Compact Data Structures Meet Databases

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

Abstract

We describe two success stories on the application of compact data structures (cds) to solve the problem of the excessively redundant space requirements posed by worst-case-optimal (wco) algorithms for multijoins in databases, and particularly basic graph patterns on graph databases. The aim of cds is to represent the data *and* additional data structures on it, using total space close to that of the plain (and, sometimes, compressed) data, while efficiently simulating the data structure operations. Cds turn out to be a perfect approach for the described problem: We designed and implemented cds that effectively use space close to that of the plain or compressed data, which is orders of magnitude less than existing systems, while retaining worst-case optimality and performing competitively with those systems in query time, sometimes being even considerably faster.

2012 ACM Subject Classification Information systems → Data structures; Theory of computation → Data structures design and analysis

Keywords and phrases succinct data structures, tries, multidimensional grids, text searching

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.2

Category Invited Talk

Funding Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile.

1 Motivation

1.1 Graph databases

Graph databases [48, 26] have gained momentum with the rise of large unstructured repositories of information that emphasize relations between entities. They have become an attractive alternative to the relational model in cases where the information has no fixed structure. Dozens of graph database management systems [41, 51, 12, 36], prototypes [1, 35, 29, 2], models and languages [25, 17, 27, 3], and large repositories like Wikidata [54], illustrate how active is the interest on this relatively new technology.

A graph database represents information in the form of a labeled graph or network. There are many possible models to represent information in this way, such as knowledge graphs [27], property graphs [17], and RDF [34], to name a few. In general, the graph nodes represent objects and the edges between them represent relations. The models differ on what kind of information can be associated with the nodes or the edges, whether the edge labels can also be objects, and so on. For concreteness, we will focus on the RDF model, where the graph is seen as a set of *triples* (s, p, o), where s is the *subject* (or source node), p is the *predicate* (or label of the edge), and o is the *object* (or target node). Consider the graph of Figure 1 (cf. [4]) as our running example. The nodes are scientists and the Nobel prize. The arrows indicate that a scientist advised another and that a scientist won the Nobel prize.

The language to query a graph database also varies. In the widely used SPARQL standard [25], queries are built on relatively small *graph patterns*, which have to be matched in the database graph. In its simplest form, this can be just a *triple pattern*, which searches for the existence of a certain edge (or triple). The triple pattern specifies constants or variables for the subject, predicate, and object of the desired triples; every matching triple in the graph corresponds to *binding* the variables of the triple pattern. In our example, the

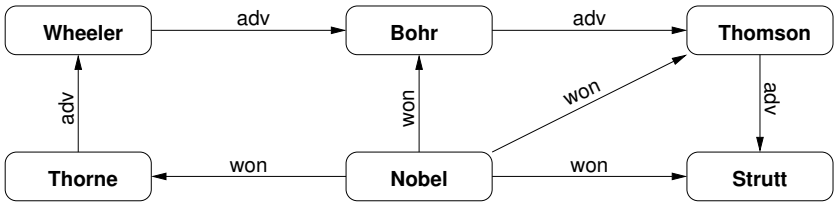

Figure 1 An example labeled graph.

triple $(\mathsf{Nobel}, \mathsf{won}, ?x)$ returns all the bindings of x to Nobel prize winners (i.e., $x = \mathsf{Thorne}$, $x = \mathsf{Bohr}$, etc.).

Basic graph patterns (BGPs) are sets of triple patterns sharing variables. They correspond to matching a subgraph in the database, returning all the variable bindings that make the subgraph occur. BGPs are akin to *multijoins* in relational databases, or *full conjunctive queries* in logic databases. In our example, the BGP

$$(?y, \mathsf{adv}, ?x),\ (\mathsf{Nobel}, \mathsf{won}, ?x),\ (\mathsf{Nobel}, \mathsf{won}, ?y) \tag{1}$$

returns pairs (y, x) where y advised x and both won the Nobel prize (those are $(y, x) = (\mathsf{Bohr}, \mathsf{Thompson})$ and $(y, x) = (\mathsf{Thompson}, \mathsf{Strutt})$).

The third type of graph pattern are the *regular path queries (RPQs)*. An RPQ is basically a regular expression that matches variable-length paths in the database graph, so that the sequence of traversed labels belong to the language denoted by the regular expression. RPQs are distinctive of graph databases and cannot be emulated in the relational algebra. In our example, the RPQ "$\mathsf{Wheeler}\ \mathsf{adv}^+\ ?x$" retrieves the academic descent of Wheeler (i.e., $x = \mathsf{Bohr}$, $x = \mathsf{Thomson}$, and $x = \mathsf{Strutt}$).

1.2 Worst-case optimality and the space problem

While triple patterns are easily solved with plain data retrieval structures, BGPs and RPQs are much more challenging and pose serious performance issues on graph database engines (e.g., it is typical to set timeouts in the minutes). Join evaluation is the most costly part in relational queries, and this carries over graph databases, where in addition it is not strange to see BGPs joining tens of triple patterns (e.g., up to 22 were found in a Wikidata query log [33]). Languages like SPARQL also enable projections, unions, and other operations, though the efficiency focus of database engines is generally on BGPs and RPQs.

An important breakthrough in the resolution of multijoin queries was the development of *worst-case optimal (wco)* join algorithms. A join algorithm is wco if its time complexity is of the order of the so-called *AGM bound* [7], that is, the maximum possible output of the query over some database with the same table attributes and sizes of the one at hand. It was shown that the techniques used by relational engines since the sixties, where multijoins were performed pairwise, were doomed to be non-wco. At the same time, several wco algorithms were developed [43, 44, 53, 31, 45, 42]. This technique was translated to graph databases [29], where it is particularly relevant because multijoins tend to be large and complex [45, 1, 30, 29]. It was shown that wco algorithms considerably outperformed traditional join algorithms on complex queries, especially when the BGPs contained cycles [1].

This improvement came at the cost of space, however. For example, the most popular wco algorithm, *Leapfrog Triejoin (LTJ)* [53], requires to index the rows of every database table as sequences of values in trie data structures, *in every possible ordering of the attributes*.

