Worst-Case Optimal Graph Joins in Almost No Space: Additional Material

Anonymous Author(s)

ACM Reference Format:
Anonymous Author(s). 2021. Worst-Case Optimal Graph Joins in Almost No Space: Additional Material. In *SIGMOD '21: International Conference on Management of Data, June 20–25, 2021, Xi'an, Shaanxi, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

A Omitted theory

A.1 Proof of Lemma 3.3

Let us call LF and LF* the LF-functions associated with BWT and BWT*, respectively, as well as C and C^* their C arrays of Eq. (1). Since $C = C^*$ except that C has the extra entry $C[\$] = 3n$, we call them both C for simplicity. Figure 1 illustrates the proof.

Let q be such that $2n < q \leq 3n$. Then, $A[q] = 3(t-1)+3$ for some t, so $T[A[q]] = o_t$ and $\mathrm{BWT}[q] = p_t$. By Eq. (1), it holds

$$\mathrm{LF}^*(q) \;=\; C[p_t] + \mathsf{rank}_{p_t}(\mathrm{BWT}^*, q).$$

Since $\mathrm{BWT}^*[2n+1\,..\,3n] = \mathrm{BWT}[2n+1\,..\,3n]$ (the predicate zone), we have $\mathrm{LF}^*(q) = \mathrm{LF}(q) = q'$ for some $n < q' \leq 2n$ (in the subject zone). By the properties of LF, it holds that $A[q'] = 3(t-1)+2$, $T[A[q']] = p_t$, and $\mathrm{BWT}[q'] = s_t$. Again, by Eq. (1), it holds

$$\mathrm{LF}^*(q') \;=\; C[s_t] + \mathsf{rank}_{s_t}(\mathrm{BWT}^*, q').$$

Since $\mathrm{BWT}^*[n+1\,..\,2n] = \mathrm{BWT}[n+1\,..\,2n]$ (the subject zone), we have that $\mathrm{LF}^*(q') = \mathrm{LF}(q') = q''$ for some $0 < q'' \leq n$.

Now $A[q''] = 3(t-1)+1$, $T[A[q'']] = s_t$, and $\mathrm{BWT}[q''] = o$, where $o = o_{t-1}$ if $t > 1$ and $o = \$$ if $t = 1$. Moreover, by property (1), we have $q'' = t$. By Eq. (1), letting $c = \mathrm{BWT}^*[t]$, it holds that

$$\mathrm{LF}^*(t) \;=\; C[c] + \mathsf{rank}_c(\mathrm{BWT}^*, t).$$

Since $\mathrm{BWT}^*[1\,..\,n] = \mathrm{BWT}[2\,..\,n] \cdot o_n$, we have two cases:

(1) If $t < n$ then $\mathsf{rank}_c(\mathrm{BWT}^*, t) = \mathsf{rank}_c(\mathrm{BWT}, t+1)$ (since $\mathrm{BWT}[1] = \$ \neq c$). Furthermore, we have $A[t+1] = 3t+1$ by property (2). Thus $\mathrm{BWT}[t+1] = o_t = \mathrm{BWT}^*[t] = c$, and therefore $\mathrm{LF}^*(t) = \mathrm{LF}(t+1)$ because, by Eq. (1) in holds that

$$\mathrm{LF}(t+1) \;=\; C[o_t] + \mathsf{rank}_{o_t}(\mathrm{BWT}, t+1),$$

and $\mathrm{LF}(t+1)$ must be precisely q because $A[t+1] = 3t+1$ (property (2)) and thus $A[\mathrm{LF}(t+1)] = 3t = 3(t-1)+3 = A[q]$.

(2) If $t = n$, then $c = o_n$ and $\mathrm{LF}^*(t) = C[o_n] + \mathsf{rank}_{o_n}(\mathrm{BWT}^*, n) = C[o_n+1]$. This is the position of the lexicographically last suffix of T that starts with o_n. This position is $3n$, because $T[3n\,..\,3n+1] = o_n\,\$$, thus $\mathrm{LF}^*(t) = 3n$. But in this case it holds that $3n = q$, because $A[q] = 3(t-1)+3 = 3n$.

