do
    4: $\mu' := \mu \cup \{(x_j := c)\}$
    5: call leapfrog_search($\mu'$, $j + 1$)
    6: $c := \text{seek}(\mu, j, c + 1)$
```

### 2.3 Text-indexing techniques

**2.3.1 Suffix arrays**

Let $T[1..n]$ be a string over an ordered alphabet $\Sigma$, with each $T[j]$ an element in this alphabet except $T[n] = \$$. A suffix array $A$ of $T$ stores all suffixes of $T$ in increasing lexicographic order: if one identifies each suffix $T[i..n]$ of $T$ with the position $i$ at which it starts, then $A$ is a simple permutation of $\{n\}$ that satisfies $A[k] < T[A[k] + 1..n]$ for all $k$. Intuitively, $A[k]$ indicates the index at which the $k^{th}$ lexicographically lowest suffix of $T$ begins. For example, if $T = \$roccos$, then $A = \{3, 5, 2, 4, 6, 1, 7\}$. One use of suffix arrays is to find occurrences of a substring $P$ in $T$. These occurrences, if any, correspond to an interval $A[s..e]$; they are all suffixes of $T$ starting with $P$, thus forming a lexicographic range.

**2.3.2 Suffix arrays for graphs.**

Brisaboa et al. [10] propose to use a compact representation of suffix arrays to evaluate triple patterns. For a graph $G$ with $n$ triples, they identify each constant in the domain of graphs with consecutive integers, and treat $G$ as a string $T[1..3n + 1] = s_1p_1s_2p_2s_3p_3s_4p_4$. To save space, they use an index called a Compressed Suffix Array (CSA) [41]. Using space close to that of the text, this index supports traversing $T$ (which allows one to recreate it), and searching for substrings in time comparable to what is achieved via suffix arrays. They show how to tweak the CSA of a text $T$ representing a graph to represent the triples as cyclic, so that instead of traversing $T$ one can jump from any third component $o_3$ of a triple to its first component $s_1$. This modification allows the authors to evaluate triple patterns by finding the suffix array range $A[s..e]$ that points to the bound parts of the triple patterns, and then recovering the variable parts by simulating a traversal of the cyclic triples $T$ pointed at within $A[s..e]$. While this work could be extended to support wco algorithms like LTJ, because their CSA-based index can efficiently support $\text{leap}$ in only one direction, two orders would be needed (per Figure 2).

Our work continues the idea of using text indexes to support graph queries. However, our indexes are based on FM-indexes and wavelet trees (explained in the following), which, unlike the CSA-based index of Brisaboa et al. [10], enable the bidirectionality needed to support wco joins in just one order. Like most works using compact data structures, we assume the RAM computation model, where the typical arithmetic and logical operations on machine words of $O(\log n)$ bits are carried out in constant time.

---

papers have looked at ways to combine pairwise joins with wco joins, which can help address the space requirements of the latter. Freitag et al. [19] 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 [30] 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.
BWT[1..n] that is a permutation of T: BWT[i] = T[A[i] − 1], except that BWT[i] = T[n] = $ if A[i] = 1. Intuitively BWT[i] indicates the symbol of T directly preceding the ith lexographically lowest suffix of T (or the last symbol of T if no such symbol exists). For example, if T = roccoso$, then BWT = ooccos$.

Let $\Sigma$ = {1, $\tau$}. The BWT allows one to simulate a backwards traversal of T. Assume we know that BWT[i] corresponds to T[j]. Then the element corresponding to T[j−1] is BWT[i′], where

\[
i' = \text{LF}(i) = C[c] + \text{rank}_c(\text{BWT}, i),
\]

(1)
c = BWT[i], C[c] is the number of occurrences of symbols smaller than c in T, and rank$_c$(BWT, i) counts the number of times c occurs in BWT[1..i] (we discuss its implementation soon). Thus, if we know that BWT[i] corresponds to T[1], we retrieve T backwards with T[n] = BWT[i], T[n−1] = BWT[LF(i)], and more generally T[n−k] = BWT[LF$^k$(i)]. The backward traversal of T is thus achieved by repeated applications of the function LF, called LF-steps. For example, with T = roccoso$ and BWT = ooccos$, if we know that BWT[2] refers to T[4] = o, then BWT[LF(2)] = BWT[2+4] = BWT[4] corresponds to the element T[3] = c.

The FM-index uses the BWT to simulate searches in the suffix array. Given a string P[1..n], the goal of this search is to find the maximal interval A[s..e] such that the suffixes of the text represented by this interval are all prefixed by P. This is done in a process called backward search as follows. The interval A[sP[m]..eP[m]] of P[m] is sP[m] = C[P[m]] + 1 and eP[m] = C[P[m]] + 1. Assume that we know the interval corresponding to the occurrences of P[i+1..m]. Then, we can find the interval for P[i+1..m] as follows:

\[
sP[i..m] = C[P[i]] + \text{rank}_c(P[i], \text{BWT}, sP[i+1..m] − 1) + 1,
\]

\[
eP[i..m] = C[P[i]] + \text{rank}_c(P[i], \text{BWT}, eP[i+1..m] − 1).