That is, a table of d columns needs to be stored in $d!$ tries. In particular, supporting wco joins on triples (s, p, o) poses a space overhead factor of $3! = 6$. Other wco algorithms pose similar or worse space problems. This is particularly unfortunate with the emergence of enormous repositories of unstructured data in graph form, and hinders the adoption of the faster wco strategies in the resolution of complex multijoin queries. To illustrate, Wikidata is approaching 14 billion triples,[1] so 6 copies of it, using just 32 bits per element and without the additional trie structures, surpasses the terabyte.

1.3 Compact data structures to the rescue

Our recent research has shown that the use of *compact data structures (cds)* can play a significant role in the reduction of the space required by wco multijoin algorithms. Compact data structures [38] aim to represent the data *and* its needed data structures within space close to the *entropy*, or amount of information, present in the data. There exist to date a number of compact representations for bit vectors, sequences, trees, graphs, matrices, point grids, texts, and many others. Cds have been very successful in reducing the size of relevant data structures by orders of magnitude, as well as greatly increasing the functionality of data representations within space close to the actual information of the data.

It is then more than natural to apply cds to the problem of supporting wco algorithms on graph databases, with the aim of retaining time optimality while removing the redundancy. We have recently proved the viability of this concept in two forms.

Qdags [40, 6]: We represented relations of d attributes as d-dimensional *point grids* called *qdags*, where every tuple becomes a point (qdags are a kind of compressed quadtrees [49, 50]). To solve a multijoin between several tables, qdags traverse and intersect all their grids in synchronization. The resulting algorithm not only was proved to be wco and to require only *one* copy of the data independently of d, but it was also shown to be competitive in time with the state of the art (at least for low d; the query time is exponential on d). Qdags use orders of magnitude less space than other indices on graph databases, actually *compressing* the graph to less than its plain size. As they are compositional (i.e., the result of a query is also a qdag) we also showed how to extend their functionality to the full relational algebra.

Ring [4, 5]: We represented the database triples as *texts*, and built on text indexing cds [14, 16] to support the LTJ algorithm. The resulting structure, the *ring*, is once again wco and uses just one copy of the data. In many cases (but not when d is very small), the ring is faster than qdags, but it requires $O(2^d)$ (not $d!$ as classical approaches) copies on d-dimensional tables, though one suffices no graph databases, where $d = 3$. Both qdags and the ring could index the Wikidata in under 70 GB. In a further development [5], we solved RPQs on the ring in time competitive with the state of the art, while using an order of magnitude less space than other indices. This solution uses techniques from text searching, like converting the RPQ to its Glushkov automaton [23, 39] and exploiting the flexibility of the wavelet trees [24, 37] used by the ring to represents its text.

Our results demonstrate that cds can be used to compactly represent graph databases while efficiently solving BGPs and RPQs. In this survey we describe those results in some detail and discuss the features and challenges of this new and promising technology. We

[1] https://grafana.wikimedia.org/d/000000489/wikidata-query-service

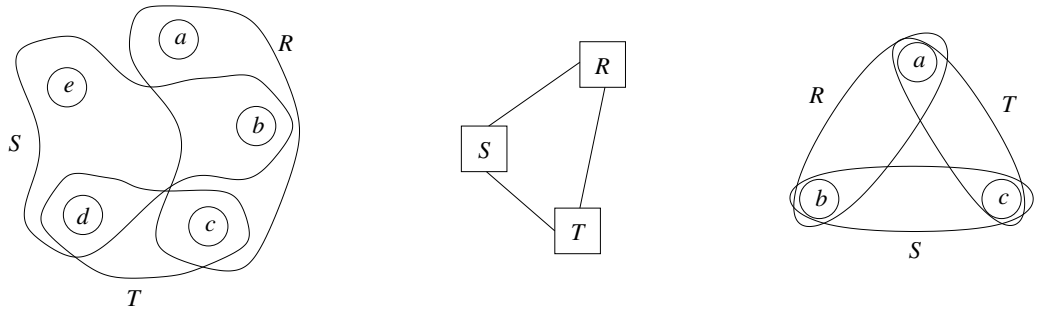

Figure 2 On the left, a join query seen as a hypergraph where nodes are attributes. In the middle, as a graph where nodes are relations. On the right, the triangle query on the graph where nodes are attributes.

strive for simplicity and informality in this introductory survey, further details and precisions can be found in the references.

2 State of the Art in (Graph) Databases

2.1 Multijoin queries

We focus on so-called *multijoin* queries, which compute the natural join between a set of tables (we discuss the case of graph databases soon). It is customary to regard multijoin queries as hypergraphs, where the nodes are attributes and the involved relations are hyperedges covering their attribute nodes. For example, the left hypergraph in Figure 2 corresponds to the join $R(a, b, c) \bowtie S(b, d, e) \bowtie T(c, d)$.

An alternative view, shown in the middle of the figure, is to regard the relations as nodes and put edges between relations that share attributes. We speak of *cyclic* and *acyclic* queries referring to this second kind of graph.

2.2 The AGM bound

Asterias, Grohe, and Marx [7] showed how to compute the maximum possible size of the output of a multijoin query. The maximization is done over every possible content of the tables participating in the join, while retaining their attributes and sizes. This bound, also shown to be tight, is since then known as the *AGM bound*. The bound immediately yields the concept of worst-case optimality: a join algorithm is *worst-case optimal (wco)* if the time it takes to compute a join is of the order of the AGM bound (possibly multiplied by a factor that depends at most polylogarithmically on the data size), because that is the possible output size for this query on *some* tables, and we need at least time to write the output. This differs from the stricter *instance optimal* algoritms, which take time proportional to the output of the query on *the given* tables.

The precise form of the bound is not important for our discussion; what is relevant for now is that the bound implies that *no* pair-wise join strategy can be wco. The paradigmatic example is the so-called "triangle query" $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ (see the right of Figure 2). If the tables have n rows, the triangle query can produce only $O(n^{3/2})$ results; however there exist tables R, S and T where every pairwise join strategy takes time $\Omega(n^2)$. It is worth noting that all the classical work on query plan optimization since the birth of the relational model built on pair-wise joins.

