Boosting Graph Joins and Matrix Multiplications in Little Space

Diego Arroyuelo*
IMFD & DCC, Pontificia Universidad Católica de Chile
Santiago, Chile
diego.arroyuelo@uc.cl

José M. Cazorla*
IMFD & DI, Universidad Técnica Federico Santa María
Valparaíso, Chile
jcazorla@usm.cl

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

Abstract

Qdags are extremely succinct representations of relations that have been used to implement worst-case-optimal joins in graph databases, as well as regular path queries and sparse matrix multiplications. While they are very fast on joins over few attributes, their performance sharply degrades over more attributes. In this paper we introduce a so-called prejoining technique that builds joins over fewer attributes and adds their outputs as new tables in the main join. We show that prejoining boosts qdags by orders of magnitude, letting them outperform a number of representative systems over a range of join shapes and on real Wikidata queries. Applied to sparse Boolean matrix multiplication, prejoining boils down to a compressed implementation of the well-known Shoor's algorithm, which we show to sharply outperform classic recursive multiplication, the only one implemented so far on little space.

CCS Concepts

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

Keywords

Worst-case-optimal joins, Graph databases, Compressed data structures, Matrix multiplication, Prejoining, Qdags

ACM Reference Format:
Diego Arroyuelo, José M. Cazorla, and Gonzalo Navarro. 2026. Boosting Graph Joins and Matrix Multiplications in Little Space. In *9th Joint Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3810460.3812779

1 Introduction

Worst-case-optimal (wco) join algorithms [7] are a relatively recent and revolutionary way to handle multijoins in relational databases, ensuring that the time complexity is proportional, up to logarithmic and data-independent factors, to the maximum possible size of the output over tables of the same size of the input ones. After proving that no pairwise join plan can be wco [7], various wco join algorithms were developed over time [12, 15–18, 22].

Wco algorithms are particularly relevant on graph databases [3], where Basic Graph Patterns (BGPs) are at the core of most query language standards like SPARQL [19] or GQL [11]. In the simplest RDF graph database model [14], the database is a labeled directed graph, or a set of triples (source node, edge label, target node). BGPs are graph patterns with constants and variables to be matched to the database graph in every possible way. They can also be seen as multijoins (and selections) over the corresponding 3-attribute relation. With this second view, wco algorithms have been extended to solve BGPs [10]. Typical BGPs tend to translate into multijoins of many tables, with complex and often cyclic shapes, and this is where wco algorithms sharply outperform classical pairwise joins.

Different wco algorithms require different indexes, or data structures built on the graph database, in order to efficiently handle queries. While the wco concept permits building them on the fly at query time, this is totally impractical on large graphs. For example, the Wikidata graph has over 14 billion edges, which requires several minutes to scan even on an SSD. On the other hand, indexes that support wco algorithms tend to use too much space per edge; for example the classic wco systems we compare with in this paper use over 100 bytes per edge. Thus indexing the Wikidata graph with them would require over 1.3 TB.

While secondary storage is an option, it is orders of magnitude slower than main memory, which slows down queries. This raised the interest in compact indexes to support wco algorithms [4, 6]. In this paper we focus on qdags [6], which store *and* index the graph using less than 5 bytes per edge (well below the 12 bytes it takes to store each triple as three 32-bit integers!) and could index the Wikidata in 64 GB, thus fitting in the RAM of a commodity server.

Qdags perform very competitively on small BGPs. However, their time complexity contains a factor of the form $O(2^d)$, where d is the number of variables in the BGP. As a consequence, actual times quickly grow as d reaches 4 or more. This weakness, which severely limits qdags' applicability, owes to their need to traverse an output relation of dimension d, which they represent as a 2^d-ary tree.

In this paper we significantly broaden the applicability of qdags by exploiting the fact that they perform much faster on lower-dimensional joins. The main idea, which we call *prejoining*, is to choose a small subset of $d' < d$ attributes, project the query to those attributes, and join the projections (with the original qdag algorithm [6]). The resulting table is then *included* in the original query, as one more table to join. The expanded query is then finally solved, again with the original qdag join algorithm. While this seems to just increase the burden, it retains worst-case optimality while significantly improving times in practice: the first join works

*All authors contributed equally to this research.

GRADES-NDA '26, Bengaluru, India

ACM ISBN 979-8-4007-2695-8/2026/05
https://doi.org/10.1145/3810460.3812779

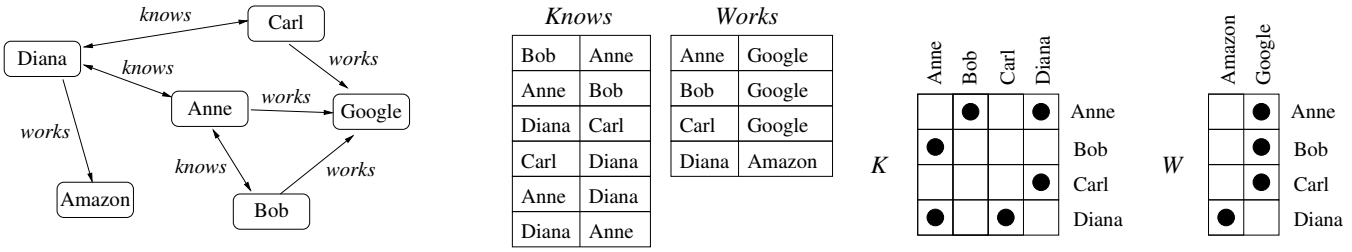

Figure 1: A toy graph database, its view as binary relations, and as two-dimensional grids.

on a lower dimension (which multiplies its time by $O(2^{d'})$ instead of $O(2^d)$), and the second contains a new table that severely limits the traversal of the output relation. One choice, for example, is to prejoin on the set of the non-lonely attributes (i.e., those that participate in the join), but we explore several other strategies to apply prejoining, including various prejoining steps.

Our experiments evaluate several prejoining strategies. We use two-dimensional qdags to represent a graph as a set of binary relations, one per label ℓ: $R_\ell = \{(s,t), s \xrightarrow{\ell} t\}$. The output of the BGP is also a (d-dimensional) qdag. Our experiments show that, by maintaining both projections of every binary relation precomputed (as a 1-dimensional qdag), our technique mildly increases the space to about 6 bytes per edge, in exchange for sharply outperforming the original join algorithm [6] and making qdags the fastest choice by far in virtually all synthetic query shapes we tried. This significantly widens the set of query patterns where qdags are practical: our experiments on real user queries on Wikidata show that prejoining makes qdags the fastest index within their tiny space usage.