\]

(2)
This simulates taking simultaneous LF-steps at all the positions $k \in [sP[i+1..m]..eP[i+1..m]]$, where BWT[k] = P[i]: the suffixes starting with P[i..m] are obtained from those starting with P[i+1..m] and preceded by P[i]. If for all $i \in [1..m]$ we have sP[i..m] ≤ eP[i..m], then the range for P is the final A[s..e]. For example, given P = roccoso$, BWT = ooccos$, P = oco, and A = (3, 5, 2, 4, 6, 1, 7), we can find the interval for P[1..3] as follows:

\[
sP[3] = C[o] + 1 = 3,
\]

\[
eP[3] = C[p] = 5,
\]

\[
sP[2..3] = C[c] + \text{rank}_c(\text{BWT}, sP[3] − 1) + 1 = 1,
\]

\[
eP[2..3] = C[c] + \text{rank}_c(\text{BWT}, eP[3] − 1) = 2,
\]

\[
sP[1..3] = C[o] + \text{rank}_o(\text{BWT}, sP[2..3] − 1) + 1 = 3,
\]

\[
eP[1..3] = C[o] + \text{rank}_o(\text{BWT}, eP[2..3]) = 4.
\]


The space required by an FM-index is that used for C and BWT. We can always store C within $n + r + o(n + r)$ bits, as a bitvector $D[1..n+r]$ with i’s at positions i + C[i]; then C[i] is computed as select$_i(D, i)$. This operation finds the position of the ith 1 in D, and is computed in constant time by storing $\delta(D)$ bits on top of D [13, 31]. Regarding BWT, most implementations use a wavelet tree, which requires space close to that of a plain representation of BWT, and can solve rank$_c$(BWT, i) queries in time logarithmic on the alphabet size r [18], enabling search of P in $O(m \log r)$ time.

2.3.4 Wavelet trees. A wavelet tree [22, 34] is a binary tree that represents a string S[1..n] from an alphabet [1, rdemanding $n \log_2 r + o(n \log r)$ bits of space (and can use compressed space). Each node represents a range of alphabet symbols. The root represents [1, rdemanding only the rdemanding of the rth character, which we denote as c. If an internal node represents [a, b], then its left child represents [a, (a + b)] and its right child represents [(a + b)/2 + 1, b]. Conceptually, a node representing [a, b] stores the subsequence $S_a[b]$, if S[a,b] ∈ [a, b]. In practice, this node only stores a bitvector B$_{a,b}$ with a 0 at position i if $S_a[b] \leq (a + b)/2$ and a 1 otherwise (marking whether S[a,b] belongs to its left or right child). The leaves store nothing. Figure 5 exemplifies the wavelet tree for ooccos$: only the bitvectors and pointers are stored, where other elements are included for illustration purposes.

Regarding 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 $\log_2 r$ non-leaf levels, storing all the bitvectors requires at most $n \log_2 r$ bits, just like a plain representation of S. To support operations efficiently (e.g., accessing S[i]), all the bitvectors B$_{a,b}$ are enhanced with o($n_{a,b}$) further bits so that operations rank$_b$ (B$_{a,b}$, i) and select$_b$ (B$_{a,b}$, i) can be answered in constant time [13, 31]. This adds $o(n \log r)$ further bits of space. The pointers of the tree add $O(\log n)$ bits to its space usage (not negligible for large r). This is avoided in a pointerless variant called a wavelet matrix [14], which offers the same functionality in less space.

Operations access, rank, and select are solved by traversing one root-to-leaf path in the wavelet tree, in $O(\log r)$ time. As an example, let us show how to access S[i]: we start with [a, b] = [1, rdemanding and i = i at the root. If B$_{a,b}$[i] = 0, we set b = (a + b)/2, i′ = rank$_b$ (B$_{a,b}$, i′), and continue by the left child. Otherwise, we set a = [(a + b)/2 + 1, i′ = rank$_b$ (B$_{a,b}$, i′), and continue by the right child. When we arrive at a leaf, it holds a = b = S[i], and moreover rank$_b$ S[i] (S, i = i′ (this is used in the LF formula, Eq. (1)).

For example, we can compute BWT[7] (whose wavelet tree is shown in Figure 5) by reading B$_{c,5}$[i′ = 7] = 0 in the root bitvector B$_{c,5}$ = 00010010, thus going left with i′ = rank$_b$ (B$_{c,5}$, 7) = 5. On the left child of the root, we read B$_{o,c}$[i′ = 5] = 1, so we go right with i′ = rank$_l$ (B$_{o,c}$, 5) = 3. Then we arrive at the leaf [0, o], so we know that BWT[7] = o and rank$_o$(BWT, 7) = i′ = 3.

As a more advanced operation, the wavelet tree can return all of the distinct values c in a range S[. . . e], also computing $[s_e] = $ rank$_c$ (S, s = 1 + 1, rank$_c$ (S, e)) for each. The algorithm takes time $O(k (\log r/k))$ to report the k distinct values in S[. . . e] [21].

Finally, wavelet trees support the range-next-value operation in time $O(\log r)$ [21], which we can use for list intersection. Specifically, given a range S[. . . e] and a threshold c ∈ [1, rdemanding, we wish to
find the smallest symbol $c \geq c_x$ that occurs in $S[\ldots e]$. The algorithm also finds the range $[s_c \ldots e]$ so that $S[s \ldots e]$ contains from the $c^\text{th}$ to the $e^\text{th}$ occurrence of $c$ in $S$ (i.e., $s_c = \text{rank}_c(S, s - 1) + 1$ and $e = \text{rank}_c(S, e)$, similar to the backward search in Eq. (2)).

