BWT Indexes for Optimal Joins in Graph Databases

Diego Arroyuelo
Department of Computer Science, Escuela de Ingeniería, Pontificia Universidad Católica de Chile, Chile
Millennium Institute for Foundational Research on Data (IMFD), Chile

Gonzalo Navarro
Department of Computer Science, University of Chile, Chile
Millennium Institute for Foundational Research on Data (IMFD), Chile

Abstract

Graph databases represent data as a labeled directed graph, where the labels refer to properties that connect the entities represented by their source and target vertices. Queries feature, most prominently, sets of edges where source, target, and/or label can be variables; each instantiation of the variables where all the edges occur in the graph is a solution to the query. Worst-case-optimal algorithms to solve those queries have been devised, but they pose significant space requirements. This overhead has hindered the adoption of worst-case-optimal algorithms in real systems. We show that a representation of the graph based on the extended BWT (eBWT), where each edge is seen as an independent string of length 3 (source, label, target) supports worst-case-optimal algorithms while using almost no extra space on top of the raw data. We then show how the idea is generalized to the relational model, where the strings can be longer than 3 and several eBWTs are needed to obtain worst-case optimality. The aim to minimize the amount of space in that case leads to consider novel eBWT variants, where columns other than the last can be chosen. Finally, we show how the same graph representation can be used to solve other typical queries, like finding graph paths that match regular expressions.

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management; Theory of computation → Database query processing and optimization (theory); Theory of computation → Data compression

Keywords and phrases Graph databases, Ring index, extended BWT, compact data structures

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.14

Funding ANID, Chile, via Millennium Science Initiative Program – Code ICN17_002
Gonzalo Navarro: Fondecyt Grant 1-230755

1 Introduction

Graph databases (GDBs) have become increasingly popular [29, 24, 26], as they enable the storage of unstructured information repositories that emphasize the relations between entities in applications such as knowledge graphs, the semantic web, social networks, transaction networks, and communication networks, among others. A number of GDB management systems and prototypes have been introduced [15, 33, 34, 1, 2, 25, 30], along with models and query languages [3, 19, 23]. The efficient implementation of GDBs, however, still faces various challenges [8], such as the efficient implementation of the key algorithms to solve the most relevant queries on GDBs. To achieve the needed efficiency, GDBs require indexes that tend to demand a lot of memory. For example, a common type of query on GDBs are basic graph patterns, which search for subgraphs that are homomorphic to a given query graph. These types of queries can be translated into the relational join of many tables, which is known to be computationally demanding. Research on efficient algorithms to compute those joins has led to the concept of worst-case optimal (wco) join algorithms [10], which offer certain guarantees of running time in a worst-case scenario. These wco algorithms, however, generally demand strong indexing schemes, which entails a significant use of memory. Leapfrog Triejoin (LTJ) [35] is one of the most popular wco join algorithms, and has proven to be efficient in practice [35, 25]. However, its high memory consumption makes it less viable in situations of limited memory or when the graphs are excessively large. This high memory usage comes from the need to store indexes for 6 different "permutations" of the graph database (a concept that will be clear soon) in order to work properly.

With other coauthors, we introduced the Ring [7, 5] as an alternative to reduce the excessive memory usage of LTJ. The Ring is able simulate the 6 permutations needed by LTJ (and the corresponding indexes) by storing only one of them. This allows implementing LTJ using little additional space on top of that used by the database itself, thereby opening the use of wco algorithms to many more applications. We originally conceived the Ring in terms of the BWT [13]. We presented it to a database audience aiming to simplify the stringology aspects as much as possible [7]. Our presentation was closer to that of the Permuterm index [18], which regards the strings as circular, and simplified it to the case of strings of length 3. In the journal version [5] we managed to get rid completely of the stringology concepts, which are difficult to grasp in other areas, and presented the Ring in terms of stable sortings.

In this paper we present the Ring to a stringology audience, and thus properly formalize its concepts in terms of the more elegant extended BWT (eBWT) [28]. The generalization of the ring to relational tables leads to the definition of eBWT variants that are new, to the best of our knowledge.

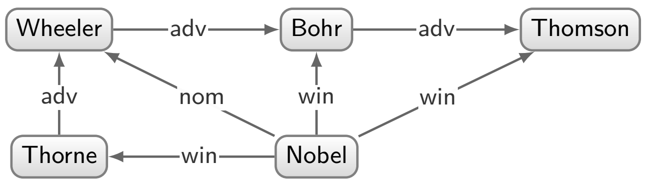
2 Basic Concepts of Graph Databases

2.1 Graph databases

A graph database is an edge-labeled graph modeled as a finite set of triples $G \subseteq \mathcal{U}^3$ where, for simplicity, $\mathcal{U}$ will represent a finite and totally ordered set of constants. We will stick to the Resource Description Framework (RDF) model [27], as it is simpler than others and sufficient to present the important concepts. In RDF, each triple $(s, p, o) \in G$ represents the directed edge $s \xrightarrow{p} o$, connecting vertex s to vertex o, with p being the edge label. The terms *subject*, *predicate*, and *object* are used to refer to s, p, and o, respectively. The number of edges (triples) in G is denoted by $N = |G|$. The alphabet for the vertices of graph G is defined as $\Sigma_V = \{s, o \mid (s, p, o) \in G\}$, while $\Sigma_L = \{p \mid (s, p, o) \in G\}$ represents the alphabet for the edge labels. We assume $\Sigma_V \cap \Sigma_L = \emptyset$, implying that no edge label p is used as a vertex in G. The domain of graph G is given by $\mathrm{dom}(G) = \{s, p, o \mid (s, p, o) \in G\} \subseteq \mathcal{U}$, so we can safely assume $|\mathcal{U}| \leq 3N$. Figure 1 illustrates a sample graph, which will be used as our running example. The figure also presents a particular enumeration of the constants in $\mathrm{dom}(G)$, as well as the corresponding set of graph triples.

2.1.1 Basic Graph Patterns

Let $\mathcal{V}$ denote an infinite set of variables such that $\mathcal{U} \cap \mathcal{V} = \emptyset$. The simplest way of querying an edge-labeled graph is through *triple patterns*, which are tuples $(s, p, o) \in (\mathcal{U} \cup \mathcal{V})^3$. The solutions to a triple-pattern query involve all potential assignments of the variables within the pattern to constants in $\mathrm{dom}(G)$ such that the resulting triple exists in G. This allows for querying all graph edges (using variables s, p, and o), all edges with a vertex $s \in \mathrm{dom}(G)$ as the subject (where s is a constant and p and o remain variables), all edges with label p (p is



Mapping:

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

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

Figure 1 An example graph database (above), a possible enumeration of its constants (below, left), and the corresponding triple representation (below, right).

constant while s and o are variables), and all objects related to subject s by label p (s and p are constant while o is a variable), among other queries. When viewing G as a ternary relation in the relational model, triple patterns are analogous to equality-based selections.

Subsequently, triple patterns can be used to formulate more complex queries, such as *basic graph patterns* (BGPs). A BGP represents a finite set $Q \subseteq (\mathcal{U} \cup \mathcal{V})^3$ of triple patterns. Let $\mathbf{V}(Q)$ be the set of variables in a BGP Q. The *evaluation* of Q upon a graph G is defined as set of mappings $Q(G) = \{\mu : \mathbf{V}(Q) \to \text{dom}(G) \mid \mu(Q) \subseteq G\}$, referred to as *solutions*, where $\mu(G)$ denotes the result of substituting each variable $x \in \mathbf{V}(Q)$ in Q with $\mu(x)$. Some examples of BGPs on the graph of Figure 1 are as follows:

- $Q = \{(\mathsf{Nobel}, \mathsf{win}, x)\}$, for $x \in \mathbf{V}(Q)$, which looks for all Nobel Prize winners. In our example graph, $Q(G)$ contains the instantiations $\mu_1(x) = \mathsf{Thorne}$, $\mu_2(x) = \mathsf{Bohr}$, and $\mu_3(x) = \mathsf{Thomson}$.
- $Q = \{(\mathsf{Nobel}, \mathsf{win}, x), (\mathsf{Nobel}, \mathsf{win}, y), (x, \mathsf{adv}, y)\}$, for $x, y \in \mathbf{V}(Q)$, which looks for all pairs of values x and y such that x advised y and both x and y won the Nobel Prize. In our example graph, $Q(G)$ contains a single instantiation, namely $\mu(x) = \mathsf{Bohr}$ and $\mu(y) = \mathsf{Thomson}$.