We then turn our attention to sparse Boolean matrix multiplication. A second key component in graph query languages are Regular Path Queries (RPQs), which seek the source and target nodes of paths whose sequence of labels belong to the language of a given regular expression. It was shown [5] that RPQs can be reduced to an expression on Boolean matrices involving sum, product, and transitive closure, where the matrices are precisely the binary relations R_ℓ. While they [5] implement a standard recursive algorithm to multiply matrices over qdags, we can regard those as joins (plus a final projection) and apply our technique. The resulting algorithm turns out to be a compressed-space implementation of Shoor's algorithm [20], well-known for its effectivenes in sparse matrix multiplication. Indeed, we show that our technique outperforms the recursive qdag matrix multiplication algorithm by up to an order of magnitude. This has applications to sparse matrix multiplication (in little space) well beyond that of solving RPQs.

2 Wco Joins and Graph Databases

2.1 Worst-case-optimal (wco) joins

Let us denote $Q := R_1 \bowtie \cdots \bowtie R_m$ the (natural) multijoin between relational tables $R_1, \ldots, R_m$. An algorithm to compute Q is said to be *worst-case-optimal (wco)* [7] if it takes time $\tilde{O}(Q^*)$, where

$$Q^* = \max\{|R'_1 \bowtie \cdots \bowtie R'_m|,\ |R'_i| \le |R_i| \text{ for all } i\}$$

and $\tilde{O}$ hides factors polylogarithmic or independent on any $|R_i|$. That is, except for polylogarithmically-bounded or data-size-agnostic factors (like the query size m, or the number of attributes of the tables), the time is bounded by the maximum possible size of the output on some tables R'_i of at most the same size of R_i.

2.2 Graph databases

A *graph database* is a labeled graph G. One can regard the edges $s \xrightarrow{\ell} t$ of G as triples (s, ℓ, t), yet we will use another customary formalism that sees G as a set of tables, one per label:[1]

$$R_\ell := \{(s,t), s \xrightarrow{\ell} t\}.$$

The *size* of G, denoted $N := |G|$, is the number of edges in G. Let $U := \{s, t,\ \exists \ell, s \xrightarrow{\ell} t\}$ be the set of node identifiers in G, and let V be a set of *variables*, disjoint from U. A *Basic Graph Pattern (BGP)*, the main construction to query graph databases, is a set Q of *triple patterns* of the form $x \xrightarrow{\ell} y$, where $x, y \in U \cup V$ and ℓ is a label. A *solution* for Q is an assignment $f : V \to U$ so that, if we replace every $x \in V$ that occurs in Q by $f(x)$, then all the resulting edges occur in G. *Solving* a BGP Q means computing

$$Q(G) = \{f : V \to U,\ f \text{ is a solution for } Q\}.$$

Analogously to the wco concept for relational tables [10], we say that an algorithm that solves a BGP Q is wco if it takes time $\tilde{O}(Q^*)$, where $Q^* = \max\{|Q(G')|,\ |G'| \le |G|\}$. This extension of the wco concept is sound, because solving Q is analogous to computing the multijoin of the tables R_ℓ corresponding to the triple patterns $x \xrightarrow{\ell} y$, apart from some equality selections (when x or y are constant) and attribute renaming. Each solution f corresponds to a row in the corresponding relational table Q.

Figure 1 shows a toy example of a graph with two labels, $\ell =$ *knows* and $\ell =$ *works*, the corresponding tables R_ℓ, and their view as grids (see next).

3 Qdags

A 2^d-*tree*[2] is a spatial representation of a d-dimensional grid of side u, defined as follows. Assume that u is a power of two for simplicity. If $u = 1$, then the tree is a single cell, which is empty or full. Otherwise, the tree has a root with 2^d children, each representing one of the subgrids that are obtained by cutting the main grid by half across the d dimensions. The subgrids that are empty become leaves; the nonempty ones are, recursively, 2^d-trees of side $u/2$. The height of a nonempty 2^d-tree is exactly $\log_2 u$.

[1] We will consequently restrict BGPs to have constant labels, but this still captures most practical queries.

[2] Better known names are quadtree for $d = 2$ and octree for $d = 3$.

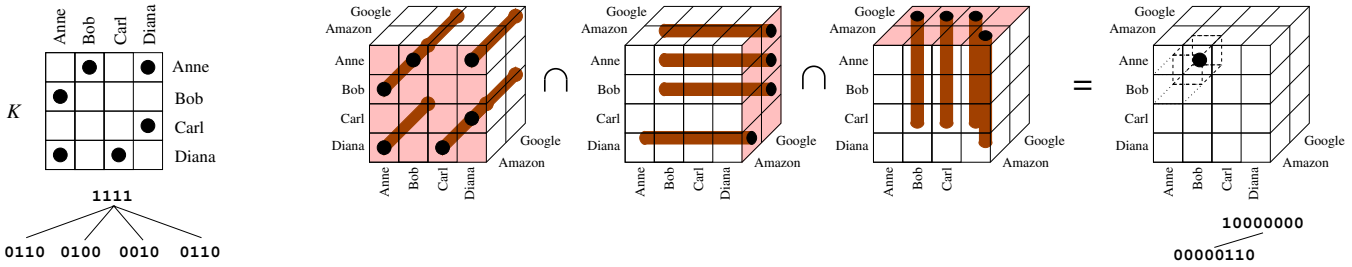

Figure 2: The grid K of table R_{knows} and the qdag resolution for the BGP $\{(x, knows, y), (x, works, z), (y, works, z)\}$. Below, the quadtree and octree representations of K and of the output, respectively

A *compressed* 2^d*-tree* [8] is a representation of a 2^d-tree using just 2^d bits per internal node (and zero per leaf node), which can navigate the tree in constant time per operation. Compressed 2^d-trees can represent d-attribute relations once we map the domain of their attributes to an integer space $[1, u]$, and regard each tuple of Q as a full cell in the d-dimensional grid. This structure is built in time $O((2^d + dN) \log u)$, by consecutively reordering the tuples.

Qdags [6] solve multijoins $Q := R_1 \bowtie \cdots \bowtie R_m$ in wco time as follows. Let $\mathcal{A}_i$ be the set of attributes of R_i and $d_i = |\mathcal{A}_i|$. The set of attributes of Q is then $\mathcal{A} = \cup_i \mathcal{A}_i$, and let $d = |\mathcal{A}|$. Assuming each table R_i is represented as a compressed 2^{d_i}-tree, qdags simulate the *extension* of each R_i to contain all the attributes $\mathcal{A}$: for each tuple $X \in R_i$, we add *all the combinations* of tuples Y on attributes $\mathcal{A} \setminus \mathcal{A}_i$, and form an extended table $\hat{R}_i$ with all the concatenated tuples $X : Y$. The qdag is a data structure that represents the d-dimensional grid Q_i of $\hat{R}_i$ in implicit form, using just $O(2^d)$ *additive* extra space and simulating the traversal of the 2^d-tree of $\hat{R}_i$ in constant time.

Once all the tables R_i are (virtually) extended to $\hat{R}_i$ and qdags for their grids Q_i are obtained, the output is simply the *intersection* of all the grids: by backtracking on the (simulated) 2^d-trees of every Q_i, stopping when the subgrid of any Q_i is empty or when we reach full leaves, we generate the (materialized) 2^d-tree of $Q_1 \cap \cdots \cap Q_m$. This is the (compressed) representation of Q.