3 One ring to index them all

We now present our ring index scheme for a graph $G$. We sort the resulting triples lexicographically (i.e., by subject, ties broken with predicates, and then ties broken with objects) to obtain all the third components are larger than all the second ones. Finally, that all the second components are larger than all the first ones, and therefore, the lexicographically largest suffix is $T[3n+1\ldots n] = s_1p_1o_1s_2p_2o_2\cdots s_np_no_n$ concatenating each triple and then $S$ (the largest symbol of all).

3.1 The BWT of cyclic triples

We map all the constants in dom($G$) to consecutive integers $[1 \ldots U]$. We then build our index on the resulting set of $n$ integer triples $(s, p, o)$. We add $U$ to all the values $p$ and $2U$ to all the values $o$, so that all the second components are larger than all the first ones, and all the third components are larger than all the second ones. Finally, we sort the resulting triples lexicographically (i.e., by subject, broken with predicates, and then ties broken with objects) to obtain the ordered triples $(s_i, p_i, o_i)$. We then define the text

$$T[1\ldots 3n+1] = s_1p_1o_1s_2p_2o_2\cdots s_np_no_n$$

concatenating each triple and then $S$ (the largest symbol of all).

3.1.1 Bended BWT. Next, we build the BWT of $T$ and slightly modify it so that when we try to find the symbol that precedes $s_1$, it returns $o_1$ instead of $o_{i-1}$ (or instead of $S$ if $i = 1$), thus regarding triples as cyclic. Loosely speaking, we do this by shifting the objects in BWT so that $o_1$ is now in $o_{i-1}$’s place. In the following we write $X \cdot Y$ to denote the concatenation of two strings $X$ and $Y$.

**Definition 3.1.** The bended BWT of BWT$[1 \ldots 3n+1]$ is

\[ \text{BWT}^*[1 \ldots 3n] = \text{BWT}[2 \ldots n] \cdot \text{BWT}[3n+1] \cdot \text{BWT}[n+1 \ldots 3n]. \]

Let $A$ be the suffix array of $T$. The following properties of $A$ and BWT are useful to understand the bended BWT$^*$:

1. Because $s_i < p_j < o_k$ for all $i,j,k$, all the suffixes starting with subjects (i.e., of the form $T[3r+1\ldots]$) precede in $A$ all those starting with predicates (i.e., $T[3r+2\ldots]$), and those precede all the suffixes starting with objects (i.e., $T[3r+3\ldots]$). The lexicographically largest suffix is $T[3n+1\ldots]$ = $S$, so $A[3n+1] = 3n+1$ and thus BWT$[3n+1] = o_n$.

2. (Because the triples are sorted in $T$, each suffix starting with $s_i$ is lexicographically smaller than the one starting with $s_{i+1}$, and thus it holds that $A[i] = 3(i-1) + 1$ for all $1 \leq i \leq n$. Therefore, BWT$[i] = o_{i-1}$ for $2 \leq i \leq n$ and BWT$[1] = S$.)

These two properties, and the way in which BWT$^*$ is obtained from BWT, imply that BWT$^*$ is divided into three zones: the objects, positioned in $[1 \ldots n]$, the subjects, appearing in $[n+1 \ldots 2n]$, and then the predicates in positions $[2n+1 \ldots 3n]$. Moreover, since BWT$[2n+1 \ldots] = o_1\cdots o_{n-1}$ and BWT$[3n+1] = o_n$, the zone of objects is exactly $o_1\cdots o_n$; that is, BWT$^*[n]$ denotes the object of the $n$th triple in the order. This means that BWT$^*$ is of the form

$$\text{BWT}^* = (o_1\cdots o_n) \cdot \text{(subjects)} \cdot \text{(predicates)} \quad (3)$$

**Example 3.2.** Figure 6 shows the index corresponding to the graph of Figure 3. On the top-left, we map dom($G$) to the interval $[1 \ldots 9]$. The top-center shows the resulting set of $n = 13$ triples (e.g. (Bohr,adv,Thompson) becomes $(1, 7, 3)$). On the top-right we see the result of shifting the predicates by $U = 9$ and the objects by $2U = 18$. After sorting the shifted triples, we concatenate them and append $S$ to form the text $T$, where we distinguish identifiers of subjects in blue, of predicates in red, and of objects in green.

Below $T$ we show its suffix array $A$, which is largely divided into zones pointing to subjects, to predicates, and to objects (and $A[40]$ points to the suffix $S$). We show how $C$, below $A$, identifies the areas of the suffixes starting with each different identifier. For example, the suffixes $A[C[6] + \ldots C[6+1]] = A[5\ldots 13] = (13, 16, 19, 22, 25, 28, 31, 34, 37)$ are those starting with 6.

Below $A$, we show BWT, which for each $q$ gives $T[A[q-1]]$. Again, disregarding its extremes, it is divided into three zones, composed only of object, subject, and predicate identifiers, left to right. Below it, we show BWT$^*$, the bended BWT$^*$.

The figure shows how we recover the first triple, $(1, 16, 21)$ (the shifted version of $(1, 7, 3)$) by starting from position $1$. Since BWT$*[1] = 21$, we know that the object is 21. It is the first 21 (i.e., rank$_2$(BWT$^*$, 1) = 1), so we go to the next position after $C[21] = 32$ (i.e., $L^P(1)$ = 33), to find the corresponding predicate at BWT$*[33] = 16$. Since rank$_3$(BWT$^*$, 33) = 3, we compute $L^P(33) = C[16] + 3 = 16$ and find the subject at BWT$*[16] = 1$. Note also that $L^P(16) = 1$ because $C[1] = 0$ and rank$_1$(BWT$^*$, 16) = 1, so if we keep going we cycle over the triple. To find the $n$th triple, we apply the same process starting with BWT$*[n]$.