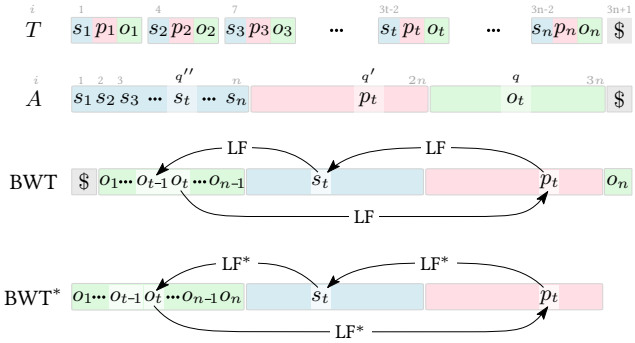

Figure 1: Illustration of the proof of Lemma 3.3

In both cases, we have that $\mathrm{LF}^*(t) = q$ and have thus completed the cycle, $\mathrm{LF}^*(\mathrm{LF}^*(\mathrm{LF}^*(q))) = q$, as claimed: $q'' = t = \mathrm{LF}^*(q')$; $q = \mathrm{LF}^*(q'') = \mathrm{LF}^*(t)$ and $T[A[q]] = o_t$; $q' = \mathrm{LF}^*(q) = \mathrm{LF}^*(\mathrm{LF}^*(t))$ and $T[A[q']] = p_t$; $\mathrm{LF}^*(\mathrm{LF}^*(\mathrm{LF}^*(q''))) = q''$.

A.2 Proof of Lemma 3.6

The case for $b = 0$ constants is trivial as $[s\,..\,e] = [1\,..\,n]$.

Next, if $b = 1$, and d is the lone constant in t, then $[s\,..\,e] = [C[d]+1\,..\,C[d+1]]$; note $T[A[k]] = d$ for all $s \leq k \leq e$.

When $b = 2$, let $[s\,..\,e]$ be the result of the backward search for the string P that concatenates the two constants (i.e., $P = \alpha\beta$, or $P = \beta\gamma$, or $P = \gamma\alpha$). The search takes $O(\log U)$ time since it involves two applications of Eq. (2).

Finally, for $b = 3$ we obtain the range $[s\,..\,e]$ by backward searching for $P = \alpha\beta\gamma$, in $O(\log U)$ time as well.

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

A.3 Rings in compressed space

We show that the space of the ring index can be made close to the size of a compressed representation of the text.

Assume we divide $\mathrm{BWT} = \mathrm{BWT}_o \cdot \mathrm{BWT}_s \cdot \mathrm{BWT}_p$ as in Section 4.1. By representing BWT_o, BWT_s, and BWT_p separately, and using compressed bitvectors [11] in the wavelet matrices, the main component of the space, $3n \log_2 U$ bits, is reduced to the empirical zero-order entropy of the strings [3, 8],

$$n(\mathcal{H}_0(\mathrm{BWT}_o) + \mathcal{H}_0(\mathrm{BWT}_s) + \mathcal{H}_0(\mathrm{BWT}_p)),$$

where, if symbol c appears n_c times in S over alphabet $[1, \tau]$, then

$$\mathcal{H}_0(S) \;=\; \sum_{c \in [1,\tau]} \frac{n_c}{n} \log_2 \frac{n}{n_c} \;\leq\; \log_2 \tau;$$

all our sums range over the symbols c where $n_c > 0$.

SIGMOD '21, June 20–25, 2021, Xi'an, Shaanxi, China

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

Further, when S is the BWT transform of T, the zero-order entropies become even smaller high-order entropies of T [8]. The reason is that the space is not only $|S|\mathcal{H}_0(S)$, but also $\sum_{i=1}^{k} |S_i|\mathcal{H}(S_i)$ for any partition $S = S_1 \cdots S_k$. In our case, for example, BWT_o can be partitioned into the subranges where the o_i symbols are followed by each specific substring sp, and therefore, extending our notation n_c to n_x for cyclic substrings x of T, we require space

$$\sum_{sp \in U \times U} n_{sp} \sum_{o \in U} \frac{n_{spo}}{n_{sp}} \log_2 \frac{n_{sp}}{n_{spo}} = \sum_{sp \in U \times U} n_{sp} \log_2 n_{sp},$$