Figure 2 continues the example of Figure 1, showing how the BGP equivalent to $K(x, y) \bowtie W(x, z) \bowtie W(y, z)$ is solved with the qdag mechanism (we plot only one of the two outputs for legibility).

Although we can backtrack over large areas of the d-dimensional output grid without finding points of Q, it is shown [6] that the process is wco, taking time $O(Q^* \cdot 2^d m \log u)$. Qdags are remarkable for not requiring space exponential in d for the indexes, but there is instead a 2^d factor penalty in their query time. On experiments where quadtrees (i.e., 2^2-trees) are used to represent the tables R_ℓ in graph databases (particularly, a subset of the Wikidata graph), the index uses 4.75 bytes per edge (classic wco indexes use over 100) and outperforms most systems by far, matching the best ones, on BGPs with a few variables (up to $d = 3$). Yet, as expected from the theory, time quickly degrades with BGPs of $d \geq 4$ variables.

4 Boosting Wco Joins on Qdags by Prejoining

Consider the following BGP, called a "3-star" (we write it as a join and use more friendly names for tables R_ℓ):

$$Q(a, b, c, d) := R(a, b) \bowtie S(a, c) \bowtie T(a, d).$$

Solving Q with qdags involves, first, extending tables R, S, and T to $\hat{R}(a, b, c, d)$, $\hat{S}(a, b, c, d)$, and $\hat{T}(a, b, c, d)$, which is done in just $O(2^d)$ time ($d = 4$ in our case). Now, the backtracking on the grids of $\hat{R}$, $\hat{S}$, and $\hat{T}$ to compute their intersection explores a space of u^4 cells (where $u := |U|$). At each point in the backtracking we descend by each of the $2^d = 16$ children of the three qdags, stopping when we find any leaf. We may explore a lot of (a, b, c, d) range combinations just to find out that there is no common value of a in $\hat{R}$, $\hat{S}$, and $\hat{T}$.

Instead, consider the following. Compute the projections of R, S, and T onto the common attribute a, represent them as compressed 2^1-trees, and compute using the qdag join algorithm

$$I(a) := \pi_a(R) \bowtie \pi_a(S) \bowtie \pi_a(T).$$

Now, instead of the original query, solve

$$Q(a, b, c, d) := I(a) \bowtie R(a, b) \bowtie S(a, c) \bowtie T(a, d).$$

While this seems to add an additional burden, it has very good chances of improving the performance. The computation of I is a join of dimension $d = 1$, which is very fast with qdags. The final join now intersects also $\hat{I}(a, b, c, d)$, which will allow the backtrack to proceed only towards the values of a that do occur in the intersection of $\hat{R}$, $\hat{S}$, and $\hat{T}$. We now formalize the idea.

4.1 The prejoining step

The key component of our strategy is dubbed a *prejoining step*.

Definition 4.1. Given a join $Q(\mathcal{A}) := \bowtie_i R_i(\mathcal{A}_i)$, a *prejoining step* on a subset $\mathcal{A}' \subset \mathcal{A}$ of attributes performs the following steps:

(a) project all R_i (having attributes in $\mathcal{A}'$) onto $\mathcal{A}' \cap \mathcal{A}_i$;
(b) join the resulting tables into a new table

$$I(\mathcal{A}') := \bowtie_i \pi_{\mathcal{A}' \cap \mathcal{A}_i}(R_i)$$

(on the R_is with $\mathcal{A}' \cap \mathcal{A}_i \neq \emptyset$);
(c) add I to the set of tables to join, thereby computing

$$Q(\mathcal{A}) := I \bowtie (\bowtie_i R_i)$$

(omitting every R_is for which $\mathcal{A}_i \subseteq \mathcal{A}'$). □

The prejoining steps can be applied in many ways, several times along the join, and can be a powerful mechanism to define join strategies. We now prove that adding a prejoining step retains worst-case optimality in general.

Theorem 4.2. *Any prejoining step retains worst-case optimality.*

Proof. The AGM bound [7] for $Q = R_1 \bowtie \cdots R_m$ is the least value $Q^* = \Pi_i |R_i|^{x_i}$, where $x_1, \ldots, x_m$ are the weights of a *fractional edge cover* of the join: Consider a hypergraph $\mathcal{G}(\mathcal{A}, \mathcal{E})$ whose nodes are the attributes $\mathcal{A}$ and each joined relation $R_i(\mathcal{A}_i)$ is a hyperedge of $\mathcal{E}$, connecting the attributes $\mathcal{A}_i$. A fractional edge cover assigns a fractional weight to each hyperedge so that the sum of the weights of all hyperedges inciding on each attribute $a \in \mathcal{A}$ is at least 1.

Now let $x_1, \ldots, x_m$ be the optimal values to obtain Q^*, and consider our prejoining step that builds table $I(\mathcal{A}')$ on the selected variables $\mathcal{A}' \subset \mathcal{A}$. We first project on the set $\mathcal{A}'$. This takes time proportional to $\sum |R_i|$, which is always wco. We then compute $I(\mathcal{A}')$ with the first join. To bound I^*, define a new hypergraph $\mathcal{G}'(\mathcal{A}', \mathcal{E}')$ where we remove from $\mathcal{G}$ all the nodes in $\mathcal{A} \setminus \mathcal{A}'$, and retain the same hyperedges of $\mathcal{E}$ on the remaining nodes. Clearly any fractional edge cover $x_1, \ldots, x_m$ of $\mathcal{G}$ is also valid on $\mathcal{G}'$, because the sum of weights of hyperedges inciding on the attributes $a' \in \mathcal{A}'$ have not changed. Therefore, the AGM bound to compute I is $I^* \leq Q^*$ (indeed, there may very well be a fractional edge cover $x'_1, \ldots, x'_m$ yielding $I^* < Q^*$).

Now consider the final join, which now incorporates table $I(\mathcal{A}')$. This corresponds to a new hypergraph $\mathcal{G}''(\mathcal{A}, \mathcal{E} \cup \{\mathcal{A}'\})$, where we have added the hyperedge corresponding to $I(\mathcal{A}')$. Note that we can simply assign zero weight to this new hyperedge, and the fractional edge cover $x_1, \ldots, x_m$ for $\mathcal{G}$ can be turned into a valid fractional edge cover $x_1, \ldots, x_m, 0$ for $\mathcal{G}''$. The AGM bound for this final join is then at most Q^* (again, $\mathcal{G}''$ could have a better fractional edge cover leading to a smaller bound).

Overall, the prejoining step leads to an overall join complexity in $\tilde{O}(Q^*)$ if the two joins it involves are done worst-case-optimally. □

A burden of prejoining is the need to compute the projections $\pi_{\mathcal{A}' \cap \mathcal{A}_i}(R_i)$, as they take time proportional to the database size (see next). While still wco, this may be in practice too expensive in the typical case where the output is not very large and indexes avoid traversing the tables in full. Exploiting the fact that our qdag-based indexing uses very little space, we will precompute and store the two projections of each table $R_\ell(s, t)$: $\pi_s(R_\ell)$ and $\pi_t(R_\ell)$, as compressed 2^1-trees. Our experiments show that this increases the space only mildly.