A number of wco algorithms appeared since then [43, 44, 53, 31, 45, 42]; we describe the most popular one in some detail next. For the particular case of acyclic queries, it is indeed possible to obtain the famous instance-optimal Yannakakis' algorithm [57]. It is also possible to obtain intermediate measures, like the *fractional hypertreewidth (fhw)*, which is related to the strategy of separating a cyclic query into a tree of cyclic components, solving each of those with a wco algorithm, and then solving the resulting acyclic query instance-optimally [1] (we omit some details for simplicity). The fhw bound is then the sum of the AGM bounds for the nodes in the best possible decomposition of the query into a tree of cycles.

2.3 Leapfrog Triejoin

Leapfrog Triejoin (LTJ) [53] is the most popular wco multijoin algorithm. Instead of performing the joins pair-wise (or, we could say, table-wise), LTJ proceeds *attribute-wise* over all the tables at the same time. We say that LTJ *binds* one attribute at a time, meaning that it finds all the possible values it may get in the output. Say that we decide to start by binding attribute A. We then find the values of A that appear in *all* the joined tables. For each such value $A = a$, we run a branch where the tables keep only their rows where A has value a, and continue binding the next variables. This branching continues until either there are no binding values for an attribute (and thus the current branch is abandoned), or we have bound all the attributes (and then we output all the possible combinations of the non-joined attributes, as a Cartesian product). We show an example soon.

For LTJ to run efficiently, it is convenient to arrange the tables as *tries* [19], or digital trees. Each row of the table becomes a root-to-leaf path in its trie. The order in which the attributes are read root to leaf must correspond to the order in which they are bound along the query process, and the attributes not participating in the join must come at the end. Each branch of LTJ then starts with a pointer to a node of the trie of each joined relation (all start at the root). When it comes to bind A, all the relations having attribute A intersect the children of their current node. For each value a that appears in the children of all the relevant trie nodes, LTJ descends to that child in all those tries and continues by that branch, where now we have bound $A = a$.

Because one cannot predict which attributes will be joined in queries, and furthermore it is convenient to choose different binding orders to improve performance, LTJ requires that each relation with d attributes is indexed in $d!$ tries, one per possible attribute ordering. This is the main problem for using LTJ in practice. An interesting alternative is to build query plans that combine wco algorithms with (non-wco) pairwise joins [35, 20].

2.4 The case of graph databases

A graph database can be seen as a single relation over three attributes; every edge $s \xrightarrow{p} o$ is interpreted as a tuple (s, p, o) in the relation (for subject, predicate, and object, following the RDF terminology [34]). Alternatively, it can be seen as a set of relations over two attributes, one per predicate p containing the pairs (s, o) such that the edge $s \xrightarrow{p} o$ is in the graph.

Standard query languages like SPARQL and many others feature two core query types, *Basic Graph Patterns (BGPs)* and *Regular Path Queries (RPQs)*.

Basic graph patterns BGPs can be seen as a composition of a selection and a join in the corresponding relational database. A BGP is a set of *triple patterns*, each describing a graph edge where each of the components s, p, and o can be a fixed constant (hence the selection) or a variable. Shared variables among the triple patterns of the BGP correspond to equijoins by

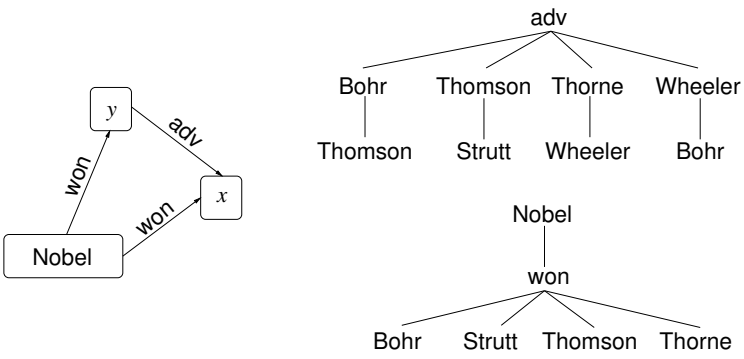

Figure 3 On the left, the subgraph of Eq. (1). On the right, the tries to traverse when solving this query using LTJ.

the corresponding attributes. When the predicates are constant, we can see the hypergraph of the query as a classic labeled digraph; we can support variable predicates by allowing labels be variables too. With this modelling, solving the BGP query corresponds exactly to finding all the bindings of the variables that make that graph be a subgraph of the database. On the left of Figure 3 we see the graph of the BGP of Eq. (1); note it is analogous to the triangle query.

By regarding every triple pattern *as a relation*, where some attributes may by bound from the beginning (if they are constants) or be named after a variable otherwise, we can adapt LTJ to solve BGPs, resulting in a wco algorithm as well [29]. The relevant parts of the tries for our example query, in the correct order to bind first y and then x, are shown on the right of Figure 3 (we use the first trie for $(?y, \mathsf{adv}, ?x)$, starting at the node "adv", and two copies of the second trie for $(\mathsf{Nobel}, \mathsf{won}, ?x)$ and $(\mathsf{Nobel}, \mathsf{won}, ?y)$, starting at the node "won"). When we bind y, we intersect the lists of children of both nodes, obtaining bindings $y = \mathsf{Bohr}$, $y = \mathsf{Thomson}$, and $y = \mathsf{Thorne}$. Branching with each such value of y we intersect the only child of each node in the first trie with the children of "won" in the second, obtaining $x = \mathsf{Thomson}$ when $y = \mathsf{Bohr}$, and $x = \mathsf{Strutt}$ when $y = \mathsf{Thomson}$.

The space issues of LTJ carry over the graph database formulation, so we require to store $3! = 6$ tries, each representing the whole database graph in a different order, (s, p, o), (s, o, p), (o, s, p), (o, p, s), (p, o, s), and (p, s, o). Alternatives are giving up with wco algorithms, as mentioned, or building some orders at query time, which is generally too expensive.