the equality holding because the triples are unique and then $n_{spo} = 1$. The cases of BWT_s and BWT_o are analogous. The space for the whole BWT is then

$$\sum_{sp \in U \times U} n_{sp} \log_2 n_{sp} + \sum_{po \in U \times U} n_{po} \log_2 n_{po} + \sum_{os \in U \times U} n_{os} \log_2 n_{os}$$

bits. To this we must add $o(n \log U) + O(U^3 \log n)$ bits of encoding overhead, though the latter is a very pessimistic upper bound [8] (as our C-Ring space shows). This is essentially the same space an order-2 statistical compressor on T can achieve if it encodes each symbol using the other two symbols of its triple as the context.

B Variables appearing more than once in a triple pattern

When a variable x has more than one occurrence in a triple pattern t, these are consecutive (in the cyclic interpretation). Thus, to support `leap(`t, x, c`)` on our unmodified index, we can process t backwards. First, we consider only the second occurrence (which is just behind the matched part of the triple pattern). Every time we find a binding $x := c_x$, we perform a second backward search step 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. However, 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 split T into five texts of total length $3n$ (plus the terminators \$): T_{xyz} contains the triples (s, p, o) where s, p, and o are all different, T_{xxy}, T_{xyx}, and T_{yxx} contain the triples where $s = p$, $s = o$, and $p = o$, respectively, and the other component is different, and T_{xxx} contains the triples where $s = p = o$. We then create five bended BWTs, one per text.

At query time, the occurrences of each partially bound triple pattern t_i' span five intervals $A[s_i \,..\, e_i]$, one per BWT. However, triple patterns t_i containing two copies of the same variable have two intervals only: one in the BWT of T_{xxx} and the other in the BWT of T_{xxy}, T_{xyx}, or T_{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 BWT of T_{xxx} only. This arrangement ensures that the aforementioned algorithm handling multiple occurrences of a variable, every match $x := c_x$ we find is valid.

Triples that span several intervals $A[s_i \,..\, e_i]$ 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 asymptotically the same, even in compressed space. On graphs where the edge labels are disjoint from the node labels, we need only consider T_{xyz} and T_{xyx}, and if they have no self-loops then no triple-pattern with a variable appearing twice can match.

C 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 to facilitate reproducibility of our experiments [1].

C.1 The Wikidata graphs

We take the same Wikidata graphs as proposed for the Wikidata Graph Pattern Benchmark (WGPB) [7], 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 [7], triples whose predicates appear in fewer than 1,000 triples or more than 1,000,000 triples were also removed.

C.2 System details

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

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. 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.

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/.

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.

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 the webpage for further details on how we configured and ran these systems [1].

```
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
```

Figure 2: A difficult real-world query in SPARQL syntax

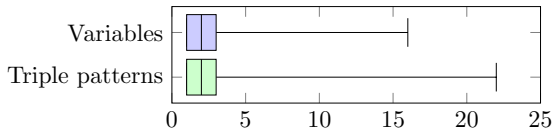

Figure 3: Box-plots of number of variables and triple patterns in real-world queries

C.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. [7] 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 [9] 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 [12]. We project all variables and limit the results to 1000 (per WGPB). This process yielded 1,315 queries for testing. We provide an example of one of the more challenging cases in Figure 2, which looks for information about people who died on the same date. The average number of triple patterns and variables per query was 2.43 and 2.55, respectively; in Figure 3 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 22 triple patterns and 16 variables. In Table 1 we show the most common types of triple patterns; we find all possible types (aside from all constants), where 432 triple patterns (9.0%) have variable predicates.

C.4 String identifiers

As aforementioned, we work directly on datasets and queries where the identifiers are already mapped to integers. This can be seen as unfair with the database systems managing strings as reporting strings may induce additional time overhead and their space can be significant compared to the integer triples. In our dataset, for example, the strings occupy 1,427 MB, nearly 150% of the 932 MB used by the integer triples in plain form. Blazegraph, Jena, Jena-LTJ, RDF-3X, and Virtuoso include this overhead, while Ring, C-Ring, EmptyHeaded and Qdag do not.