4.2 Implementation of projections

Projections are not included in the original qdag functionality [6]. We implemented them over the compressed 2^d-tree so as to produce the projected $2^{d'}$-tree without decompressing the original one. The compressed format [8] stores the tree nodes levelwise. For each node, it stores just the 2^d bits that describe which children contains full cells (1) and which do not (0). It turns out that, with the help of sublinear-space additional data structures, such a representation can be efficiently traversed in constant time per operation.

We traverse the original 2^d-tree T levelwise, and generate the set of the same-level nodes of the $2^{d'}$-tree T'. From the 2^d-bit signature of each node v of T, corresponding to node v' in T', we compute the corresponding children of v' and set the corresponding 1s in its $2^{d'}$-bit signature. Each (nonempty) child of v is then associated with the corresponding child of v'. When we complete the level, we collect all the next-level of nodes of T associated with nodes of T', and start the process of the next level.

4.3 Prejoining strategies

Though prejoining steps retain worst-case optimality, they can increase the original qdag join cost: consider tables $X(a, b) = \{(1, 1), (2, i),\ 1 \leq i \leq N\}$ and $Y(a, b) = \{(1, 1), (3, i),\ 1 \leq i \leq N\}$. The qdag join algorithm will not descend by $a = 2$ nor $a = 3$, as the branch in X or in Y will be empty, and thus will finish in $O(1)$ time. If, instead, we prejoin on $\{b\}$, we will generate table $I(b) = \{i,\ 1 \leq i \leq N\}$, taking time $O(N)$.

This example highlights the importance of choosing good prejoining strategies. We now describe strategies that have worked best in practice, as shown later in the experiments.

One-shot prejoining. A simple strategy is to do only one prejoining step, defining $\mathcal{A}'$ as the set of non-lonely attributes (i.e., occurring in more than one table; lonely attributes do not participate in the join). In our 3-star example, b, c, and d are lonely, so $\mathcal{A}' = \{a\}$ and one-shot prejoining works just a described.

Leapfrog prejoining. A promising strategy is to perform several prejoining steps, choosing $\mathcal{A}'$ to be increasing subsets of the attributes. This loosely simulates Leapfrog TrieJoin [22], which is the most popular wco algorithm. To illustrate leapfrog prejoining, consider the following BGP, called a "square":

$$Q(a, b, c, d) := R(a, b) \bowtie S(b, c) \bowtie T(c, d) \bowtie U(d, a).$$

Leapfrog prejoining on this case performs three prejoining steps:

$$\begin{aligned} I_1(a) &:= \pi_a(R) \bowtie \pi_a(U), \\ I_2(a, b) &:= I_1 \bowtie R \bowtie \pi_b(S), \\ I_3(a, b, c) &:= I_2 \bowtie S \bowtie \pi_c(T), \\ Q(a, b, c, d) &:= I_3 \bowtie T \bowtie U. \end{aligned}$$

Note those are all valid prejoining steps according to Def. 4.1. For example, after computing I_1, the final join becomes $Q := I_1 \bowtie R \bowtie S \bowtie T \bowtie U$. Now, for the second step $I_2(a, b)$, the involved tables are I_1, R, and S, and the final query becomes $Q := I_2 \bowtie I_1 \bowtie R \bowtie S \bowtie T \bowtie U$, but we remove I_1 and R because their attributes are contained in $\mathcal{A}' = \{a, b\}$. Similarly, I_2 and S are removed after including I_3 in the join, so we finish with $Q := I_3 \bowtie T \bowtie U$.

A good feature of leapfrog prejoining is that it can reduce the cost of the qdag algorithm even in cases where there are no lonely attributes (as in the square example).

Kernel prejoining. This strategy complements one-shot prejoining with additional filtration of the involved relations, as follows:

(1) A first prejoin $I(\mathcal{A}') := \bowtie_i \pi_{\mathcal{A}' \cap \mathcal{A}_i}(R_i)$ is made on the non-lonely attributes $\mathcal{A}'$.
(2) We *reduce* each table $R_i(\mathcal{A}_i)$ involved in the join, as follows:
$$R'_i(\mathcal{A}_i) := R_i \ltimes I = R_i \bowtie \pi_{\mathcal{A}' \cap \mathcal{A}_i}(I).$$
(omitting every R'_i such that $\mathcal{A}_i \subseteq \mathcal{A}'$).
(3) We compute the final join,
$$Q(\mathcal{A}) := I \bowtie (\bowtie_i R'_i).$$
(omitting R'_i as before, and also I if $\mathcal{A}' \subset \mathcal{A}_i$ for some R_i).

For example, applied on the 3-star, this strategy first computes $I(a) := \pi_a(R) \bowtie \pi_a(S) \bowtie \pi_a(T)$, where in our qdags the projections are precomputed. It then reduces $R' := R \bowtie I$, $S' := S \bowtie I$, and $T' := T \bowtie I$. Finally, it returns $Q := R' \bowtie S' \bowtie T'$.

It is not hard to see that all (1)–(3) are valid per Def. 4.1. The name of the strategy owes to the general idea of kernelization: reducing instances to a kernel (in our case, the entries of R_i having intersection with I) that is actually hard to deal with, but hopefully much smaller.

Weak kernel prejoining. Note that the kernel strategy makes sense only when there are lonely attributes. The following weaker version can be used even when this is not the case:

(1) define again $\mathcal{A}'$ as the set of non-lonely attributes of Q;
(2) for each $a \in \mathcal{A}'$, compute $I_a = \bowtie_i \pi_a(R_i)$;
(3) for each R_i, define

$$R'_i(\mathcal{A}_i) := R_i \bowtie (\bowtie_{a \in \mathcal{A}' \cap \mathcal{A}_i} I_a);$$

(4) compute $Q := \bowtie_i R'_i$.

That is, the third step filters, separately, each non-lonely attribute of each relation R_i to those values that will occur in the final join. In the case of qdags over binary relations, the second step does not need to perform the projections because we have them all precomputed. Further, the third step performs at most a three-way join per R_i. For example, in the case of the square, weak kernel prejoining works as follows: it first computes $I_a := \pi_a(R) \bowtie \pi_a(U)$, $I_b := \pi_b(R) \bowtie \pi_b(S)$, $I_c := \pi_c(S) \bowtie \pi_c(T)$, and $I_d := \pi_d(T) \bowtie \pi_d(U)$, where all the projections are precomputed. It then computes $R' := R \bowtie I_a \bowtie I_b$, $S' := S \bowtie I_b \bowtie I_c$, $T' := T \bowtie I_c \bowtie I_d$, and $U' := U \bowtie I_d \bowtie I_a$. Finally, it returns $Q := R' \bowtie S' \bowtie T' \bowtie U'$.