Regular path queries RPQs are akin to regular expressions that must be matched to paths in the database graph. They may fix the starting node x and/or the ending node y, and otherwise they specify the language of the sequences of labels that can connect x with y. Apart from the regular expression operations, one can use $\hat{}p$ to denote an edge labeled p in reverse direction. This can be handled by duplicating the graph database edges so as to include those reversed labels.

There are no wco algorithms for RPQs. The standard solution is to build the *product graph* between the graph database and the nondeterministic finite automaton (NFA) of the RPQ. The product graph has nodes (u, v) for each node u of the graph and v of the NFA. There is an edge $(u, v) \xrightarrow{p} (u', v')$ iff there is an edge $u \xrightarrow{p} u'$ in the graph and we can go from v to v' by symbol p in the NFA. We then traverse the product automaton from every possible initial node (x, i) (where x may be fixed or not in the RPQ and i is the initial NFA state) towards every possible final node (y, f) (where y may be fixed or not in the RPQ and

f is a final NFA state) and report all the pairs (x, y).

Several heuristics have been proposed over this basic solution [32, 56, 46], aiming at filtering the traversal of the product graph. For example, if the RPQ forces the existence of a certain label in the path that is infrequent in the graph, then it is more convenient to focus on those edges and trying to match the RPQ path in both directions from the arrow.

An elegant way to mix BGPs and RPQs is to permit triple patterns of the form (x, R, y), where x and y are the endpoints of the RPQ R. This can then be treated as just another relation to join. For example, one can run the RPQ and materialize the output, and then run the query as a normal BGP. Or one can run the rest of the BGP so that these triples, which are likely to be more expensive, are processed at the end, only for the bound variables that have survived all the intersections.

3 Compact Data Structures

We describe in this section the compact data structures we used in our developments. Again, we aim at an intuitive description; more details and references can be found elsewhere [38].

3.1 Bitvectors

A bitvector $B[1..n]$ is a sequence of n bits that provides the following two operations:

$\boldsymbol{rank_b(B, i)}$ is the number of bits equal to $b \in \{0, 1\}$ in $B[1..i]$.

$\boldsymbol{select_b(B, j)}$ is the position of the jth occurrence of $b \in \{0, 1\}$ in B.

It is possible to represent B within $n + o(n)$ bits so that both operations are supported in constant time [11]. It is also possible to represent B in compressed space when it has many more or fewer 0s than 1s, while retaining constant-time operations. Let m be the number of 0s, then the compressed representation uses $\log_2 \binom{n}{m} + o(n)$ bits [47].

3.2 Cardinal trees

A cardinal tree is a rooted tree where each node has children with labels in $[1..\sigma]$; each node has at most one child with a given label. The basic operations supported by this data structure are:

$\boldsymbol{root(T)}$ is the root node of T.

$\boldsymbol{child(v, a)}$ is the child of node v labeled a, or *null* if there is no such child.

$\boldsymbol{parent(v)}$ is the parent of node v, or *null* if v is the root.

It is possible to represent a cardinal tree with n nodes within $(\log_2 \sigma + 2 + o(1))n$ bits, while supporting the given operations in constant time [9]. We are going to use tries of alphabet size $\sigma = 4$, in which case a more practical representation using the same space is the k^2-tree [10] (with $k = 2$). It represents each node with 4 bits, which marks which children exist. The 4-bit signatures of all the nodes are concatenated in levelwise form, into a large bitvector $T[1..]$. The node identifiers correspond to the index of their signatures in this array. The root identifier is 0, corresponding to the first signature, and the identifier of the ith child of a node with identifier v is $child(v, i) = rank_1(T, 4v + i)$. The identifier of the parent of v, instead, is $parent(v) = \lceil select(T, v)/4 \rceil - 1$.

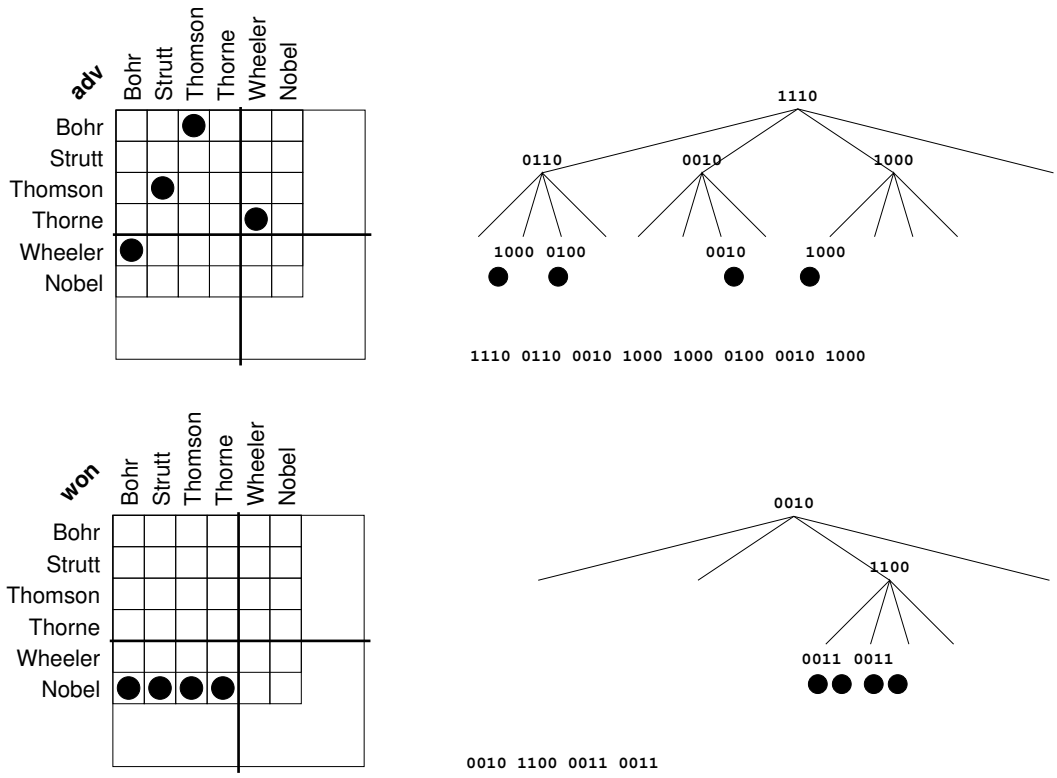

Figure 4 On the left, the relations adv and won of Figure 1 seen as two-dimensional grids. On the right, their representations as quadtrees, which are just cardinal trees of arity four (we show the signatures of the tree nodes), and their final representation as k^2-tree bitvectors at their bottom.