The next lemma, proved in the supplementary material, shows that this cyclical traversal always occurs, i.e., that our bended BWT is precisely the Extended BWT [29] of our sorted triples.

**Lemma 3.3.** BWT$^*$ sees $T[1 \ldots 3n]$ as the concatenation of $n$ cyclic triples: calling $L^P$ its LF function according to Eq. (1), for every $1 \leq t \leq n$, where $T[A[t] \ldots]$ = $s_t p_t o_t \ldots$, it holds that $T[A[L^P(1)] = o_1, T[A[L^P(L^P(1))] = p_2$, and $L^P(L^P(1)) = t$.

Backward search then works on the set of cyclic triples as well, because each step simulates a batch of LF$^*$ operations. In the sequel we use BWT to denote its bended variant, BWT$^*$, for simplicity.

3.1.2 Space requirements. The ring index of the $n$ triples is composed of the wavelet tree of the bended BWT plus its $C$ array to support LF-steps and backward searches. We note that the BWT representation is not in addition to the raw data, but it replaces the raw data, because we can retrieve the $i$th triple as

$$\text{BWT}[L^P(L^P(i))] = \text{BWT}[L^P(i)] - U, \text{BWT}[L^P(i)] - 2U,$$

where the wavelet trees compute $L^P(\cdot)$ and BWT$[\cdot]$ in time $O(\log U)$.

The representation of the bended BWT$^*$ using wavelet trees requires $3n\log_2(3U) + o(n \log U)$ bits; note $3n\log_2(3U) = 3n\log_2 U + O(n)$. Another $O(n)$ bits are required for the bivector $D$. Note that $O(n) \subset O(n \log U)$ because the $n$ triples are distinct and thus $U = O(n^{1/3})$. The total space used by the ring index is then $3n\log_2 U + o(n \log U)$ bits, which is only sublinear space on top of the $3n\log_2 U$ bits required for the raw representation of the triples.

The BWT can be computed (and bended) in $O(n)$ time and $O(n \log U)$ bits of working space [8]. Its wavelet tree is then built in time $O(n \log U/\sqrt{\log n})$ [33], which is $O(n\sqrt{\log U})$ as $U \leq n$. 

**Theorem 3.4.** Let $|G| = 3n \log_2 U$ be the space in bits of a raw representation of a graph $G$ with $n$ edges and $U = |\text{dom}(G)|$ different identifiers. Then the ring index of $G$ uses $|G| + o(|G|)$ bits of space, and can retrieve any desired edge in $O(\log U)$ time. The ring index is built in $O(n \sqrt{\log U})$ time within $O(|G|)$ bits of working space.

This space is worst-case. Sections 4 and 5 show that we can make the ring index use space close to a compressed representation of $T$ (we explain why in the supplementary material).

### 3.2 Processing joins

We now describe how to support 1eap over the BWT representation of a graph in $O(\log U)$ time, building on the algorithms described in Section 2.3.4. This running time for 1eap implies the worst-case optimality of LTJ (Algorithm 1) over our representation [44]. Thus, in this section we prove the following theorem.

**Theorem 3.5.** On a graph $G$, our ring index evaluates a basic graph pattern $Q$ formed by $m$ triple patterns in $O(Q^* \cdot m \log |\text{dom}(G)|)$ 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 query algorithm is $O(1 + |\text{var}(Q)|)$ words.

Let $Q = \{t_1, \ldots, t_m\}$ be a basic graph pattern. In this section we assume that the constants in each $t_i$ have been appropriately transformed into integers and shifted, summing $U$ to the predicate constants, and $2U$ to the object constants in each $t_i$, as is done for the BWT representation of $G$ (see Figure 6). We further assume that no variable appears more than once in a triple pattern $t_i$; the other case is discussed in the supplementary material. 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 $t$.

**3.2.1 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 a range $\text{BWT}[s..e]$ such that $A[s..e]$ points to all the subjects, all the predicates, or all the objects of $G_t$. The following lemma, proved in the supplementary material, shows how to find $[s..e]$ in $O(\log U)$ time.

**Lemma 3.6.** Let $t = (\alpha, \beta, \gamma)$ be a triple pattern with $0 \leq b \leq 3$ constants. In $O(\log U)$ time, we can find values $s, e$ such that:

- If $|G_t| = 0$, then $s = e = \perp$.
- $e = s + |G_t| - 1$, and $A[s..e]$ points to either the subjects, the predicates, or the objects in $G_t$, whichever is the first following a variable in $t$ if $b < 3$; in particular,
  - if $b = 1$, and $d$ is the constant in $t$, then $T[A[k]] = d$ for all $k \in \{s..e\}$.

**3.2.2 Supporting leaps.** We show how to support 1eap($t_i', x, c$), where $t_i'$ is either a triple pattern $t_i$ from $Q$, or one of its progressively bound versions $\mu(t_i)$ in Algorithm 1.

To evaluate 1eap($t_i', x, c$), we start by obtaining the values $s_i, e_i$ of Lemma 3.6 for $t_i$. If $s_i, e_i = \perp$, we simply return $\perp$. Otherwise, let $t_i' = (\alpha_i, \beta_i, \gamma_i)$. We search for $c \geq c_x$ (recall Definition 2.1) differently depending on where $x$ and the constants in $t_i$ are. If only $\alpha_i$ is a constant and $\beta_i = x$, or only $\beta_i$ is a constant and $\gamma_i = x$, or only $\gamma_i$ is a constant and $\alpha_i = x$, then we will find $c_x$ forwards, because $x$ follows the part of the triple pattern that is already bound. Otherwise, we will find $c_x$ backwards, that is, looking for $x$ preceding the substring that is already bound.