Table 1: Most common types of triple patterns where ? indicates a variable and s, p, o indicate constants

Type	Count
?p?	2,460
?po	1,829
???	318
s??	58
sp?	55
??o	54
s?o	2
Total	4,776

We can largely diminish the impact of the strings by storing them in succinct dictionaries [10], which map from strings to integers (to translate the queries) and back (to translate the solutions) in a few microseconds per string. For example, using the variant `HTFC-rp` with sampling 64 [10], the strings are compressed to 17%, or 235 MB (3 additional bytes per triple) and an identifier is translated to its string in about 3 microseconds. The total impact of translating the identifiers back to strings adds just 0.35 milliseconds to the times in Table 1, and at most 0.003 seconds in Figure 8.

D Indexing higher dimensions

The ring index can be extended to handle relations of higher arity. Relations of dimension $d = 4$ are called *quads*, which are commonly used to store graph databases [4–6], and higher dimensions arise in relational databases. We first describe how our machinery can be extended to deal with arbitrary dimensions, assuming we always have a ring where the bounded parts of the tuple patterns are contiguous. We then analyse how many rings we must build depending on d. It turns out that the space cost incurred by ring indexes is an order of magnitude lower than with classical indexes.

D.1 LTJ in higher dimensions (Theorem 6.1)

The Leapfrog TrieJoin algorithm, its analysis, and our implementation in Section 3, can be extended with essentially no changes from triples ($d = 3$) to tuples of any dimension d. At query time, if a contiguous part of the tuple pattern t'_i is partially bound, we know the range $A[s_i \,..\, e_i]$ corresponding to its bound component preceded by an unbound one. Bounding an additional tuple element backwards is then done as described for triples.

Bounding an element in forward direction, however, is different than as described in Section 3, because the position to bound may not immediately follow the position corresponding to the range $A[s_i \,..\, e_i]$. Let $B = \beta_1 \cdots \beta_k$ be the sequence of bounded constants in t'_i. To implement $\texttt{leap}(t'_i x, c)$ in forward direction, we backward search for B starting from the interval $\text{BWT}[C[c]+1\,..]$. If there are no results, then we return $\perp$. Otherwise, let $[s'_i \,..\, e'_i]$ the resulting interval (a subrange of $[s_i \,..\, e_i]$; in fact $e'_i = e_i$ and we only need s'_i). Then, $T[A[s'_i]\,..]$ is the lexicographically first suffix starting with

$B \cdot c_x$ for some $c_x \geq c$. To find out c_x, we start from $q_0 = s'_i$ and do $q_i := \texttt{select}_{\beta_i}(\text{BWT}, q_{i-1} - C[\beta_i])$ for $i := 1$ to k. The value q_k then corresponds to q in Section 3.2.2, from which we compute c_x in the same way. The total cost of this process is $O(k \log U) = O(d \log U)$.

Since there are md positions in the m tuple patterns, and each might have to be solved in time $O(d \log U)$, the time that multiplies AGM is now $O(d^2 m \log U)$. On the other hand, if we extend the idea of Section B we incur an additional $O(B_d)$ penalty factor, where B_d indicates the d^{th} Bell number, so we prefer not to relax wco guarantees for multiple occurrences of a variable in the same tuple pattern. We then obtain Theorem 6.1.

D.2 On the number of orders to index

A key aspect of our ring index for triples is that, because it can extend the range of bounded variables forwards and backwards, and because the triples are cyclic, it suffices to index a single order instead of the 6 orders needed in classical wco indexes. But how many indexes are needed to support leapfrog in relations of higher dimensions? 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 then turn to cyclic indexes, starting with the unidirectional version of Brisaboa et al. [2], and continuing with our own ring index.

D.2.1 Flat indexes and trie switching, variants W and TW. A classical (flat) index on d columns needs to store, in principle, all the $w(d) = d!$ possible orders to support 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 adding inter-trie pointers that allow us to reorder the variables we have already bound. For example, when indexing quads (s, p, o, g), this avoids storing a trie with the order *gspo* if we have tries for *gsop* and *sgpo* with pointers between the corresponding nodes of *gs* and *sg*: we process *gs* with the first index and then use the pointer to *sg* in the second index so as to continue with *po*.