3.3 Quadtrees

A quadtree is a geometric data structure that represents points in a discrete two-dimensional grid. The quadtree is a tree of arity four. The root represents the whole grid, which is divided as evenly as possible into four quadrants. Each quadrant is recursively represented by a child of the root: top-left, top-right, bottom-left, and bottom-right. If the grid has no points, the corresponding quadtree node is a leaf and the grid is not further subdivided. When the nodes represent single cells, they also become leaves that store a point or not.

A compact representation of a quadtree can be obtained by seeing it as a cardinal tree of arity $\sigma = 4$. Each grid point then corresponds to a root-to-leaf path of length $\ell = \lceil \log_4(u^2) \rceil = \lceil \log_2 u \rceil$, on a $u \times u$ grid. Since all the paths in this trie are of the same length, we do not need to store information on the children of the nodes of depth ℓ.

Figure 4 shows how our two predicates adv and won can be regarded as two-dimensional grids (as done with qdags). We also show how those grids are represented as quadtrees, which in turn are seen as cardinal trees of $\sigma = 4$ children. Their final concrete representation, as k^2-trees, is just a sequence of bits.

The space of this representation is, in the worst case, $4 \log_2 u$ bits per point, which is twice the $2 \log_2 u$ bits needed by a representation as pairs of coordinates. When the points have some regularity, like clustering, the space decreases, as shown in the figure for relation won. Within this space, the quadtree can efficiently search for points.

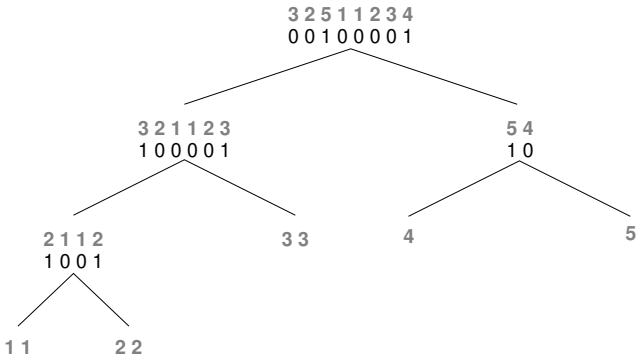

Figure 5 The wavelet tree of $S = 32511234$. Each node v shows in gray the string S_v it represents (but does not store) and below it the bitvector B_v it stores.

3.4 Wavelet trees

A *wavelet tree* [24, 37] represents a sequence $S[1..n]$ over an alphabet $[1..\sigma]$ using $n \log_2 \sigma + o(n \log \sigma)$ bits, so that several interesting queries can be answered, including the following ones in $O(\log \sigma)$ time:
$\boldsymbol{access(S, i)}$ returns $S[i]$.
$\boldsymbol{rank_a(S, i)}$ is the number of symbols equal to $a \in [1..\sigma]$ in $S[1..i]$.
$\boldsymbol{select_a(S, j)}$ is the position of the jth occurrence of $a \in [1..\sigma]$ in S.

The wavelet tree is a balanced binary tree with σ leaves, where each node handles a range of the alphabet; the root represents $[1..\sigma]$ and each leaf represents one symbol. If an internal node v represents range $[a..b]$, then its left child represents $[a..m]$ and its right child represents $[m+1..b]$, with $m = \lfloor (a+b)/2 \rfloor$. The node v represents, virtually, the subsequence S_v of S with symbols in $[a..b]$, but it only stores a bitvector B_v of length $|S_v|$, where $B_v[i] = 0$ if $S_v[i]$ belongs to the left child of v, and $B_v[i] = 1$ otherwise. Figure 5 shows an example.

The wavelet tree has height $\lceil \log_2 \sigma \rceil$. At each level, it stores exactly n bits because each position of S is in exactly one node at that level. By representing those bitvectors so that they answer *rank* and *select* in constant time (Section 3.1), the space is $n + o(n)$ bits per level and $n \log_2 \sigma + o(n \log \sigma)$ overall. Note that a plain representation of S requires $n \log_2 \sigma$ bits, and it can be less if we use compressed bitvectors or give the tree a Huffman shape.

It is not hard to see how to support the basic operations in $O(\log \sigma)$ time, with a top-down or bottom-up traversal on the wavelet tree. For example, to compute $S[i]$, we start with v being the root. If $B_v[i] = 0$, we move to its left child with $i := rank_0(B_v, i)$, otherwise we move to its right child with $i := rank_1(B_v, i)$. When we arrive at a leaf, its symbol is $S[i]$. Wavelet trees can perform more complex operations on S; we will mention them as needed.

3.5 The FM-Index

Rather than describing the general FM-Index [14, 16], which belongs to the realm of text compression and searching, we show the ideas that adapt to our particular case of interest. Consider a set of n distinct strings of length ℓ, $S_i[1..\ell]$ for $1 \le i \le n$. Sort them lexicographically and write them one per row. The last column of symbols, read downwards, is called L_ℓ.