When we process a triple pattern $t_i' = (\alpha_i, \beta_i, \gamma_i)$ backwards, we have the range $A[s_i..e_i]$ pointing to the first constant of the occurrences of $t_i'$, and want to find the smallest $c_x \geq c$ such that some of the (cyclic) suffixes $T[A[k]..s_i]$, $s_i \leq k \leq e_i$, are preceded by $c_x$. Equivalently, we want to find the smallest $c_x \geq c$ in $BWT[s_i..e_i]$. This corresponds to the range-next-value operation of Section 2.3.4, which is supported in $O(\log U)$ time by the wavelet tree of BWT.

When we rather process $(\alpha_i, \beta_i, \gamma_i)$ forwards, only one of $\alpha_i, \beta_i,$ or $\gamma_i$, is bound. By Lemma 3.6, we know that $T[A[k]] = d$ for all $s_i \leq k \leq e_i$. We must thus find the smallest $c_x \geq c$ that follows some of those $d$s in $T$, i.e., $T[A[k] + 1] = c_x$. We solve this by first, finding the leftmost occurrence of $d$ in $BWT[C[c] + 1..]$, where the suffixes in $A$ start with a symbol $c$ or larger. Such an occurrence $q$ is found in $O(\log U)$ time using the wavelet tree, by making $p = \text{rank}_q(BWT, C[c])$ and $q = \text{select}_p(BWT, p + 1)$. This is the position in $A$ of the leftmost suffix $A[q]$ that starts with a symbol
In this section the most relevant engineering and implementation aspects of our ring index. Our implementation is available for reviewing online [5].

4.1 Representation
In order to reduce the size of the universe, we work with a representation that is slightly more involved. Following Eq. (3), we cut the BWT into three components:

$$\text{BWT}[1..3n] = \text{BWT}_0[1..n] \cdot \text{BWT}_s[1..n] \cdot \text{BWT}_p[1..n],$$

where \(\text{BWT}_0\) is formed only by objects, \(\text{BWT}_s\) only by subjects, and \(\text{BWT}_p\) only by predicates. With a little more work and three independent \(C\) arrays, \(C_o\), \(C_s\), and \(C_p\), we can use the identifiers in non-shifted form (i.e., without adding \(U\) and \(2U\)). This reduces the alphabet size, which improves both space and operation time for the wavelet trees. The mapping from \(\text{dom}(G)\) to consecutive integers is done by sorting the triples by predicate (using \(\text{std::sort}\)), and then hashing subjects and objects to ensure uniqueness.\(^3\)

4.2 Algorithm
We implement Algorithm 1 with some improvements on the description of Section 3.2. First, for each \(t'\) we maintain the values \(s_i, e_i\) instead of computing them from scratch during each \(\text{leap}\). We find \(s_i, e_i\) at the beginning of \(\text{leapfrog}\) search as described in Lemma 3.6, and then update them after each further binding of \(t'\). The working space then rises to \(O(mv)\), which is still very low.

The second optimisation is to handle the lonely variables [23], i.e., variables that appear in only one triple pattern \(t\), in a different way: once the other variables of \(t\) 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 \(\text{BWT}[s_i..e_i]\), which is done as described in Section 2.3.4.

4.3 Variable elimination order
The running time of \(\text{LTJ}\) can often sharply depend on selecting a good order in which variables are eliminated. 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 \(t\), our index quickly computes the initial ranges \(A[s_i..e_i]\), so that the number of triples matching the pattern is exactly \(e_i - s_i + 1\). We use this to construct the following elimination order for variables that appear in more than one triple pattern. First we compute \(c(t) = (e_i - s_i + 1)/n\) for each such triple \(t\). Then we estimate the cardinality of each variable \(x\) as \(c_{\text{min}}(x) = \min_t c_{Q(t)}(t)\). The variables \(x\) are then bound by increasing order of \(c_{\text{min}}(x)\), but ensuring that each new variable shares a triple pattern with some previous one, if possible.

4.4 Indexing
We implemented the ring index in C++11 over the succinct data structures library, \(\text{sdsl}\) (https://github.com/simongog/sdsl-lite). To construct the indexes, we first build the suffix array \(A\) with quicksort, and then extract the BWTs. Because the alphabets are generally large, we implemented the wavelet trees as wavelet matrices [14]. We provide two flavours of indexes, with the bitvectors of the wavelet matrices stored either in plain or compressed form. The compressed bitvectors enable high-order entropy compression of the BWT [27]. These bitvectors use a parameter \(b\) in \(\text{sdsl}\); larger values for \(b\) offer better compression but slower operations.

5 Experimental results
We now compare our system – running a modified version of \(\text{LTJ}\) 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 index, where we expect the compressed variant to use less space but to have slower query times.

We run two benchmarks over the Wikidata graph [45], which we choose for its scale, diversity, prominence, data model (it has labelled edges) and real-world query logs [9, 28]. The first benchmark is the Wikidata Graph Pattern Benchmark (WGBP) proposed by Hogan et al. [23] for a sub-graph of Wikidata, with diverse graph patterns. The second benchmark evaluates real-world graph patterns extracted from Wikidata query logs at full scale.