We call TW the class of indexes supporting 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 constants defined and need to intersect using any of the $(d-l)$ remaining variables. Each such arrangement requires storing a different order. This formula is maximised for $l = \lfloor d/2 \rfloor$, yielding a lower bound of $tw(d) = \lceil \frac{d}{2} \rceil \binom{d}{\lfloor d/2 \rfloor}$ orders. We now prove that $tw(d)$ is also an upper bound by building a sufficient set of $tw(d)$ orders.

Lemma D.1. *Let a set of strings $S[1\,..\,l+1]$ be (l,d)-complete if, for every $0 \leq m < l$, 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. Then there exists a $(d-1,d)$-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 just 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.

Note that, as soon as as $d-l-1 \leq l+1$, that is, $l+1 \geq d/2$, we will need to create no new strings, because there will be sufficiently many ones 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. □

The trie-switching technique can be used with our indexes as well, without storing any extra space: if we have found a range for some contiguous bounded variables in one index, we can search another index for the same variable values in another desired order using backward search, as long as they are contiguous too. Each such change of index costs $O(d \log U)$ time, which is within the time complexity given in Theorem 6.1. Thus, in the following discussion we analyse both versions of each new index scheme: with and without trie-switching.

D.2.2 Cyclic indexes, variants CW and CTW. On indexes supporting cyclic tuples but not trie switching (which we call CW indexes), exactly $cw(d) = (d-1)!$ indexes 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 they are the same when regarded as cyclic, 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. Equivalently we note that $(d-1)!$ is the number of Hamiltonian cycles in the complete directed graph of d nodes, where each directed cycle corresponds to a CW index, and we need $(d-1)!$ indexes to traverse the d nodes in any order.

As explained, we can enable trie switching on cyclic indexes, leading to what we call CTW indexes. Seen as a lower bound for trie switching, $tw(d)$ is the number of prefixes of length $\lfloor d/2 \rfloor + 1$ that cover every possible subset of $\lfloor d/2 \rfloor$ positions followed by any other position. Since each starting point in the cycle yields a sequence of $\lfloor d/2 \rfloor + 1$ elements, we need $ctw(d) \geq \lceil tw(d)/d \rceil$ orders. 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 o.

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

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

Proof. Consider a $(d-2,d-1)$-complete set of strings TW (Lemma D.1) forming a TW index for the positions $[1\,..\,d-1]$. Let $\text{dom}(S)$ be the set of symbols in string S, and let TW_S be the set of strings of TW prefixed with any S' such that $\text{dom}(S') = \text{dom}(S)$.

We build a CTW index for the positions $[1\,..\,d]$ by simply appending d to each of the strings in TW. To see that this index

is valid, consider an 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 $d \le 13$, is asymptotically stronger.

Lemma D.3. *Let a set of strings $S[1\,..\,l]$ be (l,d)-sufficient if, for every $0 \le m \le l$, 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. Then there exists a (d,d)-sufficient set of $sw(d) = \binom{d}{\lfloor d/2 \rfloor}$ strings.*

Proof. We proceed by induction on l, building for every $0 \le l \le d$ an (l,d)-sufficient set. For $l \le d/2$, this set will have $\binom{d}{l} \le sw(d)$ 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 extending some string of the (l,d)-sufficient set and adding a new initial 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 extend it into all the possible strings of an $(l+1,d)$-sufficient set. □

Lemma D.4. *It holds that $ctw(d) \le 2 \cdot sw(\lfloor d/2 \rfloor) \cdot tw(\lceil d/2 \rceil)$.*

Proof. Let $c = \lfloor d/2 \rfloor$, SW_s (resp. TW_s) be a (c,c)-sufficient (resp. $(d-c-1, d-c)$-complete) set of strings, and SW_l (resp. TW_l) be a (c,c)-sufficient (resp. $(d-c-1, d-c)$-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 $2 \cdot sw(c) \cdot tw(d-c)$.