2.1.2 Regular Path Queries

Apart from BGPs, the second major component of most graph query languages are the so-called *Regular Path Queries (RPQs)*. An RPQ essentially specifies a regular expression on the alphabet Σ_L, and looks for the paths in the graph where the concatenation of the edge labels belongs to the language denoted by the regular expression. An RPQ may also fix the initial and/or the final nodes of the desired paths. The answer to the query are the extremes of all matching paths, in the form of pairs of nodes (s, o). Additionally, the regular expression may use arrows in reverse order, by specifying a "reversed" version $\hat{p}$ of the symbols $p \in \Sigma_L$. This can be handled by adding an edge $o \xrightarrow{\hat{p}} s$ per graph edge $s \xrightarrow{p} o$, so we omit this feature in the sequel.

For instance, the RPQ $\mathsf{Thorne}\ \mathsf{adv}^+\ x$, where x is a variable, looks for all academic descendants of Thorne in the graph of Figure 1 (the regular expression adv^+ denotes all the nonempty repetitions of the symbol adv). Its solutions are $\mu(x) = \mathsf{Wheeler}$ (because of the path $\mathsf{Thorne} \xrightarrow{\mathsf{adv}} \mathsf{Wheeler}$), $\mu(x) = \mathsf{Bohr}$ (because of $\mathsf{Thorne} \xrightarrow{\mathsf{adv}} \mathsf{Wheeler} \xrightarrow{\mathsf{adv}} \mathsf{Bohr}$), and $\mu(x) = \mathsf{Thompson}$ (because of $\mathsf{Thorne} \xrightarrow{\mathsf{adv}} \mathsf{Wheeler} \xrightarrow{\mathsf{adv}} \mathsf{Bohr} \xrightarrow{\mathsf{adv}} \mathsf{Thompson}$). Similarly, $x\ \mathsf{adv}^+\ y$, for variables x and y, looks for all instantiations of these variables such that y is an academic descendant of x.

2.2 Worst-Case-Optimal joins

Joins are some of the most resource-intensive operations in relational algebra, making their efficient computation essential for the performance of database systems. A sloppy implementation might end up computing the whole cross product of the involved relations, wasting resources. For a join query Q and a database instance D, a join algorithm enumerates $Q(D)$, the solutions for Q over D. In order to formalize the study of efficient join algorithms, Atserias et al. [10] introduced the concept of the *AGM bound* (named after its authors). Consider a natural join query Q on a database instance D, which comprises a collection of relations. A natural join can be seen as the subset of the Cartesian product where the values in the common attributes coincide (and only one column per common attribute is preserved). The AGM bound of Q over D, denoted Q^*, is the highest possible number of tuples that can result from evaluating Q on any database instance D' that includes, for each relation R in D, a relation R' with the same attributes as R and such that $|R'| \leq |R|$.

A join algorithm is termed *worst-case optimal* (wco) if its running time is bounded by $O(Q^*)$, possibly multiplied by polylogs on N (the number of tuples in D) and factors independent of N (such as $|Q|$ or the number of attributes in D). Although originally conceptualized for relational databases, this idea has also been extended to include graph databases. Even though BGPs are more general than natural joins, it is still possible to apply the AGM bound by considering each triple pattern in a BGP as a relation comprising the triples that match its constants [25].

2.3 Leapfrog TrieJoin

Next, we outline the Leapfrog Triejoin algorithm [35] (LTJ, for short), which is arguably the most commonly used wco join algorithm in practice. In order to work properly, LTJ requires the graph database to be structured using tries. The idea is to represent each triple (graph edge) storing its components in a trie following a particular order (e.g., in *spo* order). Indeed, to achieve worst-case optimality, all $3! = 6$ possible permutations of the tuples need to be maintained, resulting in the storage of 6 tries (the rationale for this requirement will be explained subsequently). We will call T_{spo}, T_{sop}, T_{pso}, T_{pos}, T_{osp}, and T_{ops} these tries. Figure 2 shows the 6 tries corresponding to our running-example graph of Figure 1.

Let $Q = \{t_1, \ldots, t_q\}$ be a BGP, where t_i is a triple pattern, for $1 \leq i \leq q$. Let $\mathbf{V}(Q) = \{x_1, \ldots, x_v\}$ be the set of variables of Q. The distinctive strategy of LTJ is known as *variable elimination*. This method involves $v = |\mathbf{V}(Q)|$ iterations, each focusing on a single variable from $\mathbf{V}(Q)$ to be "eliminated" from the join process. The particular order in which variables are eliminated determines a total order $\langle x_{i_1}, \ldots, x_{i_v} \rangle$ of $\mathbf{V}(Q)$, which is called a Variable Elimination Order (VEO, for short). Throughout this process, each triple pattern t_i is treated as a ternary relation that participates in the join. Once a VEO is established for Q, each triple pattern t_i will have a particular trie T_i associated with it. The trie T_i should first traverse the constants in t_i, and then traverse the variables in t_i in an order that is consistent with that of the VEO. This is why LTJ needs to store 6 tries. To illustrate this, consider the query $Q = \{(x, \mathsf{adv}, y), (z, \mathsf{nom}, x), (z, w, y)\}$, for $x, y, z, w \in \mathbf{V}(Q)$. If the VEO is $\langle x, y, z, w \rangle$, then the first triple is associated with trie T_{pso}, the second with T_{pos}, and the

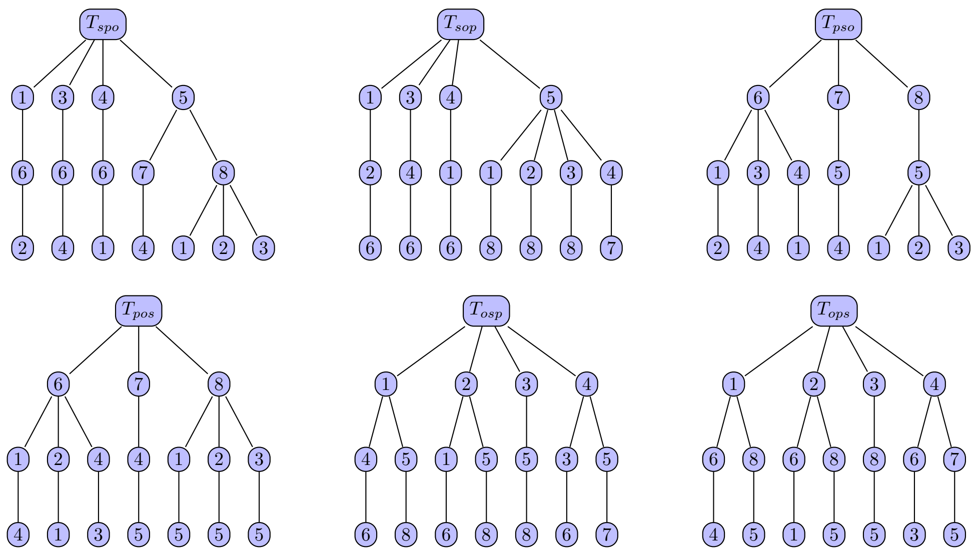


Figure 2 The 6 tries corresponding to our running-example graph.

third one with T_{osp}. If, instead, the VEO is $\langle z, y, x, w\rangle$, then the tries are T_{pos}, T_{pso}, T_{sop}. If some of these 6 tries were not stored, then some VEOs would become unmanageable.

The navigational operations that must be supported at any trie node v of a trie T are:

- $T.\mathsf{root}()$: moves to the root node of trie T.
- $T.\mathsf{child}(v, c)$: descends to the child of node v labeled c in trie T.
- $T.\mathsf{leap}(v, c_0)$: yields the smallest symbol $c \geq c_0$ that labels a child of node v in trie T.

LTJ starts at the root of each T_i and descends by the children that correspond to the constants in triple t_i. Next, we proceed to the variable elimination phase, assuming a VEO $\langle x_{i_1}, \ldots, x_{i_v}\rangle$. For $j = 1, \ldots, v$, let $Q_j \subseteq Q$ be the set of triple patterns that contain variable x_{i_j}. Starting with the initial variable x_{i_1} in the VEO, algorithm LTJ first identifies all values $c \in \mathrm{dom}(G)$ such that for every $t \in Q_1$, substituting x_{i_1} with c in t results in a non-empty evaluation of the modified triple pattern t over G (indicating that there are potential solutions to Q where x_{i_1} equals c). If the trie T related to t is consistent with the VEO, then its current node's children precisely represent the suitable values c for x_{i_1}.