We begin by describing our setup; further details for reproducing experiments are given in the online material [5].

5.1 Experimental setup
Our experiments compare various wco algorithms, as follows.

**Ring and C-Ring:** \(\text{LTJ}\) running over our ring index using plain and compressed bitvectors, respectively. The latter uses parameter \(b = 16\). The system operates in main memory.

**EmptyHeaded:** An implementation [1] of NPPR [38], which is an instance of the same wco algorithm as LTJ [39]. 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 [30].

---

\(^1\)As mentioned in Section 2.3.3, \(C\) might be stored as a bitvector to save space for large alphabets. In this case the binary search is replaced by \(c_x = \text{selects}(D, q) - q\).

\(^2\)In initial experiments with different alphabet orders, we found minor space benefits only for the compressed version of the ring. An interesting future direction would be to explore optimisations based on carefully choosing this order [4].
Qdag: The only previous succinct wco index [35], based on a quadtree representation of the graph that runs in main memory.

We disregard other compressed graph indexes [3, 10] that support only single triple patterns or pairwise joins. For reference we further include results for prominent graph database systems:

**Jena**: A reference implementation of the SPARQL standard. We use TDB, with B+-tree indexes in three orders: spo, pos, and osp. The system supports nested-loop joins.

**Jena LTJ**: An implementation [23] of LTJ on top of Jena TDB. All six different orders on triples are indexed in B+-trees.

**RDF-3X**: The reference scheme [36] 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. RDF-3X handles triple patterns by scanning ranges of triples and uses a query optimiser based on pairwise joins.

**Virtuoso**: A widely used graph database hosting the public DBpedia endpoint, among others [15]. It provides a column-wise index of quads with an additional graph (g) attribute, with two full orders (psog, psg) and three partial indexes (so, op, gs) optimised for patterns with constant predicates. The system supports nested loop joins and hash joins.

**Blazegraph**: The graph database system [42], hosting the official Wikidata Query Service [28]. 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.

The wco algorithms – Ring, C-Ring, EmptyHeaded, Graphflow and Qdag – work in memory with dictionary-encoded numeric constants in the queries, graphs, and results, while the 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 internally they may use dictionary-encoded numeric constants). The results for these popular database systems are included as a point of reference; we discuss the overhead of dictionary decoding/encoding in the supplementary material.

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 then configured per vendor recommendations [5]. All queries are run with a timeout of 10 minutes and a limit of 1000 results (as originally proposed for WGPB [23]).

### 5.2 Graph patterns benchmark

We first run the Wikidata Graph Pattern Benchmark (WGPB) [23], which uses a Wikidata sub-graph 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 7. Each pattern is instantiated with 50 queries built using random walks such that the results are nonempty. The benchmark allows us to compare different alternatives for different abstract patterns. All predicates are constant, all subjects and objects are variables, and each variable appears at most once in the same triple pattern.

#### 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 \( |S| = |O| = 51,999,296 \). Our ring index, in either variant, was built in 12.6 minutes, that is, at a rate of around 6.4 million triples per minute. The working space used for indexing was 2.3 GB, about twice the size of the text \( T \) of triples (which uses 932 MB). The construction of the BWTs takes about a minute; the rest is spent in building the wavelet matrices.

Let us contextualise the space used by the ring index. A simple representation of the dataset using 32-bit integers for all the values requires 12 bytes per triple. A packed representation requires \( \log_2(|S|) + \log_2(|P|) + \log_2(|O|) = 26 + 12 + 26 = 64 \) bits, or 8 bytes, per triple. Our ring index with plain bitvectors privileges time performance, and thus its rank and select structures pose a 57% space overhead, which adding the bitvectors \( D \) sums up to 12.70 bytes per triple, very close to the size of the simple representation. Instead, our ring index using compressed bitvectors with \( b = 16 \) requires 6.68 bytes per triple, less than the packed representation.

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 1 compares the space of the indexes, showing that even the uncompressed Ring uses 5.7–142 times less space than all other non-ring indexes except Qdag, which is also succinct. Qdag, however, uses 32% more space than C-Ring.

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 \( p \times v \) arrays of 32-bit integers, where \( p \) is the number of unique predicates (i.e., edge labels) and \( v \) is the number of unique subject/objects (i.e., nodes), thus requiring \( \Omega(pv) \) space. Hence it was not feasible to load the Wikidata graph for which \( p = 2,101 \) and \( v = 51,999,296 \). 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). \(^4\) 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.

One may compare the space of C-Ring with plain data compressors applied to the text \( T \) of triples. We thus built an additional

---

\(^4\) From personal communication with the first author of the Graphflow paper [30].
C-Ring variant with \( b = 64 \), which uses just 5.35 bytes per triple, close to the space obtained by typical compressors like gzip \( \sim 9 \) (4.83), bzip2 \( \sim 9 \) (4.78) and ppmdj (4.61). Stronger compressors obtain considerably less space, such as rar \( \sim 5 \) (3.52), p7zip (2.67), and a special-purpose front-coding plus \( \delta \)-coding of the differences that we implemented in the RDF-3X encoding (1.87). We do not run queries on C-Ring with \( b = 64 \) because it is considerably slower, for example it takes 73 microseconds to retrieve an arbitrary triple. Still, it can be regarded as a reasonable compression format that supports random access and even (slow) queries.