To see this is a valid index, consider any instantiation order $X[1\,..\,d]$ (i.e., a permutation in $[d]$) 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,c)-sufficient and SW_l being analogously obtained from a (c,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 $\mathrm{dom}(S_l[c - |X_l(p)| + 1\,..\,c]) = \mathrm{dom}(X_l(p))$.

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 $(d-c-1, d-c)$-complete, there is a string $S \in TW_s$ such that $\mathrm{dom}(S[1\,..\,|X_s(p)|]) = \mathrm{dom}(X_s(p))$ and $S[p+1] = X[p+1]$. Further, 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 $(d-c-1, d-c)$-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. □

Note that, 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 $cwt(d) \le 2^d/\pi\,(1 + O(d^{-1/2})) = O(2^d)$.

D.2.3 Bidirectional indexes, variants CBW and CBTW. Consider a cyclic index with no trie switching, which we call CBW. In the unidirectional case (CW), recall that $cw(d) = (d-1)!$. In the bidirectional case, we can remove CW indexes that are duplicated but "reversed" (e.g., *spog*, *gops*), where each index and its reverse are in CW; put another way, recalling that $cw(d)$ is the number of Hamiltonian cycles in the complete directed graph, then $ctw(d)$ is the number of Hamiltonian cycles in the complete undirected graph, which is $(d-1)!/2$, or $cw(d)/2$. We thus have the upper bound $cbw(d) \le \lceil cw(d)/2 \rceil$, that is, 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, so a lower bound is $cbw(d) \ge \lceil (d-1)!/2^{d-2} \rceil$ orders.

If we add bidirectionality to a cyclic index that supports trie switching (we call CBTW the result), the order in which we extend the initial position does not matter, only the number of different cycle segments of length $\lfloor d/2 \rfloor$ we can produce, which is the same d as without bidirectionality. We can still, however, have two different intersection variables, at the two extremes of the range of $\lfloor d/2 \rfloor$ positions. For example, with *spog* we obtain $\{s,p\}$ 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. This yields a lower bound of $cbtw(d) \ge \lceil tw(d)/(2d) \rceil$ orders. More generally, it holds that $cbtw(d) \ge \lceil ctw(d)/2 \rceil$, because any CBTW index storing r orders can be converted into a CTW index storing $2r$ orders, by storing each order and its reverse. With trie switching 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.

The upper bounds for $ctw(d)$ imply that $cbtw(d) = O(2^d)$. By our lower bounds, both are also $\Omega(2^d d^{-1/2})$. Both are then asymptotically smaller than $tw(d) = \Theta(2^d d^{1/2})$. This concludes the proof of Theorem 6.2.

D.2.4 Finding the right index. With nearly 2^d orders indexed, how to find the proper index 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 $i \ne j$ (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 $cbtw(d) = \Omega(2^d d^{-1/2})$) and is built in time $O(d^2)$ per order indexed, that is, at most $O(d^2 2^d)$.

References

[1] D. Arroyuelo, A. Hogan, G. Navarro, J.L. Reutter, J. Rojas-Ledesma, and A. Soto. 2020. Online appendix and material. https://github.com/darroyue/Ring.

[2] N. Brisaboa, A. Cerdeira, A. Fariña, and G. Navarro. 2015. A compact RDF store using suffix arrays. In *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*. 103–115.

[3] F. Claude, G. Navarro, and A. Ordóñez. 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems* 47 (2015), 15–32.

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

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

[6] A. Harth and S. Decker. 2005. Optimized index structures for querying RDF from the web. In *Proc. Latin American Web Congress (LA-Web)*. 71–80.

[7] A. Hogan, C. Riveros, C. Rojas, and A. Soto. 2019. A worst-case optimal join algorithm for SPARQL. In *Proc. International Semantic Web Conference (ISWC)*. 258–275.

[8] V. Mäkinen and G. Navarro. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms* 4, 3, Article 32 (2008).

[9] 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. International Semantic Web Conference (ISWC)*. 376–394.

[10] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. 2016. Practical compressed string dictionaries. *Information Systems* 56 (2016), 73–108.

[11] 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).

[12] J. Salas and A. Hogan. 2018. Canonicalisation of Monotone SPARQL Queries. In *Proc. International Semantic Web Conference (ISWC)*. 600–616.