Throughout the execution, we maintain a mapping μ with the solutions of Q. When a suitable value c is found for x_{i_1}, we bind x_{i_1} to c, resulting in $\mu = \{(x_{i_1} := c)\}$, and branch on this particular value c. In this branch, we go down by c in all the virtual tries T such that $t \in Q_1$. Next, the same process is applied to Q_2, determining suitable values d for x_{i_2}, and extending the mapping to $\mu = \{(x_{i_1} := c), (x_{i_2} := d)\}$. This process is repeated for the remaining variables in the VEO. Once all variables have been bound in this way, μ effectively contains a solution for Q; this occurs multiple times due to the branching for each binding of x_{i_1}, x_{i_2}, and so forth. When all bindings c for a variable x_{i_j} have been considered, LTJ backtracks and moves on to the next binding for Q_{j-1}. Upon completion of this process, the algorithm has reported all possible solutions for Q.

Determining the values c, d, and so forth, involves finding the intersection among the child nodes of the current nodes across all tries T_i, for each $t_i \in Q_j$. The algorithm LTJ carries out this intersection using the primitive $T_i.\mathsf{leap}(v, c_0)$ to implement Barbay and Kenyon's intersection algorithm [11]. Veldhuizen [35] proved that if the leap operation runs within polylogarithmic time, then LTJ is wco regardless of the VEO chosen, provided the tries have an appropriate attribute order.

We next illustrate how LTJ handles the query $Q = \{(x, \mathsf{adv}, y), (z, \mathsf{nom}, x), (z, w, y)\}$ using the VEO $\langle z, y, x, w\rangle$. The triple patterns in Q are associated with tries T_{pos}, T_{pso}, and T_{sop}, respectively. We set $\mu = \emptyset$ and begin by processing the constants adv and nom, identified as 6 and 7 in our example enumeration, descending to the respective nodes in T_{pos} and T_{pso}. We then handle the variables following the VEO. Starting with z in the second and third triple patterns of Q, we intersect the children at the current nodes of both tries (the node corresponding to string 7 in T_{pso} and the root of trie T_{sop}). The only common child is 5, so we descend to those nodes in both tries, setting $\mu = \{(z := 5)\}$. We now proceed to y, in the first and third triple patterns. We intersect the children of nodes for string 6 in T_{pos} and 5 in T_{sop}. The first intersection element is 1, leading us to descend in both tries and to update $\mu = \{(z := 5), (y := 1)\}$, and to proceed to x in the first and second triple patterns. We intersect the children of nodes corresponding to string 61 in T_{pos} and 75 in T_{pso}. Since 4 is a common value, we descend to the appropriate nodes and update $\mu = \{(z := 5), (y := 1), (x := 4)\}$. Lastly, for w, present only in the third triple pattern, no intersection is required: all children of the node for string 51 in T_{sop} are valid for w. The only value is 8, so we set $\mu = \{(z := 5), (y := 1), (x := 4), (w := 8)\}$. Having bound all variables, μ is a solution to Q. According to our enumeration, these values translate to $z :=$ Nobel, $y :=$ Bohr, $x :=$ Wheeler, $w :=$ win, resulting in the set of triples $\{(\mathsf{Wheeler}, \mathsf{adv}, \mathsf{Bohr}), (\mathsf{Nobel}, \mathsf{nom}, \mathsf{Wheeler}), (\mathsf{Nobel}, \mathsf{win}, \mathsf{Bohr})\}$ in the graph. Finally we backtrack, removing w from μ and returning to variable x, and repeat the process until all bindings for the variables of Q are found. In this example, this was the only answer for Q.

3 The Ring Index

In the sequel we show how the Ring implements the functionality needed to implement the LTJ algorithm (see Section 2.3) over a different data representation: the Ring index [7, 5].

3.1 The circular strings model

While LTJ is classically presented as running on tries, we take a more abstract view and consider the database triples (s, p, o) as *circular strings* [12], where we regard all the strings *spo*, *osp*, and *pos* as the same. We formalize circular strings using the concept of rotation.

▶ **Definition 1.** *A* rotation *of a string* $S[1..n]$ *is any* $S[i+1..n]S[1..i]$*, for* $0 \le i < n$.

▶ **Definition 2.** *The* circular string *(or* c-string*)* $\mathring{S}$ *is defined as the set of the rotations of* S.

In particular, $\mathring{spo} = \{spo, osp, pos\}$; note that $\mathring{spo} = \mathring{osp} = \mathring{pos}$. Note that, because the alphabet of the predicates is disjoint from that of subjects and objects, the three rotations of Def. 2 must be different in our case (because p is at a different position). Therefore, there is a one-to-one correspondence between triples and c-strings.

The graph database is then seen as a set of c-strings, one per triple, and we identify each trie node v with the set of c-strings corresponding to the triples that descend from v. Figure 3 shows the c-strings corresponding to three sample nodes of tries T_{spo} and T_{pos}.

For what follows, we will create two disjoint copies, Σ_{S} and Σ_{O}, of the alphabet Σ_V, in order to distinguish subjects from objects in c-strings. We will call $\Sigma_{\mathrm{P}} = \Sigma_L$ the alphabet of predicates for consistency. Thus, triples will belong to $\Sigma_{\mathrm{S}} \times \Sigma_{\mathrm{P}} \times \Sigma_{\mathrm{O}}$. Further, the three sub-alphabets will form disjoint lexicographic ranges. In our running example, we separate the alphabets by adding $|\mathcal{U}|$ to the p components of each triple in the graph, while the o components are increased by $2|\mathcal{U}|$. Let us also redefine triple patterns.

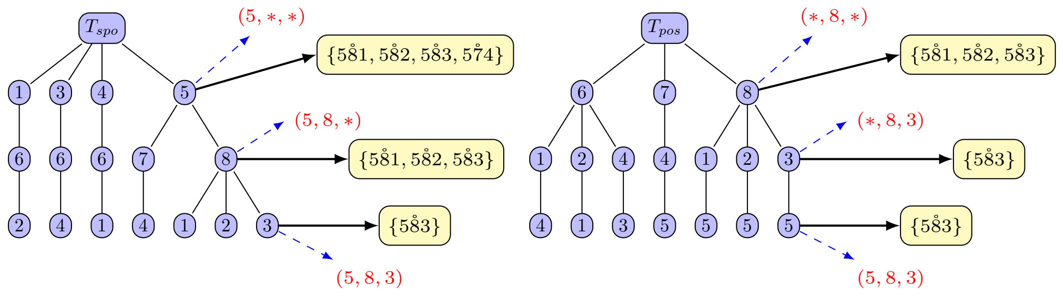

T_{spo}
$(5,*,*)$
$\{5\mathring{8}1, 5\mathring{8}2, 5\mathring{8}3, 5\mathring{7}4\}$
$(5,8,*)$
$\{5\mathring{8}1, 5\mathring{8}2, 5\mathring{8}3\}$
$\{5\mathring{8}3\}$
$(5,8,3)$
T_{pos}
$(*,8,*)$
$\{5\mathring{8}1, 5\mathring{8}2, 5\mathring{8}3\}$
$(*,8,3)$
$\{5\mathring{8}3\}$
$\{5\mathring{8}3\}$
$(5,8,3)$

Figure 3 The c-strings corresponding to three nodes of tries T_{spo} and T_{pos} (see solid arrows). The corresponding triple patterns $t(v)$ (according to Definition 3) are also shown for these nodes (see dashed arrows).

▶ **Definition 3.** *A* triple pattern *is an element of* $(\Sigma_S \cup \{*\}) \times (\Sigma_P \cup \{*\}) \times (\Sigma_O \cup \{*\})$*, where* $*$ *is a special symbol. We say that the elements* $*$ *are* unbound *and the others are* bound*. A triple* (s,p,o) matches *a triple pattern* (s',p',o') *iff* $x = x'$ *or* $x' = *$ *for every* $x \in \{s,p,o\}$*.*

Every trie node v then corresponds univocally to a triple pattern $t(v)$, where the edges that lead from the root to v define the bound components of $t(v)$. The node v then represents all the database triples that match $t(v)$. Figure 3 also shows the triple patterns corresponding to three nodes of tries T_{spo} and T_{pos}.

For our purposes, a key property of the characterization as circular strings is the following.