5 Boosting Boolean Matrix Multiplication

Consider two $u \times u$ sparse Boolean matrices, A and B. Their representations as compressed quadtrees takes space that is proportional to the number of nonzero entries, and further this space is compressed [5]. If one regards the product $C := A \times B$, it turns out that cell (a, c) is nonzero in C iff there is b such that (a, b) is nonzero in A and (b, c) is nonzero in B. Regarded as two-attribute tables $A(a, b)$ and $B(b, c)$, it follows that $C(a, c) = \pi_{a,c}(A(a, b) \bowtie B(b, c))$.

Assume we apply our prejoining step over this join. The non-lonely variable is b, so we first compute $I(b) = \pi_b(A) \bowtie \pi_b(B)$, and then compute $C(a, c) = \pi_{a,c}(I(b) \bowtie A(a, b) \bowtie B(b, c))$. This corresponds verbatim to Shoor's algorithm [20]: it first intersects the columns of A with the rows of B. For every b in the intersection, it fills the output cells at the cartesian product of all the as such that (a, b) is nonzero in A, and all the cs such that (b, c) is nonzero in B. This algorithm was implemented in a sparse non-compressed baseline [5] and found to be extremely fast, whereas the only matrix multiplication algorithm implemented on compressed quadtrees was the standard recursive one. Our technique implements Schoor's algorithm on compressed quadtrees (plus their two projections).

In the general case, we have a product $C := A_1 \times \cdots \times A_m$. Let $A_i(a_i, a_{i+1})$, then the corresponding join is $C(a_1, a_{m+1}) = \pi_{a_1, a_{m+1}}(A_1 \bowtie \cdots \bowtie A_m)$. The lonely attributes are a_1 and a_{m+1}. We will experiment with *one-shot* (where $\mathcal{A}' = \{a_2, \ldots, a_m\}$), *leapfrog*, and *weak kernel* prejoining.

6 Experimental Results

The experiments were conducted on a server equipped with two Intel Xeon Silver 4110 processors running at 2.10 GHz, providing a total of 32 physical cores and 735 GB of RAM. The implementations were developed in C++11 and compiled using the GNU Compiler Collection (g++) with optimization level -O3. We measure user times with a timeout of 600 seconds, and output all the results produced by the queries (up to the timeout).

Datasets and queries. For the experimental evaluation, we use the Wikidata Graph Pattern Benchmark (WGPB) [10], which consists of a Wikidata subgraph with 81,426,573 edges, featuring 51,999,296 nodes and 2,101 distinct predicates. The benchmark defines 17 query patterns of different sizes and topologies, including acyclic and cyclic queries (see Figure 3): paths (P2–4), 2-stars (T2, Ti2), 3-stars (T3, Ti3, J3), 4-stars (T4, Ti4, J4), triangles (Tr1–2), and squares (S1–4). Each pattern is instantiated with 50 randomly generated queries involving real Wikidata predicates, ensuring that query results are non-empty. Among those patterns, P2, P3 and P4 are also used in our matrix multiplication experiments.

Additionally, we evaluated the scalability of our proposal using the full Wikidata graph [13], which contains 958,844,164 edges, 348,945,080 nodes and 5,419 distinct predicates. For this large-scale dataset, we used queries extracted from a public Wikidata query log [13], which were filtered and processed to generate two specific workloads: one formed by 195 BGPs with up to five triple patterns with constant predicates and variables in the subject and object positions, and another with 48 queries with the P2 pattern (see Figure 3), used for the matrix multiplication experiments.

Systems compared. We compared our index with several data structures and systems representative of the state of the art. First, we included compact structures based on Ring [4], specifically the Ring and C-Ring (a compressed version) variants, as well as its adaptation for fixed predicates (RingP). These structures solve BGPs with wco algorithms. Additionally, we compared five reference systems widely used in the literature: Apache Jena, Jena LTJ [10] (non-wco and wco, respectively; this pair highlights the difference between such strategies under a similar implementation), Blazegraph [21] (non-wco, the current backend of Wikidata service), Virtuoso [9] (non-wco, widely used to support SPARQL in real systems), and EmptyHeaded [1] (a prototype supporting beyond-wco strategies).

We did not run those five systems on our own server, but instead relied on previously reported experimental results [6] run on an Intel Xeon E5-2630 processor at 2.30 GHz, with 6 cores and 96 GB of RAM. To compare the results on both platforms more fairly, we applied a normalization factor of 1.109 to the execution times of those five systems. This factor was estimated as the ratio of the execution times of the Qdag algorithm measured on both servers. This procedure provides a reasonable comparative reference, although hardware differences might still (slightly) influence the final results. As we will see, we obtain too large differences for those small factors to have any impact on the conclusions.

For matrix multiplications, we compared with the original recursive algorithm running on qdags [5], which is called K2tree in reference to the data structure underlying qdags. This is run on our current machine.

6.1 Wco joins

Our first experiment shows how prejoining improves upon the original qdag algorithm. Table 1 shows the space used by the original

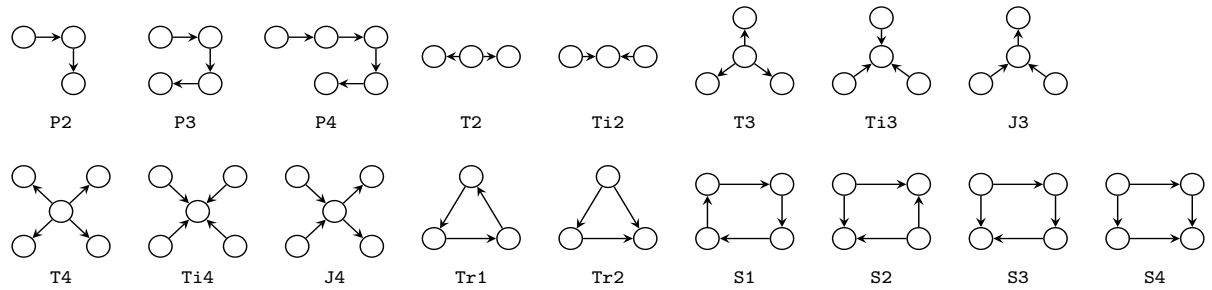

Figure 3: The 17 shapes of synthetic BGPs we test on the Wikidata graph.