### 5.2.2 Query times
Table 1 gives the average time taken by each index to sequentially evaluate all the queries. Ring is the fastest index overall, outperforming EmptyHeaded (which uses 142 times more space) by a slight margin\(^5\) and all the other non-succinct indexes by a factor of 2–36. C-Ring is about three times slower than Ring, but it is the smallest index, using about half the space of Ring while still offering faster runtimes than most other indexes. In particular, it is 150 times faster than Qdag, the next smallest index.\(^6\)

Figure 8 shows the distribution of the times for the 17 query types. For readability, the plots omit Jena, which was always slower than Jena-LTJ, showing the benefits for query times of wco joins (per Table 1, Jena-LTJ is about twice as fast overall versus Jena). Considering the median values, Ring is the fastest in 8 of the 17 patterns, Qdag in 5, and EmptyHeaded in the other 4.

Ring is the best, or near the best, in all acyclic queries except for the short paths of P2. Qdag is the best, or near the best, in all queries with just three variables (P2, T2, T12, Tr1, Tr2), but has problems with some of the larger queries, particularly acyclic ones. EmptyHeaded is the best, or near the best, in all cyclic queries, and in some acyclic queries, but has bad cases for long paths (P4) and some tree-like queries (T3, T4, J4). More generally, we observe that Ring offers more stable query times than EmptyHeaded and Qdag: the 75\% percentile never exceeds 0.05 seconds.

---

\(^{5}\)EmptyHeaded does not support limiting the number of results to 1,000. When computing unlimited results, EmptyHeaded outperforms all the other indexes except Ring, which is still faster: 110 vs. 118 milliseconds for EmptyHeaded (Table 1). The absence of this limit is less noticeable in Figure 8, as very few queries exceed 1,000 results.

\(^{6}\)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.

Comparing Ring with Qdag, the latter uses an encoding that grows exponentially with the number of nodes in patterns [35], so the gains of Ring for larger queries is expected. Comparing Ring with EmptyHeaded, we speculate that the advantage of Ring in acyclic queries is mostly due to the locally variables optimisation. To be more precise, consider queries in T4, T14, 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 locally variables optimisation (see Section 4.2). EmptyHeaded rather processes acyclic queries using a version of the traditional Yannakakis algorithm [47]\(^7\), which we speculate is not so well optimised for simple tree-like queries or long paths that may give rise to multiple lonely variables at the end.

C-Ring uses the least space among all the indexes but, 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 7 patterns and slower in 10. As Table 1 shows, however, C-Ring is much more stable than Qdag.

### 5.3 Real-world benchmark
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 [28], and selected queries with a single basic graph pattern, obtaining 1,315 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. Letting \( s, p, o \) denote any constant and \( ? \) any variable, the most common types of triple pattern were \((?, ?)\) (51.5\%), \((?, p, ?)\) (38.3\%), \((?, ?, ?)\) (6.7\%), \((s, ?, ?)\) (1.2\%), \((s, p, ?)\) (1.2\%), \((?, ?, o)\) (1.1\%), \((s, ?, o)\) (0.04\%).

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

---

\(^{7}\)EmptyHeaded works with the generalised tree decomposition of queries, which allows one to decompose complex queries into cyclic parts connected through a tree, where the tree is evaluated using Yannakakis’ algorithm [47].
The ring index is built over this graph in 2.75 hours (5.8 million triples per minute) using 22 GB of RAM. It occupies 12.4 GB (13.86 bytes per triple) with plain bitvectors. Table 2 shows index space and some statistics on the query times obtained on our benchmark. The two wco-join algorithms (Jena-LTJ, Ring) outperform the alternatives. While Ring is the smallest by a factor of 4.3–12 and has the lowest median, Jena LTJ is twice as fast on average, featuring more stable times in this benchmark. The other systems (all of which are non-wco) are slower than Ring for all the statistics.

6 Rings in higher dimensions

Our ring can be generalised to indexing relations with $d$ attributes, provided we can always choose an index where the bounded variables form a range in the tuples. At each stage of the computation, we maintain the BWT range of the leftmost constant. We can extend the range 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 leftmost to the rightmost bounded value for each intersection step. Since there can be $\Theta(d)$ variables in each query tuple, the time that multiplies $O^*$ (the AGM bound) is now $O(d^2 m \log U)$. Unlike in the case of graphs, which have fixed arity, we cannot handle variables appearing multiple times in the same tuple pattern (i.e., equality selections on attributes of the same relation) without incurring a super-exponential penalty factor on arity in the query time. We provide more details in the supplementary material. Such queries are not typically considered by wco join algorithms in a relational setting. In practice, this penalty may be avoided by only considering equalities that occur in some tuple of the relation instance, or accepting non-wco joins in such cases.

**Theorem 6.1.** On a set of $n$ tuples in $[1 \ldots U]^d$, 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 $[1 \ldots U]^d$ (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.

The remaining question is how many orders must the ring index so that, independently of the variable elimination order, the constants are always contiguous in the tuple patterns. We have shown that one order suffices for $d = 3$, but we need more for $d \geq 4$.

A classical index (called flat in Figure 2) on $d$ columns needs to store, in principle, all the $d!$ possible orders to support wco algorithms. We refer to these indexes as class W (for Worst-case-optimal). For $d \geq 4$, we can reduce the number of index orders needed through a strategy we call trie switching, which allows for variables already bound to be reordered. For example, when indexing quads $(s, p, o, g)$, this avoids storing a trie with the order $gsop$ if we have tries for $gsop$ and $sgpo$ since we can process $gs$ with the first index and then switch to the second index (reordering to $sg$) if we require $p$ next. This strategy can be enabled, for example, with inter-trie pointers between the corresponding nodes of $gs$ and $sg$. We call TW the class of indexes that support trie switching.