Now take the last symbol of each S_i and put it in front of the first symbol, that is, S_i becomes $S_i[\ell] \cdot S_i[1..\ell-1]$. Stably re-sort the strings and call $L_{\ell-1}$ the last column. Continue

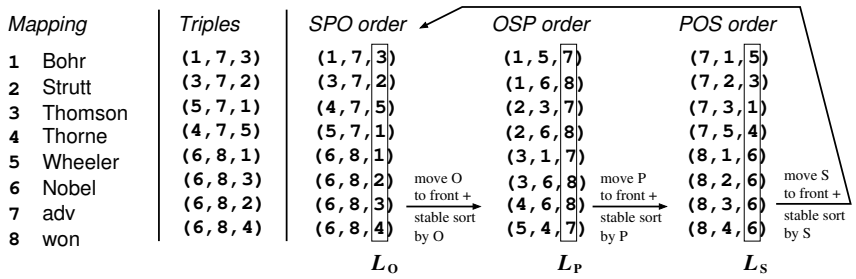

Figure 6 On the left, a mapping the nodes and labels of Figure 1 to integers. Right to it, the resulting table of triples. On the right, the three reorderings from which the columns $L_{\rm O}$, $L_{\rm P}$, and $L_{\rm S}$ are obtained.

with this process until obtaining all the strings L_j, for $1 \leq j \leq \ell$.

If we consider the strings S_i as the rows of a relational table, then the strings L_j are akin to a column store, where the table is represented column-wise and the columns have some suitable order. We do not require pointers to connect the same row across different columns, because the row i' in L_{j-1} corresponding to row i in L_j turns out to be

$$i' \;=\; C_j[c] + rank_c(L_j, i),$$

where $c = L_j[i]$ and $C_j[c]$ is the number of symbols smaller than c in L_j. The same formula navigates from L_1 to L_ℓ. We can therefore extract any row S_i in time $O(\ell \log \sigma)$ by representing the strings L_j with wavelet trees (Section 3.4), from its position in any column. We can also navigate forwards, from $L_{j-1}[i']$ to $L_j[i]$, with the inverse formula

$$i \;=\; select_c(L_j, i' - C_{j-1}[c]),$$

where c is such that $C_{j-1}[c] < i' \leq C_{j-1}[c]$.

This representation, which uses basically the same $n\ell \log_2 \sigma$ bits of a plain representation of the rows S_i, allows for other interesting queries. In particular, given some substring $X[1..t]$ and a column a, we can obtain the set of all rows S_i such that $S_i[a+1..a+t] = X$, by starting from $s_{t+1} = 1$, $e_{t+1} = n$, and then, for $j = t$ down to 1, computing $c = X[j]$ and

$$\begin{aligned} s_j &= C_{a+j}[c] + rank_c(L_{a+j}, s_{j+1} - 1) + 1, \\ e_j &= C_{a+j}[c] + rank_c(L_{a+j}, e_{j+1}). \end{aligned}$$

At the end, the desired strings are those represented in the range $L_{a+1}[s_1..e_1]$. This process is called *backward search*.

Figure 6 illustrates this structure on the three-column table that results from representing the labeled graph of Figure 1. The table is represented by the resulting columns $L_{\rm O}$, $L_{\rm P}$, and $L_{\rm S}$. These three strings, represented as wavelet trees, plus the corresponding arrays C_*, form the ring data structure for graph databases. Note that Figure 5 shows the wavelet tree of $L_{\rm O}$.

4 Qdags

Qdags [40, 6] represent each d-attribute table as a d-dimensional version of the quadtrees described in Section 3.3. A multijoin query between several tables represented by such quadtrees is solved by:

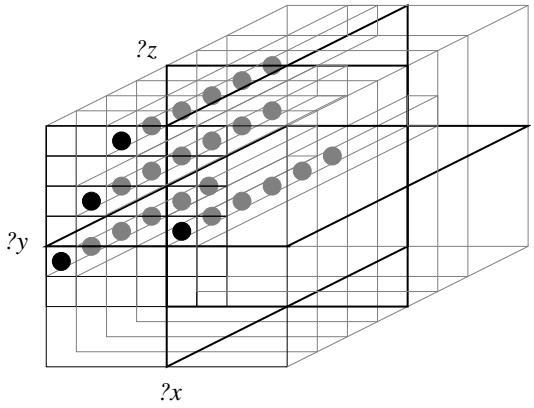

Figure 7 Extending the quadtree adv of Figure 4 to a third dimension to account for variable $?z$.

1. Converting each quadtree into one that includes the missing attributes that appear in any other joined table, all in the same order.
2. Traversing the quadtrees in synchronization to collect the points in common.
3. Writing the output of the query as a new quadtree on the increased dimension.

Qdags solve the problem of lifting the dimension of the quadtrees (step 1) at almost no extra cost. A qdag is a quadtree plus a *mapping function* that can be used to permute attributes and, most importantly, lift its dimension: for each d-dimensional point $(x_1, \ldots, x_d)$, if we raise the dimension to d', we assume that the points $(x_1, \ldots, x_d, y_{d+1}, \ldots, y_{d'})$ exist for all the possible values of $y_{d+1}, \ldots, y_{d'}$. The qdag then simulates the operations on the virtual d'-dimensional quadtree without materializing it.

To illustrate, consider the following variant of the BGP of Eq. (1)

$$(?y, \mathsf{adv}, ?x),\ (?z, \mathsf{won}, ?x),\ (?z, \mathsf{won}, ?y)$$

which has the same output in our database with the binding $?z = \mathsf{Nobel}$. Since the output will be a table with attributes $(?z, ?y, ?x)$, we need to raise the dimension of the intervening quadtrees (shown in Figure 4) to three. For the first triple pattern, $(?y, \mathsf{adv}, ?x)$, we must create the third dimension, $?z$. The corresponding qdag must represent an octree (i.e., a 3-dimensional version of a quadtree) where every point $(?y, ?x)$ in the quadtree adv is extended to every possible value of $?z$; see Figure 7. Instead of materializing that octree, the qdag combines the quadtree adv with the mapping function $(1, 2, 3, 4, 1, 2, 3, 4)$. This indicates how to traverse the 8 children of every node in the octree, reading the 4 front cubes and then the 4 back cubes; note the 4 back cubes are identical to the 4 front cubes. The quadtree is then used to support the octree navigation with just this $O(2^{d'})$ additive space and time penalty. Analogously, the qdag for the triple pattern $(?z, \mathsf{won}, ?x)$ is built from the quadtree won and the mapping function $(1, 2, 1, 2, 3, 4, 3, 4)$, and the triple pattern $(?z, \mathsf{won}, ?y)$ is built from the same quadtree won and the mapping function $(1, 1, 2, 2, 3, 3, 4, 4)$.