▶ **Theorem 4.** *The bound components of any triple pattern* $t = (s',p',o')$ *can be concatenated into a string* $P(t)$ *such that a triple* (s,p,o) *matches* t *iff* $P(t)$ *prefixes exactly one string of its c-string* $s\mathring{p}o$*. The prefixed rotation of* $s\mathring{p}o$ *is unique if* $P(t) \neq \varepsilon$*.*

Proof. Let us consider all the possibilities for $|P(t)|$.

If (s,p,o) matches t and the latter has one bound component, then $P(t)$ is univocally s, p, or o, and it prefixes spo, pos, or osp, respectively. It prefixes only one of those because the triple components have disjoint alphabets. Conversely, if $|P(t)| = 1$ and $P(t)$ prefixes spo, pos, or osp, then (s,p,o) matches t.

If there are two bound components, then we can form $P(t) = sp$, $P(t) = po$, or $P(t) = os$, which prefixes spo, pos, or osp, respectively. Again, $P(t)$ prefixes exactly one of those. Conversely again, if $|P(t)| = 2$ and it prefixes spo, pos, or osp, then (s,p,o) matches t. Note that the other choices, $P(t) = ps$, $P(t) = so$, and $P(t) = op$, do not prefix any suffix.

If all the components are bound, then three choices of $P(t)$ prefix (and equal) a string of $s\mathring{p}o$ for the only triple (s,p,o) that matches t. Finally, if there are no bound components, then $P(t) = \varepsilon$ prefixes every string, and conversely every triple matches t. ◀

Building on this result, it will suffice to represent tries T_{spo}, T_{osp}, and T_{pos}. Our plan is to identify every trie node v with a set of rotations, which except for the root will contain exactly one rotation of each c-string $s\mathring{p}o$ whose triple (s,p,o) matches $t(v)$. By Theorem 4, the rotations are exactly those that start with a given string $P(t)$, so v descends from the root by $P(t)$. The way to efficiently represent that set of suffixes is considered next.

For example, consider the trie T_{pos} on the right of Figure 3. Its rightmost node v descending by 83 has $t(v) = (*,8,3)$ and $P(t) = 83$. The node represents the single c-string $5\mathring{8}3$, and contains exactly one rotation of that c-string, 835, which starts with $P(t)$.

3.2 The extended Burrows-Wheeler Transform

The *extended BWT (eBWT)* [28, 14] is an extension of the Burrows-Wheeler Transform (BWT) [13] to a set of strings. It builds on the concept of *omega-order* [28] on strings, which is defined as follows.

▶ **Definition 5.** *An* infinite string *S over alphabet Σ is a mapping $S : \mathbb{N}^+ \rightarrow \Sigma$. The* prefix *$S[..i]$ of the infinite string S is the finite string $S[1]S[2] \cdots S[i]$. The* suffix *$X = S[i..]$ of the infinite string S is the infinite string such that $X[j] = S[i+j-1]$.*

▶ **Definition 6.** *The* ω-extension *S^ω of a string S is the infinite string formed by concatenations of strings S, that is, $S^\omega[i] = S[1 + ((i-1) \bmod |S|)]$.*

▶ **Definition 7.** *Given a string S, let $m \geq 1$ be the maximum value such that $S = U^m$ for some string U; we then say that $exp(S) = m$ and $root(S) = U$. The* omega-order*, denoted $<_\omega$, between strings S and T is then defined as follows: $S <_\omega T$ iff*

- *$root(S) = root(T)$ and $exp(S) < exp(T)$, or*
- *$S^\omega < T^\omega$, where $<$ is the lexicographic order.*

Note that, if S and T are not a proper prefix of the other, then $<_\omega$ coincides with the lexicographic order, but it can differ otherwise.

▶ **Definition 8.** *Given a set of strings $\{S_1, \ldots, S_n\}$, the eBWT is a permutation of the symbols in the strings S_i given by listing the rotations of all strings S_i in omega-order, and then concatenating the last symbols of each rotation.*

We are now in position to define the Ring index.

▶ **Definition 9.** *The* Ring index *of a set $\mathcal{D}$ of triples is the eBWT of the set $\mathcal{S} = \{spo,\ (s,p,o) \in \mathcal{D}\}$, after properly separating the alphabets into Σ_S, Σ_P, and Σ_O.*

Because of the alphabet separations, it follows that the rotations X of all the c-strings $\mathring{spo}$ have $root(X) = X$ and $exp(X) = 1$; therefore the omega-order boils down to the lexicographic order between their ω-extensions. Further, because of our alphabet separation, such order boils down in turn to the plain lexicographic order of the strings. The Ring's eBWT then lists all the strings of all c-strings $\mathring{spo}$ in the graph, sorts them lexicographically, and collects their last symbols.

Figure 4 (left) shows all the $\mathring{spo}$ strings corresponding to the edges of the graph from Figure 1. Since $|\mathcal{U}| = 8$ in this case, we sum 8 to the p component of each triple, and 16 to the o components, and thus $\Sigma_S = [1..8]$, $\Sigma_P = [9..16]$, and $\Sigma_O = [17..24]$. On the right, the triples are lexicographically sorted, obtaining the corresponding Ring index as the last column of the table.

The following theorem shows that we can identify each node v of the three chosen tries with a range of the eBWT of the Ring index.

▶ **Theorem 10.** *There is a distinct interval in the Ring representing the triples that descend from each node v of T_{spo}, T_{osp}, and T_{pos}.*

Proof. If v is the root, we can define its range as the whole eBWT. Otherwise, by Theorem 4, there is a string $P(t)$ for the triple pattern $t = t(v)$ that prefixes exactly one string of the c-string $\mathring{spo}$, or equivalently a rotation of spo, for each triple (s,p,o) matching t. By Def. 9, the three rotations of spo are listed in the Ring in lexicographic order, and thus those prefixed by $P(t)$ form a single range. Those ranges are unique once we define how $P(t)$ is formed. As

4	14	17
14	17	4
17	4	14
3	14	20
14	20	3
20	3	14
5	16	19
16	19	5
19	5	16
5	15	20
15	20	5
20	5	15
5	16	17
16	17	5
17	5	16
5	16	18
16	18	5
18	5	16
1	14	18
14	18	1
18	1	14

sort
$\Longrightarrow$

1	14	18
3	14	20
4	14	17
5	15	20
5	16	17
5	16	18
5	16	19
14	17	4
14	18	1
14	20	3
15	20	5
16	17	5
16	18	5
16	19	5
17	4	14
17	5	16
18	1	14
18	5	16
19	5	16
20	3	14
20	5	15

Figure 4 On the left, the 3 rotations of each c-string $\mathring{spo}$ of our running example graph. On the right, the strings are lexicographically sorted. The last column is the corresponding eBWT. The outer red rectangle corresponds to the node v of T_{spo} with $t(v) = (5, *, *)$, and the inner one to $t(v) = (5, 16, *)$.

seen in Theorem 4, there are multiple choices to form $P(t)$ only when $|P(t)| = 3$, that is, when t is a triple. In this case it corresponds to just one c-string $\mathring{spo}$ and the singleton range can be arbitrarily fixed to be that of any of its three rotations. ◀

Consider, for instance, the node corresponding to $t = (5, *, *)$ in T_{spo} of Figure 2, whose subtree includes the triples $(5, 7, 4)$, $(5, 8, 1)$, $(5, 8, 2)$, and $(5, 8, 3)$. This node is represented by the interval $[4..7]$ in the Ring shown in Figure 4 (right side), highlighted with the larger red box, which contains exactly the same triples mentioned before (with components p and o shifted accordingly, as we have already explained). On the other hand, interval $[5..7]$ (highlighted with the smaller red box) corresponds to the node for triple pattern $t = (5, 8, *)$ in the trie T_{spo} of Figure 2. Actually, notice that interval $[1..7]$ in this sample Ring corresponds to the root of trie T_{spo}, interval $[8..14]$ is associated with the root of trie T_{pos}, and interval $[15..21]$ represents the root of trie T_{osp}. Each subinterval within these regions corresponds to nodes in the respective tries.