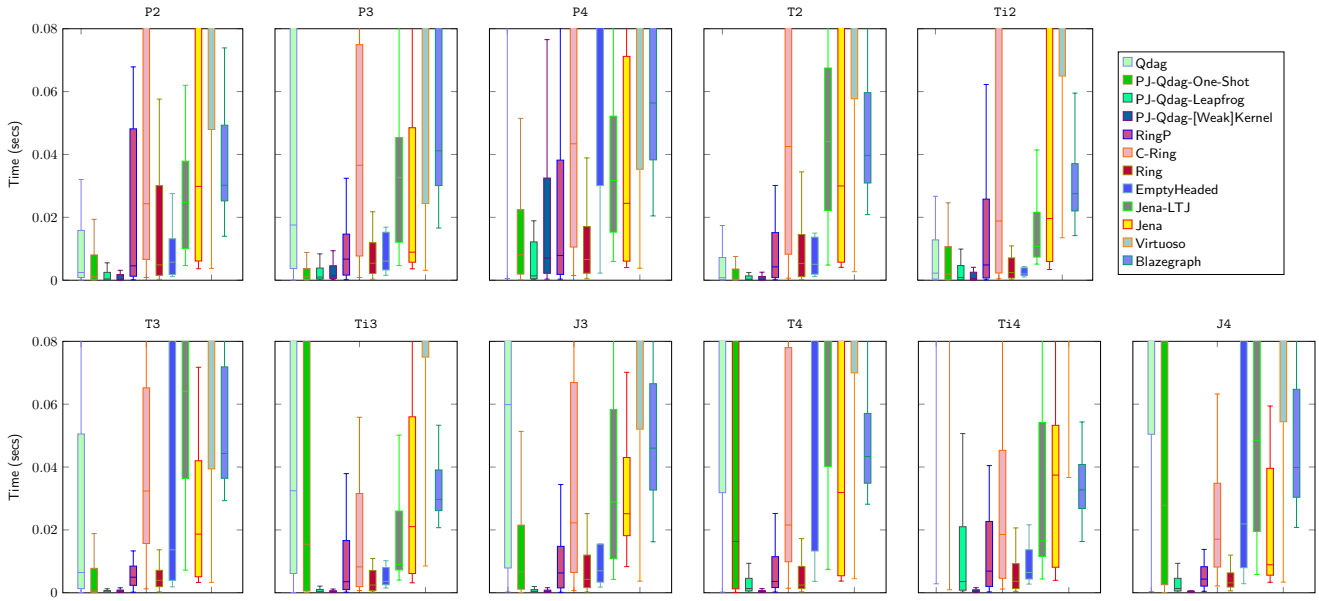

Figure 4: Query times (in seconds) of Qdags and state-of-the-art systems for queries with lonely attributes.

qdags (Qdag), our qdags with prejoining (PJ-Qdag), and the other systems. Qdags and PJ-Qdags are built over the edges sorted lexicographically by their strings, building one 2^2-tree per predicate. PJ-Qdags additionally store two 2^1-trees with the projections of each table, which as seen pose a 30% space overhead over Qdags. This is still well within the most compressed representations of the graph, only paralleled by the two compact Ring variants. The other classical systems use 8–200 times more space than PJ-Qdags.

The table also shows the construction time of the compact indexes. Qdags are built at a rate of 270 thousand edges per second; PJ-Qdags use almost 50% additional time to compute the two projections of each quadtree. The Ring variants require 3–8 times more construction time than PJ-Qdags.

We tried out the *leapfrog* strategy on all the query topologies. In this case, the attributes are chosen by increasing minimum size of their precomputed projections, yet the non-lonely ones are chosen first. On queries with lonely attributes (paths and stars) we also tried *one-shot*. On those with exactly one non-lonely attribute (P2 and stars) we also tried *kernel*, and on the others *weak kernel*, in order to avoid computing projections during the join. Since those strategies are complementary (coinciding on P2, T2, and Ti2) we sometimes put them together as "[Weak]Kernel" in the plots.

Table 1: Index space, in bytes per edge, for all systems, and construction time, in seconds, for the compressed ones.

System	Space	Build time	System	Space
Qdag	4.75	298	Jena	48.42
PJ-Qdag	6.17	441	Jena LTJ	96.83
RingP	4.90	3444	Blazegraph	99.86
C-Ring	6.69	1398	Virtuoso	104.89
Ring	11.17	1337	EmptyHeaded	1292.28

Figures 4 and 5 show barplots with the execution times of all the systems. In terms of distribution of query times, it can be seen that PJ-Qdag always outperforms drastically the original Qdag, becoming clearly the fastest system in almost all cases; it is only matched on squares by EmptyHeaded, which uses 200 times more space. The average results, in turn, are reported in Appendix A.

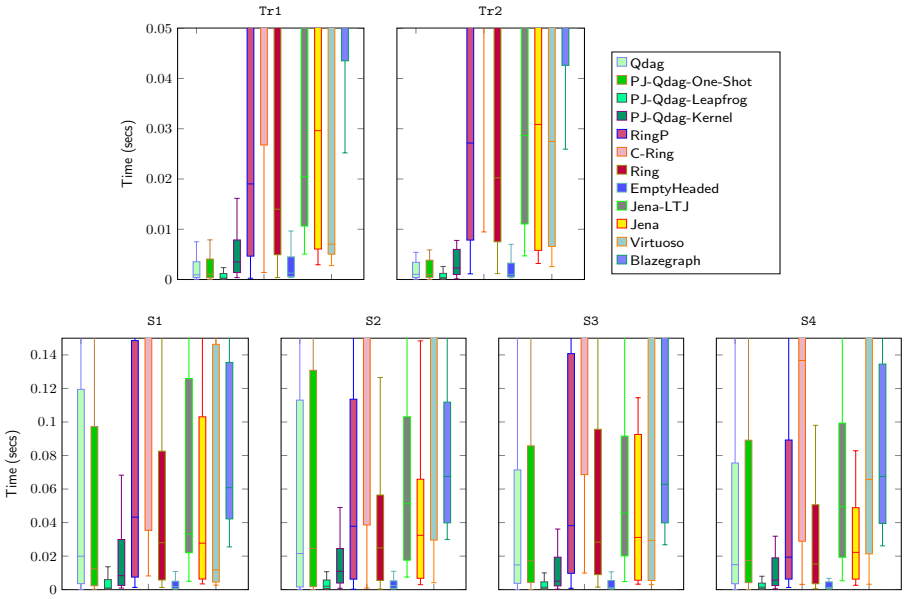

Figure 5: Query times (in seconds) of Qdags and state-of-the-art systems for queries with no lonely attributes.

Considering averages, PJ-Qdag is also the fastest strategy in most cases, though it yields to EmptyHeaded on Ti3 and S1, to the Ring (which uses nearly twice the space) on T4, and to Blazegraph (which uses 16 times more space) on Ti4. The best PJ-Qdag strategies are up to 2,000 times faster than the original Qdags on the average.

The *one-shot* strategy improves upon Qdag, but not by a large margin. More complex strategies obtain orders-of-magnitude improvements: *leapfrog* gives the best results on paths, triangles, and squares, while *kernel* yields the best results on stars. On small paths (P2–3), *one-shot* and *weak kernel* are also competitive. In T4 and Ti4, we obtained better averages (but similar distribution) with an extra filtration step after reducing all the edges in *kernel*: let a be the central attribute and b, c, d, e the others, then after reduction we add two additional prejoining steps, $J_1(a, b, c)$ and $J_2(a, d, e)$. These results, on the 4-stars, are shown as PJ-Qdags-Kernel* in Table 3.