Optimality Note that the intersection process may work on subgrids where no output points are found, so the intersection process is not necessarily instance-optimal. It was shown, however, that there is always a database which, essentially, has points wherever the algorithm traverses in the grids, which makes this multijoin algorithm wco.

Full algebra The algorithm is compositional, since the output is also a quadtree (and hence a qdag, with the identity mapping function). This compositionality leads to including the other operations of the relational algebra. For this sake, qdags are extended to the so-called *lazy qdags (lqdags)*, which are akin to the syntax tree of the algebraic expression, through which the results flow on demand. The scheme stays wco for Boolean operations (under some constraints), but not for other operations like general selections and projections.

In practice The practical implementation uses k^d-trees to represent the quadtrees, as described in Section 3.3. The resulting quads are evaluated on a subset of Wikidata, where one two-dimensional qdag is built for each distinct predicate. The qdags use less than 5 bytes per triple, about half of the plain representation and 10–300 times less than state-of-the-art engines. Their times to solve BGPs from a query log are competitive, from much faster to much slower depending on the query types. Qdags perform better in general on lower dimensions of the output and are unbeaten on small cyclic queries.

5 The Ring

The ring [4] is a text-based compressed representation for the database triples, which can simulate the 6 tries needed by the LTJ algorithm with just a single copy of the data. The high-level idea is that each (s, p, o) triple is regarded as a *circular* string that can be navigated forwards or backwards. Any of the 6 orders can be then obtained by starting somewhere on the circle and moving in some direction.

As described in Section 3.5, the ring represents the table of triples (s, p, o) by means of the sequences L_{O}, L_{P}, and L_{S}. The key idea to simulate the LTJ algorithm of Section 2.3 is that every node of each of the 6 tries corresponds to a range in some of the L_* sequences: both represent sets of triples with some attributes already bound. We then start by associating each triple pattern in the BGP to a range in some L_* corresponding to its bound values. To find that range, we use backward search (Section 3.5). For example, for the BGP of Eq. (1) the triple $(?y, \mathsf{adv}, ?x)$ corresponds to the range $L_{\mathrm{S}}[1..4]$, whereas $(\mathsf{Nobel}, \mathsf{won}, ?x)$ and $(\mathsf{Nobel}, \mathsf{won}, ?y)$ correspond to $L_{\mathrm{O}}[5..8]$ (see Figure 6).

The LTJ algorithm is then started, binding the variables one by one. The main primitive needed to implement the intersections carried out by LTJ is: given a value k, find the leftmost child of the current trie node with value $k' \geq k$. In the ring, this corresponds to finding the smallest value $k' \geq k$ appearing in a given range $L_*[i..j]$. This can be done in logarithmic time on the wavelet tree of L_* [8, 37]. Wavelet trees also implement the needed primitives to simulate trie navigation on the sequences L_*, forwards or backwards as needed [21, 37].

For example, if we first bind $?y$, we must find the common values between $L_{\mathrm{S}}[1..4]$ and $L_{\mathrm{O}}[5..8]$, yielding 1 (Bohr), 3 (Thomson), and 4 (Thorne). Those are the instances of $?y$ that advised someone and won a Nobel prize (recall Section 2.4). Consider the branch $y = 1$. We use the backward search formula to move from $L_{\mathrm{S}}[1..4]$ to $L_{\mathrm{O}}[1..1]$ (which represents the further bounded triple pattern $(\mathsf{Bohr}, \mathsf{adv}, ?x)$), and from $L_{\mathrm{O}}[5..8]$ to $L_{\mathrm{P}}[2..2]$ (which represents $(\mathsf{Nobel}, \mathsf{won}, \mathsf{Bohr})$; this triple pattern is now totally bound). Now we bind $?x$, looking for the common values in $L_{\mathrm{O}}[1..1]$ and $L_{\mathrm{O}}[5..8]$. We here find the common value 3 (Thomson), and take that binding by moving from $L_{\mathrm{O}}[1..1]$ to $L_{\mathrm{P}}[5..5]$ (representing the triple $(\mathsf{Bohr}, \mathsf{adv}, \mathsf{Thomson})$) and from $L_{\mathrm{O}}[5..8]$ to $L_{\mathrm{P}}[6..6]$ (representing $(\mathsf{Nobel}, \mathsf{won}, \mathsf{Thomson})$). Now we have bound all the variables and the three triples represent one solution of the BGP: Bohr advised Thomson and both won the Nobel prize.

Optimality and practical performance The ring handles BGPs in wco time, since it directly simulates the LTJ algorithm. Depending on how much it compresses its wavelet trees, the ring can use about the same space of qdags, and it is also competitive in time with the state of the art (it is faster than qdags in most cases, but not on small cyclic queries). Without wavelet tree compression, it uses about 13 bytes per triple (close to the space needed by the raw graph data, and still 5–140 times smaller than the other indices) and it is on average twice as fast as the next-best competitor.

Higher dimensions The ring can be extended to higher dimensions, needing much fewer than the $d!$ copies required by classical schemes (e.g., one needs 7 rings, instead of 720 tries, for $d = 6$). This makes it usable to implement LTJ on relational tables where the classical wco indices are completely impractical. Still, the number of required rings grows as $O(2^d)$, so it soon ceases to be practical as well.

Regular path queries The ring was also used to solve RPQs [5] by just performing the classical traversal of the product automaton, with a couple of twists. First, the NFA is produced by Glushkov's algorithm [23, 39], which obtains the worst-case minimum number of states and has some regularities that are exploited (e.g., all the transition leading to a given state have the same label). Second, the wavelet trees of the sequences L_* are enhanced so as to avoid spending any time on edges of the product automaton that lead to no active NFA states, or that cycle on the automaton. The resulting algorithm, even if not using any filtration technique, is competitive with the state of the art (3 times faster than the next-best, on average), while using 3–5 times less space than all of them. More recent developments using filtration techniques become about 5 times faster than the others.