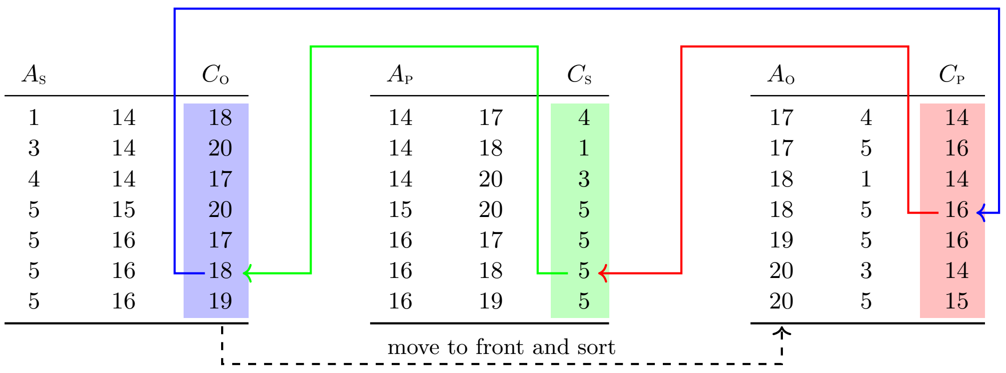
Roughly speaking, the Ring index then maps the trie operations to operations on the eBWT of the triples. In the sequel we show precisely how this is done.

3.3 The eBWT as a representation of the graph

The following definition will help us discuss how the eBWT works on the graph.

▶ **Definition 11.** *Let $S = a_1a_2a_3 \cdots a_k$ be a string where $a_i \in \Sigma_i$ for all i. We then say that the* order *of S is the string $123 \cdots k$.*

In our case, the order of spo is SPO, the order of pos is POS, and the order of osp is OSP. Because the three alphabets form lexicographic ranges, eBWT contains a range where all the listed rotations are of order SPO, and thus the eBWT contains only symbols of Σ_{O}, a second range where all the rotations are of order POS and the eBWT contains only symbols of Σ_{S}, and a third range where all the rotations are of order OSP and the eBWT contains only symbols of Σ_{P}. We store those three segments of the eBWT, respectively, as strings,



A_S		C_O	A_P		C_S	A_O		C_P
1	14	18	14	17	4	17	4	14
3	14	20	14	18	1	17	5	16
4	14	17	14	20	3	18	1	14
5	15	20	15	20	5	18	5	16
5	16	17	16	17	5	19	5	16
5	16	18	16	18	5	20	3	14
5	16	19	16	19	5	20	5	15

Figure 5 The Ring index, formed by the three columns C_X, and how each column is obtained by moving some other column to the front and reordering (the dashed arrow shows the case SPO $\rightarrow$ OSP). We also show, with solid arrows, the process for retrieving a triple in 5,16,18 from the index.

called *columns*, $C_O[1..N]$, $C_S[1..N]$, and $C_P[1..N]$ (see the different ranges and colors in the last column in Figure 4), and say that their orders are SPO, POS, and OSP, respectively.

Let us first show that our representation allows accessing any desired triple (s, p, o) starting from its position in any of the columns C_X. For example, to retrieve the triple represented at $C_O[i]$, we know immediately that $o = C_O[i]$. Now, by the classic BWT invariant [13, 16] (which holds true in the eBWT [28, Prop. 10]), we find the entry $C_P[j]$ corresponding to $C_O[i]$ with

$$j = \mathsf{LF}_O(i) = A_O[o] + \mathsf{rank}_o(C_O, i), \tag{1}$$

where $A_X[c]$ counts the number of times symbols smaller than c occur in C_X, and $\mathsf{rank}_c(X, i)$ counts the number of times c occurs in $X[1..i]$. With j we obtain $p = C_P[j]$, compute the position k of the triple in C_S with

$$k = \mathsf{LF}_P(j) = A_P[p] + \mathsf{rank}_p(C_P, j),$$

and complete the process with $s = C_S[k]$. The eBWT invariants imply that

$$i = \mathsf{LF}_S(k) = A_S[s] + \mathsf{rank}_s(C_S, k),$$

that is, we cycle on the string: the eBWT effectively regards the strings as circular and this permits retrieving the triples starting from any of their components.

See Figure 5 for an example of this process. In this case, we have $i = 6$, and hence $o = C_O[6] = 18$. So, $j = A_O[18] + \mathsf{rank}_{18}(C_O, 6) = 2 + 2 = 4$. Then, $p = C_P[4] = 16$, so $k = A_P[16] + \mathsf{rank}_{16}(C_P, 4) = 4 + 2 = 6$. Then, $s = C_S[6] = 5$. So, we get back to i since $i = A_S[5] + \mathsf{rank}_5(C_S, 6) = 3 + 3 = 6$. The corresponding *spo* triple is $(5, 16, 18)$.

The rationale of (say) Eq. (1) is the following. C_O corresponds to the order SPO and C_P to the order OSP. The second order is obtained from the first by stably reordering the tuples by their O component, in accordance with the lexicographic order. In a stable reordering of SPO, position $C_O[i] = o$ will be moved to position j such that (i) all the symbols less than o come before j (and those are counted in $A_O[o]$), and (ii) all the symbols o preceding the one at $C_O[i]$ also come before j (thus we add $\mathsf{rank}_o(C_O, i)$). Figure 5 also shows one of those reorderings, from SPO to OSP.

Because the shifted symbols are stored in different strings C_X, the Ring remaps their alphabets back to $\Sigma_X = [1..|\Sigma_X|]$. The nodes that are both subjects and objects use the same identifier in both Σ_{S} and Σ_{O}, in the range $[1\,..\,|\Sigma_{\mathrm{S}} \cap \Sigma_{\mathrm{O}}|]$. Also, the Ring stores the arrays A_{X} using $N + o(N)$ bits, whereas function rank is computed in time $O(\log |\mathcal{U}|)$ using a wavelet tree [22] representation of the columns $C_{\mathrm{X}}[1\,..\,N]$, just like a typical FM-index implementation [17]. Overall, it represents a graph database of N triples over universe $\mathcal{U}$ using $3N \log |\mathcal{U}| + o(N \log |\mathcal{U}|)$ bits (logarithms are to the base 2), and can retrieve any triple in time $O(\log |\mathcal{U}|)$. This is almost the same space needed to store the $3N$ triple identifiers in plain form. What is striking is that, within this space, the Ring can implement all the operations required by the LTJ algorithm, as we see next.

3.4 Implementing the operations on the eBWT

Having separate columns, it is more convenient to identify the root of a trie starting with symbol of alphabet Σ_{X} as the whole range $C_{\mathrm{X}}[1\,..\,N]$. This implements operation root depending on the next triple component we will descend by.

Now assume we are in a range of the eBWT (that is, a range of C_{S}, C_{P}, or C_{O}) that corresponds to a trie node v. Recall that this means that the range is that of all the rotations that start with $P(t(v))$. Let us show how to simulate the trie operations child and leap on this representation.

The "easy" case

In the "easy" case, v is represented as $C_{\mathrm{X}}[i\,..\,j]$ and the children of v belong to the alphabet Σ_{X}. We can then use an extension of the LF_{X} functions to compute the eBWT range corresponding to a desired child of v. For example, if we are at $v = \mathsf{root}()$ and want to descend to $\mathsf{child}(v, s)$, then we move from the range $C_{\mathrm{S}}[1\,..\,N]$, which represents v, to the range $C_{\mathrm{O}}[i\,..\,j] = C_{\mathrm{O}}[A_{\mathrm{S}}[s]+1\,..\,A_{\mathrm{S}}[s+1]]$, which represents $u = \mathsf{child}(v, s)$. This is the range of all the rotations starting with s. We can now descend to $\mathsf{child}(u, o)$ with the famous *backward search* formula [16], which essentially computes the range of all values $\mathsf{LF}_{\mathrm{O}}(k)$ for all $i \le k \le j$ where $C_{\mathrm{O}}[k] = o$:

$$\begin{aligned} i' &= A_{\mathrm{O}}[o] + \mathsf{rank}_o(C_{\mathrm{O}}, i-1) + 1, \qquad (2) \\ j' &= A_{\mathrm{O}}[o] + \mathsf{rank}_o(C_{\mathrm{O}}, j), \end{aligned}$$

so that now $C_{\mathrm{P}}[i'\,..\,j']$ is the range of all the rotations starting with os, and represents $w = \mathsf{child}(u, o)$. If we further descend to $\mathsf{child}(w, p)$, the analogous formulas will boil down to a single entry of C_{S} corresponding to the trie leaf representing the triple (s, p, o) (in this case, using the order POS).