We not only explored the described strategies. Indeed, we programmed a query plan generator that produced *all* the possible sequences of prejoining steps. For example, there are a total of 2,403 plans for squares, using from one to 14 prejoining steps. We tried out them all, with increasing number of steps, until the costs only worsened. The general strategies we have distilled are always among the best, with the exception of the one we described that extends *kernel* on 4-stars, which improves some bad cases of *kernel* and thus its average values (though not the barplots).

6.2 Real queries on the large Wikidata graph

Finally, we compare the compact indexes on actual queries posed by Wikidata users, on our large Wikidata graph, with a limit of 1000 results (limiting is common in real-life scenarios). We will consider only the *one-shot* strategy, which guarantees that there exists at least one output per result of the prejoin. This allows limiting the output to L results on PJ-Qdags by stopping after finding L results in the prejoin and then similarly stopping again in the final join. For the other strategies, we could use iterators for the prejoins so as to extract, progressively, as many results as needed to obtain L final results. We did not implement this last strategy, but use as a proxy a slightly more pessimistic version where we use $2L$ as the limit for the prejoins, and this turns out to be sufficient in our case.

Table 2 shows that PJ-Qdags use 25% more space than the original Qdags on this graph, in exchange for being twice as fast on average. PJ-Qdag is indeed faster on average than any other index using similar space, being outperformed only by the Ring, which is 33% faster at the expense of using twice its space. Further, the Ring has a larger median time than any Qdag structure. Prejoining thus makes qdags a very competitive low-space alternative on real-life queries.

Note that the table distinguishes a simple strategy that always uses *one-shot* from a second one (best plan) that uses the plan that turned out to be the best in our previous experiment on the WGPB dataset, depending on the query shape. For queries where *leapfrog* was the best plan, or queries with a shape not among those we studied, we used *one-shot*. While the average difference with respect to *one-shot* is not so significant, the best plan yields a considerably lower median that outperforms that of the Ring.

We also show a second table including only the queries whose shape is among those 17 we studied (recall Figure 3); those are 88% of the queries. The table shows that PJ-Qdags with the best plan outperform all the systems by far on these queries. Indeed, the queries left out in this table are shown to be much harder, as evidenced by the sharply reduced average times in all the systems. This shows that studying strategies for more complex queries using PJ-Qdags is a relevant future work plan.

6.3 Matrix multiplications

To test matrix multiplications, we solved queries of types P2–P4 by multiplying the 2, 3, and 4 corresponding matrices. P2 is useful to compare the time of a single multiplication, whereas P3 and P4 show

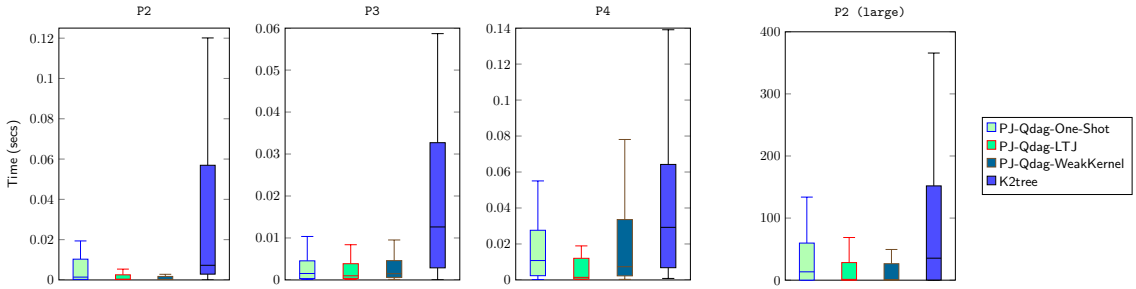

Figure 6: Matrix multiplication time (in seconds). The rightmost plot shows times on the large Wikidata graph.

Table 2: Bytes per edge and times in seconds of the compact indexes for real queries on the large Wikidata graph.

System	Space	Average	Median	Timeouts
Qdag	6.11	10.73	0.006	3
PJ-Qdag-One-Shot	7.64	4.78	0.019	1
PJ-Qdag (best plan)	7.64	4.47	0.012	1
RingP	4.19	5.32	0.060	1
C-Ring	7.31	7.77	0.315	2
Ring	12.15	3.59	0.042	1
On the 172 (out of 195) queries that belong to the 17 shapes:				
System	Space	Average	Median	Timeouts
Qdag	6.11	7.71	0.004	2
PJ-Qdag-One-Shot	7.64	0.42	0.016	0
PJ-Qdag (best plan)	7.64	0.05	0.010	0
RingP	4.19	1.29	0.058	0
C-Ring	7.31	1.60	0.304	0
Ring	12.15	0.17	0.040	0

the effect of running two and three products as a single join using PJ-Qdags, with the *one-shot*, *leapfrog* and *weak kernel* strategies. Note that, in order to stay compositional, PJ-Qdag must not only project the join output onto the two attributes of the resulting matrix as explained, but also onto each of them individually.

As can be observed in the first three plots of Figure 6, the PJ-Qdag variants consistently outperform K2tree, with clear improvements in both median times and variability. This trend is confirmed by the average results reported in Table 5 (columns P2–P4), where PJ-Qdag achieves approximately 2- to 6-fold speedups in P2, about 9- to 13-fold speedups in P3, and 1- to 7-fold speedups in P4, depending on the variant. The best averages are always obtained by *leapfrog*, which outperforms *weak kernel* more clearly as more matrices are multiplied. The same happens with the distributions, which are similar for P2 and P3, but not for P4. Strategy *one-shot* is never the best, but still sharply outperforms K2tree.

On the right of Figure 6 we compare the time of a single multiplication (P2) on the large Wikidata graph. Once again, PJ-Qdag clearly outperforms K2tree, achieving 2.5- to 7-fold speedups, as shown by the average results in the last column of Table 5. This time, *weak kernel* sharply outperforms *leapfrog* on the average; the distributions are much closer but still slightly favor *weak kernel*.

7 Conclusions

We have introduced *prejoining*, a technique that dwarfs the costs of graph database joins using qdags [6], and outperforms all other systems in almost all BGPs. On actual user queries, prejoining makes qdags the fastest index option from those using about 6–7 bytes per graph edge. Prejoining also serves to speed up qdag-based matrix multiplication, virtually replacing the classic recursive algorithm by the well-known Schoor's algorithm [20].

In terms of theory, we have shown that prejoining retains worst-case optimality. In practical terms, prejoining can be regarded as a query optimization technique for wco joins. We have explored various strategies to use prejoining steps, finding some that perform impressively well depending on the query topology. Though these results clearly highlight the possibilities of this technique, more work is needed towards defining general strategies to choose favorable prejoining steps on arbitrary queries.