When the search space was too large, we resorted to approximation wco algorithms depending on the index capabilities on dimensions that would be intractable with classes, we can prove various bounds (see supplementary material). For example, using trie switching, the cyclic unidirectional index [10] is of class CTW and our ring index is of class CBTW. By calling w, tw, cw, ctw, cbw, and cbtw the number of orders that must be indexed by indexes of the corresponding classes, we can prove various bounds (see supplementary material).

Theorem 6.2. The following bounds hold, with $sw(d) = \left( \frac{d}{\log d} \right)^2$:

- $w(d) = d!$ and $cw(d) = (d - 1)!$.
- $\left( \frac{cw(d)}{d^{d-1}} \right) \leq cbw(d) \leq cw(d)/2$ for $d > 2$.
- $tw(d) = \left( \frac{1}{2} \right) \cdot sw(d) = \Theta(d^{1/2})$.
- $\left( \frac{tw(d)}{d^{d-1}} \right) \leq ctw(d) \leq tw(d - 1)$, so $ctw(d) = \Omega(d^{1/2})$.
- $ctw(d) \leq 2 \cdot sw(\lceil d/2 \rceil) \cdot tw(\lceil d/2 \rceil) = O(d^2)$.
- $\left( \frac{ctw(d)}{2} \right) \leq cbtw(d) \leq ctw(d)$, so $ctw(d) = \Theta(ctw(d))$.

Thus, the ring must index between $\Omega(d^{2/3} \log d)$ and $O(d^2)$ orders.

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. Table 3 shows the results, which verified our exact formulas for $w(d)$, $tw(d)$, and $cw(d)$. However, there is a gap between our lower bounds for $ctw(d)$, $cbw(d)$, and $cbtw(d)$, and the approximations we obtained for the larger values of $d$.

In summary, our support for cyclic bidirectional tuples slashes the number of required orders by an order of magnitude, enabling wco algorithms on dimensions that would be intractable with classical approaches (flat indexes), even using trie switching.

In comparison, Qdag [35] needs to index only one order, but query time is $O(Q^* \cdot 2^d m \log U)$. On the other hand, a cyclic unidirectional index would have query time complexity $O(Q^* \cdot dm \log U)$, by always going backwards, but it would need to index $ctw(d)$ orders, roughly twice as many as our ring index. We can implement such a cyclic unidirectional index with our BWT too, while extending the index of Brisaboa et al. [10] would yield $O(Q^* \cdot 2^d m \log U)$ time because it can move forwards, not backwards.

### Table 3: Number of orders that must be indexed to support wco algorithms depending on the index capabilities

<table>
<thead>
<tr>
<th>$d$</th>
<th>Flat W</th>
<th>Cyclic CW</th>
<th>Cyclic CTW</th>
<th>Ring CBW</th>
<th>Cyclic CBTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>24</td>
<td>8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>120</td>
<td>10</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>720</td>
<td>20</td>
<td>10,12</td>
<td>10,13</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>5040</td>
<td>35,50</td>
<td>[79,168]</td>
<td>18,25</td>
</tr>
</tbody>
</table>

### 7 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 index supports the worst-case-optimal Leapfrog Triejoin algorithm using almost no extra space on top of the raw triple data, and even in compressed space. Our ring index further offers fast on-the-fly statistics to help query optimisation. Our experiments show that the ring index uses a fraction of the space of traditional indexes while ranking amongst the best in terms of query response times as well.

Regarding future directions, per Section 6, it would be of interest to investigate ring indexes that support higher-arity relations for relational databases, property graphs, etc. An interesting trade-off to explore is maintaining high-arity relations, which necessitate fewer joins, and decomposing those relations into several lower-arity relations, which necessitate fewer index orders.

Currently our index is read-only, but it could support insertion and deletion of graph edges in $O(\log U \log n)$ time using a compressed representation of string collections based on dynamic wavelet trees [27]. This structure would still use $|G| + o(|G|)$ bits on a graph $G$ (and even compressed space); the price would be an additional penalty factor of $O(\log n)$ in query times. Alternatively, we can trade such a penalty factor for amortised update times by taking the union of results over a small dynamic text index where new triples are added, and a constant amount of increasing static rings for handling space overflows [32]. Various static rings can be merged periodically with the dynamic index to build a bigger ring, which can be done with efficient algorithms to merge BWTs [7].

Our focus has been on indexing for wco joins, where our query planning strategy is currently based on simple techniques. 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 [24], adaptive plans that use statistics collected during query evaluation [30], hybrid plans that combine wco and non-wco join algorithms for higher-arity relations [19, 30], among others. Our index may further support custom optimisations. For example, a useful statistic would be to find how many different elements are associated with a BWT range, which can be computed, at least in the backward direction, in logarithmic time, by roughly doubling the space [20]. The backward direction also enables faster intersection [21], which makes optimisations using a purely backwards (unidirectional) BWT-based index worth exploring. 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.

More generally, we believe that this paper opens up many interesting lines of research regarding time vs. space trade-offs in the context of worst-case optimal join algorithms.

Source code, scripts, queries, data, instructions for running experiments and omitted proofs are available online [5].

### Acknowledgments
We thank Amine Mhedhbi for help with Graphflow and the reviewers for their feedback. This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002.
References