For instance, assume that we want to go down by $s = 5$ and then by $o = 19$. We start from the range $C_{\mathrm{O}}[1\,..\,7]$, corresponding to $T_{spo}.\mathsf{root}()$. Going down to the child labeled $s = 5$ corresponds to restricting the range to $C_{\mathrm{O}}[A_{\mathrm{S}}[5] + 1\,..\,A_{\mathrm{S}}[6]] = C_{\mathrm{O}}[4\,..\,7]$. Then, descending by $o = 19$ from this node corresponds to moving to the range $C_{\mathrm{P}}[i'\,..\,j']$, with $i' = A_{\mathrm{O}}[19] + \mathsf{rank}_{19}(C_{\mathrm{O}}, 3) + 1 = 4 + 1 = 5$ and $j' = A_{\mathrm{O}}[19] + \mathsf{rank}_{19}(C_{\mathrm{O}}, 7) = 4 + 1 = 5$. The range $C_{\mathrm{P}}[5\,..\,5]$ then corresponds to the range of a node v in the trie T_{osp}, which represents the triple pattern $t(v) = (5, *, 19)$. Note that, although we do not represent the trie T_{sop}, we can find a node in some trie that corresponds to the same set of c-strings.

Operation $\mathsf{leap}(v, c_0)$ is slightly more complex, and resorts to the geometric capabilities of the wavelet tree. In the "easy case" again, node v is represented by a range $C_{\mathrm{X}}[i\,..\,j]$ and c_0 belongs to Σ_{X}; for example we are in $C_{\mathrm{O}}[i\,..\,j]$ and c_0 is an object. We then use the wavelet tree ability to compute, in $O(\log |\mathcal{U}|)$ time, the least value not smaller than c_0 in a range $[i\,..\,j]$ of the string C_{X} it represents [21].

The "hard" case

If v is the root and we descend by triple component $\textsc{x}$, we are always in the "easy" case, because we can use $C_{\textsc{x}}[1..N]$ as $\mathsf{root}()$. It is also always the easy case if $t = t(v)$ has two bound components (or, equivalently, v has depth 2): given any length-2 prefix $P(t)$ of the eBWT rotations, their range is in $C_{\textsc{x}}$, where $\textsc{x}$ is the unbound component of t. For example, if $P(t) = sp$, then its eBWT range is within $C_{\textsc{o}}$. The only "hard" cases occur when exactly one component is bound and we are forced to move "forward", instead of "backward" as Eq. (2) does: if only $\textsc{s}$ is bound in t (the cases of $\textsc{p}$ and $\textsc{o}$ are analogous), then the range of $P(t)$ is within order $\textsc{spo}$, and then the representation of v is of the form $C_{\textsc{o}}[i..j]$. We can then descend by o using Eq. (2), but we cannot descend by p in this way. Rather, we must restrict the range $C_{\textsc{o}}[i..j]$ so that it goes from representing all the rotations starting with s to representing all the rotations starting with sp; the order is still $\textsc{spo}$.

To descend in this hard case, we reorder the string $P(t)$: we descend from the root $C_{\textsc{p}}[1..N]$ by p, and from the resulting interval $C_{\textsc{s}}[i'..j']$ of all the rotations starting with p, we descend by s to the interval $C_{\textsc{o}}[i''..j'']$ of the rotations starting with sp.

For instance, assume that we have instantiated $s = 5$, which corresponds to $C_{\textsc{o}}[4..7]$ in our example Ring. This range is depicted by the larger box in the Ring of Figure 4. Next, suppose that we instantiate $p = 16$ and want to go down in the corresponding trie. The resulting range is represented by the smaller box in Figure 4. Notice that in this case we remain at $C_{\textsc{o}}$ (i.e., in trie T_{spo}) after descending with $p = 16$ (unlike the "easy" case, where we jump to a different trie after going down). However, using $\mathsf{LF}_{\textsc{p}}$ to go down by $p = 16$ is not feasible since we are currently in $C_{\textsc{o}}$. Instead, we descend from $C_{\textsc{p}}[1..7]$ to $C_{\textsc{s}}[A_{\textsc{p}}[16]+1..A_{\textsc{p}}[17]] = C_{\textsc{s}}[5..7]$, and then proceed with $s = 5$ to obtain the desired range $C_{\textsc{o}}[i'..j']$, with $i' = A_{\textsc{s}}[5] + \mathsf{rank}_5(C_{\textsc{s}}, 4) + 1 = 5$ and $j' = A_{\textsc{s}}[5] + \mathsf{rank}_5(C_{\textsc{s}}, 7) = 7$.

Finally, we implement $\mathsf{leap}(v, c_0)$ in the hard case as follows. Consider the same case above, where only s is bound, v is represented by $C_{\textsc{o}}[i..j]$, and $p_0 \in \Sigma_{\textsc{p}}$. We start from $C_{\textsc{s}}[A_{\textsc{p}}[p_0]+1..N]$, which is the range of all the rotations starting with predicates $p \geq p_0$. We then map that range using an analogous of Eq. (2):

$$\begin{aligned} i' &= A_{\textsc{s}}[s] + \mathsf{rank}_s(C_{\textsc{s}}, A_{\textsc{p}}[p_0]) + 1, \\ j' &= A_{\textsc{s}}[s+1], \end{aligned}$$

and obtain $C_{\textsc{o}}[i'..j']$, the range of all the rotations starting with s and following with some $p \geq c_0$. The first element of that range, $C_{\textsc{o}}[i']$, is a triple with the desired minimum value $p \geq p_0$. We then obtain p by traversing the circular string as described in Section 3.3.

4 Rings in Higher Dimensions

The theory of worst-case-optimal join algorithms was initially developed for the relational model, and only then adapted to graph databases [25], where BGPs are modeled mainly as multijoins on a single table of three columns. The space problem is even more serious for tables with more than three attributes: a table with d attributes (which we will say to be of dimension d) needs $d!$ tries, each storing the equivalent of a copy of the database, in order to implement worst-case-optimal joins using LTJ. This, of course, makes the technique unaffordable even for modest amounts of attributes! A natural question is: can we extend the Ring to dimensions $d > 3$ so as to use space proportional to the raw data?

The answer is, unfortunately, no: the key Theorem 4 does not hold for higher dimensions. Already for $d = 4$, let $\{\textsc{s}, \textsc{p}, \textsc{o}, \textsc{g}\}$ be the attributes. No matter how we choose the c-strings (say, $\mathring{spog}$), there are subsets of attributes for which $P(t)$ do not prefix any rotation of $\mathring{spog}$ (say, $\{\text{S},\text{O}\}$ or $\{\text{P},\text{G}\}$).

It is not hard to see, however, that *two* c-strings, like $\mathring{spog}$ an $\mathring{sopg}$, suffice for Theorem 4 to hold on $d = 4$, in the sense that any string $P(t)$, in some convenient order, prefixes some rotation of $\mathring{spog}$ *or* of $\mathring{sopg}$. Compared to storing $4! = 24$ tries, storing just two eBWTs yields an enormous space reduction!

The Ring simulation of Section 3 can be extended to having two or more eBWTs, by jumping from one to another as needed. The following definition captures what is needed from the chosen set of c-strings for such a simulation to work.

▶ **Definition 12.** *A string Y is a substring of a c-string $\mathring{S}$ iff it is a substring of S^ω. If $|Y| \leq |S|$, this is equivalent to Y being a substring of some rotation of S.*

▶ **Definition 13.** *A set $\mathcal{S} = \{S_1, S_2, \ldots\}$ of strings over alphabet $\Sigma_d = \{1, 2, \ldots, d\}$ is* complete *for dimension d iff, for every $\mathcal{Y} \subset \Sigma_d$ and $\text{X} \in \Sigma_d \setminus \mathcal{Y}$, $\mathcal{Y}$ can be ordered to form a string Y such that $Y\text{X}$ or $\text{X}Y$ are a substring of some $\mathring{S}_i \in \mathcal{S}$.*

The elements of $\mathcal{S}$ are strings *of attributes*, for example $\mathcal{S} = \{123\}$ is complete for $d = 3$ and $\mathcal{S} = \{1234, 1324\}$ is complete for $d = 4$. A Ring for dimension d is built on a complete set $\mathcal{S}$ of strings of length d. For each data tuple $(a_1, \ldots, a_d)$ and each $S_i \in \mathcal{S}$, we extract the string $a_{S_i[1]} \cdots a_{S_i[d]}$ (a permutation of the tuple). For example, with $d = 3$ and $\mathcal{S} = \{123\}$, we extract from the tuple (s, p, o) the string *spo*. With $d = 4$ and $\mathcal{S} = \{1234, 1324\}$, we extract from the tuple (s, p, o, g) the strings *spog* and *sopg*. The Ring consists of one eBWT for each string S_i, which indexes the rotations of all the c-strings $a_{S_i[1]} \overset{\circ}{\cdots} a_{S_i[d]}$.