Actually, the prejoining concept is more general and can lead to many other interesting ideas. Though particularly useful on qdags, which are very sensitive to the number of attributes in the join, prejoining is a general technique that can be applied on top of any wco algorithm. It can be used to implement a variant of GenericJoin [17], where a subset $\mathcal{A}'$ of the attributes is selected, table $I(\mathcal{A}')$ is recursively built, and then the joins $\bowtie_i (R_i \ltimes t)$ are recursively computed and aggregated for every $t \in I$. Prejoining replaces this last step by $(\bowtie_i R_i) \bowtie I$. In particular, our *leapfrog* strategy simulates Leapfrog Triejoin [22] (an instance of GenericJoin). Prejoining can also partially simulate Generalized Hypertree Decomposition (GHD) [2]: for each supernode S defined by GHD, prejoin with $\mathcal{A}_S = \cup_{\mathcal{A}_i \cap S \neq \emptyset} \mathcal{A}_i$; the final join is an acyclic query that can be solved instance-optimally. GHD offers time guarantees beyond worst-case optimality (i.e., the fractional hypertree width).

Finally, our improvements to matrix multiplication will impact on the cost of solving regular path queries (RPQs) within little space [5]. We plan to evaluate such impact, as well as aim to design improved algorithms based on prejoining for computing transitive closures, the other most relevant operation on RPQs.

Acknowledgments

This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile (all authors). José M. Cazorla was also funded by ANID Ph. D. Scholarship program, ID 19731, Chile. Gonzalo Navarro was also funded by Fondecyt Grants 1-230755 and 1-260080, ANID, Chile.

References

[1] Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. Empty-headed: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS) 42*, 4 (2017), 1–44.

[2] Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. Empty-headed: A relational engine for graph processing. *ACM Transactions on Database Systems 42*, 4 (2017).

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

[4] Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Reutter, J. L., Rojas-Ledesma, J., and Soto, A. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems 29*, 2 (2024), article 5.

[5] Arroyuelo, D., Gómez-Brandón, A., and Navarro, G. Evaluating regular path queries on compressed adjacency matrices. *The Very Large Databases Journal 34* (2025), article 2.

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

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

[8] Brisaboa, N. R., Ladra, S., and Navarro, G. Compact representation of Web graphs with extended functionality. *Information Systems 39*, 1 (2014), 152–174.

[9] Erling, O. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull. 35*, 1 (2012), 3–8.

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

[11] ISO/IEC. Information technology — Database languages — GQL, 2024. `https://www.iso.org/standard/76120.html`.

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

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

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

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

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

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

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

[19] Prud'hommeaux, E., and Seaborne, A. *SPARQL Query Language for RDF*. W3C Recommendation. 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[20] Schoor, A. Fast algorithm for sparse matrix multiplication. *Information Processing Letters 15*, 2 (1982), 87–89.

[21] Thompson, B., Personick, M., and Cutcher, M. The bigdata® rdf graph database. In *Linked data management*. Chapman and Hall/CRC, 2016, pp. 221–266.

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

A Average Query Time Results

Tables 3 and 4 show the average times, per system and per query shape, of queries with and without lonely attributes, respectively. Table 5 shows the average times for matrix multiplication.

Table 3: Average query time (in seconds) for queries with lonely attributes across Qdags and state-of-the-art systems.

System	P2	P3	P4	T2	Ti2	T3	Ti3	J3	T4	Ti4	J4
Qdag	0.0182	0.1891	6.7441	0.0069	0.0690	0.0677	1.1936	0.3060	9.3110	52.8996	13.6294
PJ-Qdag-One-Shot	0.0180	**0.0054**	0.0229	0.0067	0.0262	0.0216	1.1741	0.0237	6.7893	38.5017	0.9308
PJ-Qdag-Leapfrog	**0.0109**	**0.0057**	**0.0078**	0.0050	0.0140	0.0016	0.0672	0.0020	1.5574	3.2910	0.0053
PJ-Qdag-[Weak]Kernel	0.0133	0.0078	0.0496	**0.0031**	**0.0075**	**0.0013**	0.0813	**0.0013**	2.0440	1.1237	**0.0006**
PJ-Qdag-PG-Kernel*	-	-	-	-	-	-	-	-	0.0767	0.6564	0.0024
RingP	0.7500	0.0270	0.1868	0.0605	0.0662	0.0102	1.7310	0.0222	0.1128	11.2780	0.0120
C-Ring	2.2916	0.1228	0.4971	0.2881	0.2694	0.0735	2.8878	0.1315	0.3549	12.3383	0.0641
Ring	0.3744	0.0238	0.0868	0.03437	0.5552	0.0079	0.5552	0.0212	**0.0426**	10.7783	0.0110
Jena	67.5167	26.7693	0.9061	45.1280	13.6541	0.5362	9.2204	0.0458	0.6527	14.0966	0.2357
Jena LTJ	0.0292	0.0367	0.0582	0.0568	0.0159	0.0847	0.0318	0.0434	0.1029	0.0480	0.0822
Blazegraph	0.0695	0.0786	1.2284	0.0675	0.0373	0.0648	0.0373	0.1662	0.0550	**0.0433**	0.3203
Virtuoso	0.3451	0.7316	0.3487	1.9408	0.8390	0.2539	0.3522	0.4369	0.3162	13.8899	0.5927
EmptyHeaded	0.0588	0.0877	0.4164	0.0740	0.0214	0.1103	**0.0209**	0.1662	0.2776	0.0829	0.2015

Table 4: Average query time (in seconds) for queries with no lonely attributes across Qdags and state-of-the-art systems.

System	Tr1	Tr2	S1	S2	S3	S4
Qdag	0.0141	0.0094	0.2874	0.2631	0.0903	0.0903
PJ-Qdag-Leapfrog	**0.0052**	**0.0016**	0.0186	**0.0106**	**0.0086**	**0.0045**
PJ-Qdag-WeakKernel	0.0177	0.0080	0.0250	0.0245	0.0378	0.0207
RingP	0.2291	0.1337	0.1907	0.1486	0.2400	0.0952
C-Ring	1.0078	0.7267	0.4832	0.4013	1.1314	0.3689
Ring	0.1679	0.0935	0.0872	0.0505	0.1998	0.0472
Jena	14.6143	13.5740	13.5652	27.0671	31.5361	0.6527
Jena LTJ	0.0716	0.0700	0.0870	0.0889	0.1201	0.0795
Blazegraph	0.3348	0.3218	0.1346	0.4084	35.7139	0.6425
Virtuoso	0.0647	0.1146	0.2193	0.4055	0.3022	0.2468
EmptyHeaded	0.1780	0.0032	**0.0041**	0.0122	0.0192	0.0051

Table 5: Average matrix multiplication time (in seconds). The rightmost column shows results on the large Wikidata graph.

System	P2	P3	P4	P2 (large)
PJ-Qdag-One-Shot	0.0328	0.0074	0.0260	171.6631
PJ-Qdag-Leapfrog	**0.0109**	**0.0053**	**0.0078**	120.1707
PJ-Qdag-WeakKernel	0.0134	0.0080	0.0512	**61.7885**
K2tree	0.0677	0.0697	0.0544	443.1026