6 Now What?

Our research has demonstrated that cds can successfully implement the core of graph database engines, providing wco multijoin algorithms that are also efficient in practice, and removing all of the redundancy associated with those algorithms. As such, they can make a reality the efficient querying of the huge graph databases that are emerging.

But we have just scratched the surface of the problem. There are many issues to consider in the way, on many of which we are working. We list only some of the most prominent ones.

How to handle higher dimensions? While three dimensions (or four, in some models) suffice to describe graph databases, relational ones may have many more columns. Both qdags and the ring have time or space troubles with higher dimensions, and even if they can handle them better than current schemes, they soon become impractical. In order to provide competitive solutions for relational data, we must probably combine wco and non-wco schemes [35, 20]. Cds have demonstrated that they can provide more than the basic functionality on the data, so an interesting question is what can they support in the direction of combining both kinds of query plans.

Can we provide more functionality? A formidable challenge is to combine BGPs and RPQs in an efficient manner, as this is supported in SPARQL and other query languages. Interactive querying requires retrieving (possibly only some) results in decreasing order of relevance [52]. Providing more semantics to the nodes leads to problems like supporting similarity joins [13], spatio-temporal predicates, and so on. The concept of wco with those extended semantics is yet to be studied. Again, cds can provide novel and more efficient solutions to those problems.

How to scale to a real system? Real graph database systems are very complex, and thus it is not direct to put our performance improvements, which focus on specific subproblems, into use. Consider for example going from our BGPs and RPQs to the full SPARQL support. The fastest path is to integrate our research prototypes into an existing system. A good candidate for this is MillenniumDB [55], a full-fledged graph database system with strong algorithmic foundations and designed to plug-and-play different solutions to subproblems.

Can we support graph analytics? BGPs serve not only for *querying* graph databases, but also as building blocks to support *graph analytics* [28]. In our example graph, we could ask how likely is that the advisee of a Nobel winner also wins the prize, by *counting* the number of answers to BGP queries (rather than listing them all). Graph analytics require various sorts of summarization operations on the query results, where in addition it is acceptable to return approximate answers. It is interesting to see if cds can support counting (perhaps approximately) the number of results of queries without listing them all; some of their extended functionality can be of use. More in general, we can explore the use of cds to represent other objects that are key in analytics, like matrices. There is some preliminary work in this direction [15, 18, 22].

References

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

2 W. Ali, M. Saleem, B. Yao, A. Hogan, and A.-C. Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, 31(3):1–26, 2022.

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

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

5 D. Arroyuelo, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Time- and space-efficient regular path queries. In *Proc. 38th IEEE International Conference on Data Engineering (ICDE)*, pages 3091–3105, 2022.

6 D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):article 8, 2022.

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

8 J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19–37, 2013.

9 D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

10 N. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

11 D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

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

13 S. Ferrada, B. Bustos, and A. Hogan. Extending SPARQL with similarity joins. In *Proc. 19th International Semantic Web Conference (ISWC)*, pages 201–217, 2020.

14 P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

15 P. Ferragina, G. Manzini, T. Gagie, D. Köppl, G. Navarro, M. Striani, and F. Tosoni. Improving matrix-vector multiplication via lossless grammar-compressed matrices. *Proceedings of the VLDB Endowment*, 2022. To appear. See `www.dcc.uchile.cl/gnavarro/ps/pvldb22.pdf`.

16 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

17 N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proc. SIGMOD International Conference on Management of Data*, pages 1433–1445, 2018.

18 A. P. Francisco, T. Gagie, D. Köppl, S. Ladra, and G. Navarro. Graph compression for adjacency-matrix multiplication. *SN Computer Science*, 3:article 193, 2022.

19 E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

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

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

22 F. Geerts, T. Muñoz, C. Riveros, J. van den Bussche, and D. Vrgoc. Matrix query languages. *SIGMOD Record*, 50(3):6–19, 2021.

23 V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

24 R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

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

26 A. Hogan. *The Web of Data*. Springer, 2020.

27 A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutiérrez, S. Kirrane, J.E. Labra Gayo, R. Navigli, S. Neumaier, A.-C. Ngonga Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool Publishers, 2021.

28 A. Hogan, J. L. Reutter, and A. Soto. In-database graph analytics with recursive SPARQL. In *Proc. 19th International Semantic Web Conference (ISWC)*, pages 511–528, 2020.

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

30 O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *Proc. 20th International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

31 M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems*, 41(4):22, 2016.

32 A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proc. International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 177–194, 2012.

33 S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*, pages 376–394, 2018.

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

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

36 J. J. Miller. Graph database applications and concepts with Neo4j. In *Proc. Southern Association for Information Systems Conference*, pages 141–147, 2013.

37 G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

38 G. Navarro. *Compact Data Structures — A practical approach*. Cambridge Univ. Press, 2016.

39 G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge Univ. Press, 2002.

40 G. Navarro, J. Reutter, and J. Rojas-Ledesma. Optimal joins using compact data structures. In *Proc. 23rd International Conference on Database Theory (ICDT)*, pages 21:1–21:21, 2020.

41 T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19:91–113, 2010.

42 H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th Symposium on Principles of Database Systems (PODS)*, pages 111–124, 2018.

43 H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proc. 31st Symposium on Principles of Database Systems (PODS)*, pages 37–48, 2012.

44 H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

45 D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 2:1–2:8, 2015.

46 V.-Q. Nguyen and K. Kim. Efficient regular path query evaluation by splitting with unit-subquery cost matrix. *IEICE Transactions on Information and Systems*, 100-D(10):2648–2652, 2017.

47 R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

48 I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2nd edition, 2015.

49 H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

50 H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

51 B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata®RDF Graph Database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.

52 N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment*, 13(9):1582–1597, 2020.

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

54 D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

55 D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. MillenniumDB: A persistent, open-source, graph database. *CoRR*, abs/2111.01540, 2021.

56 X. Wang, J. Wang, and X. Zhang. Efficient distributed regular path queries on RDF graphs using partial evaluation. In *Proc. International Conference on Information and Knowledge Management (CIKM)*, pages 1933–1936, 2016.

57 M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*, pages 82–94, 1981.