Consider a trie node v representing a tuple pattern $t = t(v)$, now of d elements. If $\mathcal{S}$ is complete, then by definition it is possible to concatenate the bound elements of t in some order into a string $P(t)$ that prefixes all rotations in the eBWT of some S_i. We note that there could be various ways to concatenate the bound attributes of t to form strings $P(t)$ so that they prefix rotations in different eBWTs.

We can then descend from v to $\mathsf{child}(v, x)$, with $x \in \Sigma_{\text{X}}$, in two ways. The "easy" one, where v is represented by a range in C_{X}, uses the analogous of Eq. (2). The "hard" case discards the current range and recomputes the result from scratch: it forms $P(t(v))$ such that its order of attributes Y, preceded or followed by X, occurs in some $\mathring{S}_i$ (this works when $\mathcal{S}$ is complete). It then retraverses the tries from the root following the path of $P(t(v))x$ or $xP(t(v))$, using the analogous of Eq. (2) at each step.

Suppose, with Rings for SPOG and SOPG, that we want to go down using s, then o, then p, and finally g. Initially, without further information, we may descend by s from the root of the trie T_{spog}, obtaining a range of the SPOG Ring. Then, when it comes to go down by o, we note that this is not possible on this Ring, so we move to the SOPG Ring: we start with o and continue by s using the "easy" case, reaching a node in the trie T_{sopg} (i.e., a range in the SOPG Ring). To proceed with p, we can remain in the same Ring, yet it corresponds to the "hard" case, because p is to the right of *so* in the order SOPG. This can be solved with "easy" cases, either by re-running p, o, and s on the same SOPG Ring, or using o, p, and s if we move to SPOG. In both cases, the final descent by g corresponds to an easy case.

To implement $\mathsf{leap}(v, x_0)$, there are also two possibilities. In the easy case, we proceed on the wavelet tree exactly as explained (for the easy case) in Section 3. In the hard case, there are two subcases:

- We can form $P(t(v))$ whose order Y is such that $\text{X}Y$ occurs in some $\mathring{S}_i$. In this case, we retraverse the tries from the root to find Y, and finish as in the easy case. (Note that we were already in some range for the unbound attributes of $t(v)$, but not in one from where we could descend by x.)

- We can form $P(t(v))$ whose order Y is such that $Y\textsc{x}$ occurs in some $\mathring{S}_i$. In this case, we start from the range $C_{\textsc{x}}[A_{\textsc{x}}[x_0]+1, N]$ in the corresponding eBWT, and descend by $P(t(v))$ from it.

We then track the first cell in the resulting range, using LF, back to $C_{\textsc{x}}$ to find the answer x Because of the hard cases, we simulate each child or leap operation in time $O(d \log |\mathcal{U}|)$, which still yields a worst-case-optimal query algorithm, as d is independent of N.

Now that we know how to use multi-eBWT Rings to handle higher dimensions, the question is: how many eBWTs (i.e., strings in $\mathcal{S}$) are needed to be complete in dimension d? Arroyuelo et al. [5] studied this problem, showing that 5 eBWTs (instead of $5! = 120$ tries) are necessary and sufficient for $d = 5$, and 7 eBWTs (instead of $6! = 720$ tries) are necessary and sufficient for $d = 6$. The optimal number for $d = 7$ is shown to be between 10 and 12, and for $d = 8$ is between 21 and 25 (they obtain various lower bounds). In asymptotic terms, they prove that the number of eBWTs needed is $O(2^d)$ and $\Omega(2^d/\sqrt{d})$. These results make worst-case-optimal algorithms much more feasible for relational databases.

5 Order Graphs

A way to visualize complete sets $\mathcal{S} = \{S_1, S_2, \ldots\}$ is what are called *order graphs* [5].

▶ **Definition 14.** *An* order graph *of dimension d has, as nodes, all the permutations of $[1\,..\,d]$, and has directed labeled edges of the form $Z \xrightarrow{\textsc{x}} \textsc{x}Y$.*

In the order graphs that model complete sets of attribute strings, Z is always of the form $Y\textsc{x}$. For example, the order graph for $\mathcal{S} = \{123\}$ consists of the cycle

$$123 \xrightarrow{3} 312 \xrightarrow{2} 231 \xrightarrow{1} 123;$$

recall the example in Figure 5, where we can see how one can move from the SPO order to OSP using $\mathsf{LF}_{\textsc{o}}$, then to POS using $\mathsf{LF}_{\textsc{p}}$, and then back to SPO using $\mathsf{LF}_{\textsc{s}}$. The order graph for $\mathcal{S} = \{1234, 1324\}$ consists of the two cycles:

$$1234 \xrightarrow{4} 4123 \xrightarrow{3} 3412 \xrightarrow{2} 2341 \xrightarrow{1} 1234,$$

$$1324 \xrightarrow{4} 4132 \xrightarrow{2} 2413 \xrightarrow{3} 3241 \xrightarrow{1} 1324.$$

Each edge $e = Y\textsc{x} \xrightarrow{\textsc{x}} \textsc{x}Y$ of the order graph implies that the Ring stores a column C_e over the alphabet $\Sigma_{\textsc{x}}$, so that the function LF_e maps from order $Y\textsc{x}$ to order $\textsc{x}Y$. That is, LF_e applied on a position of the source order leads to the corresponding position in the target order. Backward search, as in Eq. (2), also maps from the first to the second order. The cycles arise because the eBWT, by definition, always stores the *last* column of its rotations.

An alternative way to understand Def. 13 is that, for every Y (in some order) and $\textsc{x}$, there must exist a *directed path* labeled $Y\textsc{x}$ or $\textsc{x}Y$ in the order graph: we can go from the source to the target node using LF or backward search and obtain the desired cell or range in the order of target node. This leads to a definition of complete order graphs.

▶ **Definition 15.** *An order graph of dimension d is* complete *iff, for every $\mathcal{Y} \subset \Sigma_d = [1\,..\,d]$ and $\textsc{x} \in \Sigma_d \setminus \mathcal{Y}$, $\mathcal{Y}$ can be ordered to form a string Y such that concatenating the labels over some directed path in the graph one obtains $Y\textsc{x}$ or $\textsc{x}Y$.*

P	O	S
14	17	4
14	18	1
14	20	3
15	20	5
16	17	5
16	18	5
16	19	5

move to front and sort

O	P	S
17	14	4
17	16	5
18	14	1
18	16	5
19	16	5
20	14	3
20	15	5

Figure 6 Reordering by the second column to go from order POS to order OPS.

$$\begin{tikzcd}
 & & 2341 \arrow[r, "1"] & 1234 \\
 & & 3421 \arrow[u, "2"] \arrow[dr, "1"] & \\
1423 \arrow[d, "3"] & 4213 \arrow[l, "1"] \arrow[ur, "3"] & & 1342 \arrow[dl, "2"] \\
3142 & & 2134 \arrow[ul, "4"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 2134 \arrow[r, "4"] & 4213 \arrow[r, "3"] & 3421 \arrow[d, "1"] \\
 & 1324 \arrow[u, "2"] \arrow[dl, "4"] & & 1342 \\
4132 \arrow[dr, "2"] & & 3241 \arrow[ul, "1"] & \\
 & 2413 \arrow[ur, "3"] & &
\end{tikzcd}$$

Figure 7 Two particular complete order graphs of minimum size 8 for $d = 4$.

The definition of order graph, however, opens up new alternatives that are not possible under the eBWT framework, namely when Z is not of the form YX in Def. 14. Consider the list of the rotations listed in lexicographic order by the eBWT, and regard the possibility of storing *another* column of the rotations, not necessarily the last. For example, if we take the order 1234 and store the third column, the corresponding LF function on that column will stably reorder the rotations by the attribute 3, so that the ordering obtained is 3124 (recall the rationale of LF given after Eq. (1)). Backward search also works correctly using the analogous of Eq. (2). Such a column corresponds to a graph edge $1234 \xrightarrow{3} 3124$, and the order graph is not anymore a set of cycles of length d. Figure 6 shows how this reordering by another column operates on our example graph of dimension $d = 3$.

While complete sets of c-strings yield complete order graphs, there are complete order graphs that do not correspond to complete sets. For $d = 4$, for example, we have a minimal complete order graph that consists of a single cycle of length 8, instead of two cycles of length 4 (for both, the index stores 8 columns C_e):

$$3421 \xrightarrow{1} 1342 \xrightarrow{2} 2134 \xrightarrow{3} 3214 \xrightarrow{4} 4321 \xrightarrow{1} 1432 \xrightarrow{2} 2143 \xrightarrow{4} 4213 \xrightarrow{3} 3421.$$

More curious optimal complete order graphs for $d = 4$ exist; Figure 7 shows two of them.

A natural question is whether complete order graphs can yield smaller Rings than complete sets. The answer is indeed affirmative! For $d = 5$, there exists a complete order graph formed by a single cycle of 20 edges [5], whereas the smallest complete set requires 5 eBWTs: seen as an order graph, that amounts to 5 cycles of length 5, that is, 25 edges. The length-20 cycle is known to be a minimal complete order graph for $d = 5$. In general, however, it is unknown whether some minimal complete order graph is always a single cycle, or a set of cycles. It is only known that some minimal complete order graph is always formed by a set of cycles with trees possibly sprouting from the nodes [5]. That is, every node must have indegree 1. As it can be inferred from this discussion, no effective ways to build minimal complete sets of c-strings or order graphs for a dimension d exist; we have resorted to exhaustive search.

6 Regular Path Queries

Assume an RPQ such that the initial node s is fixed. A classic solution to this problem is to build the nondeterministic automaton of the regular expression and traverse the graph from s, in DFS or BFS order. As we traverse an edge $s \xrightarrow{p} o$, we feed the automaton with symbol p, and abort the branch if the automaton runs out of active states. We must also record the active automaton states with which we had already visited each graph node, to avoid repeating the same nodes with the same states (i.e., we deactivate the repeated states in this second visit). We report (s, o) for every node o where we arrive with a final automaton state activated.

The other cases are solved analogously: if only the final node o is fixed, we can reverse the regular expression the symbols (i.e. convert every p with $\hat{p}$ and vice versa). If both are fixed, we can start from either of those and stop as soon as the other node is found with a final state. If none is fixed, we start a search from every possible value of s.

6.1 Simulation with the Ring

This process can be simulated with the Ring without using additional data structures. In this case, it is easier to start assuming that only o is fixed, and derive the other cases analogously, so we use the reversed automaton. Our first step is to descend from the trie root by o, $v = \mathsf{child}(\mathsf{root}(), o)$, which is represented by the range $C_{\text{P}}[i \mathinner{.\,.} j] = C_{\text{P}}[A_{\text{O}}[o] + 1 \mathinner{.\,.} A_{\text{O}}[o+1]]$. We now analyze the automaton states to determine the symbols that would lead to active states. For each such symbol p, we descend to $u = \mathsf{child}(v, p)$ using an analogous of Eq. (2), for predicate p instead of for object o. The resulting range, $C_{\text{S}}[i' \mathinner{.\,.} j']$, contains all the distinct sources s that lead to o by symbol p. Since there cannot be repetitions in this list, we extract each value of s and iterate from it as the new source node, that is, setting $o \leftarrow s$.

Note that we do not need the Ring component C_{O} for this simulation. Further, one can avoid duplicating the edges to support reversed symbols if one uses the "non-typical" case technique to handle them. Such a technique [32] was implemented over a bit-parallel simulation of the automaton, and shown to be very competitive in space and time with classic solutions. Further, it was shown that the used Ring components are equivalent to a known succinct representation of labeled graphs [31, Sec. 9.1.4]

6.2 Larger and smarter

An earlier Ring-based implementation [6], instead, does include the reversed edges, and thus it can always use the "typical" case formulas. This enables other optimizations that make the process faster. One possibility is that, instead of trying out the labels p that are relevant for the automaton, one can enumerate the distinct labels p that occur in $C_{\text{P}}[i \mathinner{.\,.} j]$, which could be less. The wavelet tree of C_{P} can do this enumeration in logarithmic time per resulting symbol [21]. Further, the wavelet tree can be enriched with information on the automaton states that are sources of symbols in ranges of Σ_L. Those can be intersected with the currently active states as we traverse the wavelet tree, so as to prune subtrees that cannot activate any state. This technique computes the intersection between the predicates that lead to active automaton states and those that leave from the current node, in optimal time, that is, proportional to the alternation complexity of both sets [11]. An analogous technique on the wavelet treee of C_s avoids visiting sources of $C_s[i' \mathinner{.\,.} j']$ that have already been visited with all the currently active states.

Using even more space (but still several times less than classic solutions), one can implement even smarter traversals, in considerably less time [6]. For example, the regular expression can be cut at an edge with a label p that is infrequent in the graph. From each edge $s \xrightarrow{p} o$, we launch a backward traversal from s to nodes that activate the initial automaton state, and a forward traversal from o to nodes that activate the final automaton states, and report the Cartesian product of both sets.

7 Perspective

We believe that the most important take-away message for the stringology audience is that there are algorithmic and combinatorial problems in the database community, particularly (but not only) in graph databases, that can benefit from approaches rooted in stringology: the LTJ algorithm works on tries, the Ring structure builds on the eBWT, and RPQs are an extended form of regular expression matching. We have also seen how combinatorial problems arise, like finding lower bounds to the number of Rings needed in dimension d, finding optimal ones efficiently, or characterizing the shapes of optimal order graphs. There is also space for the use of compact data structures: the Ring is a non-redundant version of the six tries used by LTJ, and we have used wavelet trees to solve the more complex LTJ primitives, as well as to find better query plans [4]. Other remarkable uses of compact data structure are compressed quadtrees to solve BGPs on any dimension d [9] and ordinal tree representations to extend BGPs with topological queries [20].

On the other hand, the application of stringology techniques to databases bounces back with new challenges and ideas. The most intriguing one is probably the idea of permuting the eBWT matrix using an arbitrary column, not necessarily the last one. Could this technique bring meaningful functionalities in other scenarios?

References

1 Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems*, 42(4):20:1–20:44, 2017.

2 Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, 31(3):1–26, 2022.

3 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

4 Diego Arroyuelo, Fabrizio Barisione, Antonio Fariña, Adrián Gómez-Brandón, and Gonzalo Navarro. New compressed indices for multijoins on graph databases. *CoRR*, 2408.00558, 2024.

5 Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, 29(2):article 5, 2024.

6 Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. Optimizing RPQs over a compact graph representation. *The VLDB Journal*, 33:349–374, 2024.

7 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In *Proc. 47th ACM International Conference on Management of Data (SIGMOD)*, pages 102–114, 2021.

8 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, and Domagoj Vrgoc. Tackling challenges in implementing large-scale graph databases. *Communications of the ACM*, 67(8):40–44, 2024.

9 Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):article 8, 2022.

10 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

11 Jérémy Barbay and Claire Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):4:1–4:18, 2008.

12 Nieves Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, Antonio Fariña, and Gonzalo Navarro. Space/time-efficient rdf stores based on circular suffix sorting. *The Journal of Supercomputing*, 79:5643–5683, 2023.

13 Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

14 Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In Hideo Bannai and Jan Holub, editors, *Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 25:1–25:18, 2022.

15 Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Engineering Bulletin*, 35(1):3–8, 2012.

16 Paolo Ferragina and Giovanni Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

17 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

18 Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):article 10, 2010.

19 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proc. 44th ACM International Conference on Management of Data (SIGMOD)*, pages 1433–1445. ACM, 2018.

20 José Fuentes-Sepúlveda, Adrián Gómez-Brandón, Aidan Hogan, Ayleen Iribarra-Cortés, Gonzalo Navarro, and Juan Reutter. Worst-case-optimal joins on graphs with topological relations. In *Proc. 34th International World-Wide Web Conference (WWW)*, 2025. To appear.

21 Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

22 Roberto Grossi, Ankur Gupta, and Jeff S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

23 Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. W3C recommendation, 2013. URL: `https://www.w3.org/TR/sparql11-query/`.

24 Aidan Hogan. *The Web of Data*. Springer, 2020.

25 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019.

26 Emil Eifrem Ian Robinson, Jim Webber. *Graph Databases (2nd Edition)*. O'Reilly Media, Inc., 2015.

27 Frank Manola and Eric Miller. *RDF Primer*. W3C Recommendation. 2004.

28 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theoretical Computer Science*, 387(3):298—312, 2007.

29 Amine Mhedhbi, Amol Deshpande, and Semih Salihoglu. Modern techniques for querying graph-structured databases. *Foundations and Trends in Databases*, 14(2):72–185, 2024.

30 Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019.

31 Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

32 Gonzalo Navarro and Josefa Robert. Compressed graph representations for evaluating regular path queries. In *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–232, 2024.

33 Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

34 Bryan B. Thompson, Mike Personick, and Martyn Cutcher. The bigdata® RDF graph database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.

35 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.