Speeding up Qdags with Generalized Hypertree Decompositions

Diego Arroyuelo[1,3,†], Gabriel Carmona[2,†], Gonzalo Navarro[2,3,*,†] and Matilde Rivas[2,†]

[1]Departamento de Ciencias de la Computación, Pontificia Universidad Católica de Chile, Chile
[2]Department of Computer Science, University of Chile, Chile
[3]Millennium Institute for Foundational Research on Data (IMFD), Chile

Abstract

Qdags have proved to be extremely compact representations of graph databases that can still solve basic graph patterns in worst-case-optimal time. Their times are competitive in practice for queries with few variables, but their performance quickly degrades as the number of variables grows over 3. In this paper we combine Qdags with a Generalized Hypertree Decomposition (GHD) of the query, into subqueries with fewer variables. Apart from possibly breaking the AGM bound, implementing GHDs over Qdags has the advantages of reducing the number of variables per subquery, of exploiting the compositionality of Qdags (which return the output in Qdag form), and of using little working space for the intermediate results, which is a problem of GHD compared to standard worst-case-optimal algorithms. We implement the new operations required on Qdags (semijoins) to apply Yannakakis' algorithm over the final acyclic query. We also implement algorithms that find the optimal GHD according to the AGM bounds of the subqueries and the specificities of the Qdag cost model. The result is a much stronger Qdag-based index that can efficiently handle large queries while matching or outperforming competing systems.

Keywords

Graph databases, Worst-case-optimal algorithms, Compact data structures

1. Introduction

Graph databases have become a widespread model to represent and query complex and unstructured relations [1]. A graph database represents data by means of a labeled graph, and the most fundamental query primitive is the Basic Graph Pattern (BGP), a form of subgraph matching. While BGPs can be expressed in the relational algebra, typical BGPs translate into a large number of joins, which have motivated specialized techniques to handle them more efficiently. In particular, the so-called worst-case-optimal (wco) algorithms [2, 3, 4, 5, 6, 7] ensure that the time to solve a BGP is proportional to the maximum possible output size for that BGP on some graph (called the AGM bound). Wco algorithms have been shown to outperform classical relational join plans on real-life graph databases and queries [8]. Bounds more refined than

AMW'26: Alberto Mendelzon Workshop, 2026, Arequipa, Peru
*Corresponding author.
†These authors contributed equally.
diego.arroyuelo@uc.cl (D. Arroyuelo); gabriel.carmona@ug.uchile.cl (G. Carmona); gnavarro@dcc.uchile.cl (G. Navarro); matilde.rivas@ug.uchile.cl (M. Rivas)
http://www.dcc.uchile.cl/gnavarro (G. Navarro)
0000-0002-2286-741X (G. Navarro)

the AGM bound, like the fractional hypertree width (FHW) [9], can be achieved by applying the so-called Generalized Hypertree Decomposition (GHD) to the BGP, which decomposes the query into subqueries, solves each subquery with a wco algorithm, and completes the resolution using the instance-optimal Yannakakis' algorithm [10].

A problem with most wco (and better) algorithms is that they use much indexing space. The GHD-based algorithm may further require a lot of intermediate space to store the outputs of the subqueries. This has hindered their adoption for large graph databases, and has motivated the quest for compact representations [11, 12, 13]. In this paper we focus on one such compact representation, Qdags [11], which use orders of magnitude less space than classical indices.

Qdags offer wco resolution of BGPs, and proved to be practically efficient as well, outperforming most alternative systems on queries with few variables (up to 3). Their performance starts to degrade on BGPs with 4 or more variables, however, making them quickly non-competitive.

In this paper we show that those shortcomings can be overcome, in many cases, by combining Qdags with a GHD-based resolution of BGPs. Concretely, we use the GHD to partition the BGP into subqueries with fewer variables, solve each subquery worst-case-optimally with Qdags, and then apply Yannakakis' algorithm on the resulting acyclic query, still running over the Qdag representation. While running on Qdags prevents us from reaching the FHW bound, the resolution is still wco and has three important advantages that are specific of Qdags:

1. The Qdags perform (possibly much) better on the subqueries because they are BGPs with fewer variables (their cost on d variables include a factor of the form 2^d).
2. We exploit the compositionality of Qdags (i.e., their output is a relation also represented as a Qdag) so as to run Yannakakis' algorithm directly on an indexed representation.
3. The intermediate relations produced by the GHD-based strategy are represented compactly using Qdags, thereby reducing the space needed to store intermediate results.

We face two challenges to carry out this plan: (i) we must implement the semijoin operation, used in Yannakakis' algorithm, over Qdags, which requires new algorithms; (ii) we must devise ways to find good GHDs considering the specific cost model of Qdags (the 2^d factor). While finding the best GHD is NP-hard [14], we manage to do it in reasonable time — using the AGM upper bound to estimate the cost of solving each component. We show that the result is competitive, on real-world graph databases, against the best strategy found by hand query-wise. Further, the resulting strategy outperforms the original Qdags and is competitive with previous compressed indexes [12] and uncompressed GHD-based ones [15], on a benchmark of BGP shapes with more than 3 variables. Overall, Qdags with this enhanced resolution technique become a useful and low-space alternative to solve BGPs, on many real scenarios.

2. Basic Concepts

2.1. Graph databases and Basic Graph Patterns (BGPs)

A *graph database* is a labeled directed graph $G(U, E, \ell)$, where $\ell : E \to L$ assigns label $\ell(e)$ to edge e; if $e = (u, v)$ we write $u \overset{\ell(e)}{\to} v$. We call $N_l = |\ell^{-1}(l)|$ the number of edges labeled $l \in L$, and $N = |E| = \sum_{l \in L} N_l$ the *size* of G.

Given a set V of *variables*, disjoint from U, a *Basic Graph Pattern (BGP)* is a finite set $Q \subseteq (U \cup V) \times L \times (U \cup V) \setminus U \times L \times U$ of *triple patterns*.[1] Calling $V(Q) \subseteq V$ the variables that appear in Q, a *solution* to Q is a mapping $\mu : V(Q) \to U$ that *binds* each such variable x to the node $\mu(x)$, so that, if we extend μ to the identity on U, then for each $(x, l, y) \in Q$ it holds that $\mu(x) \xrightarrow{l} \mu(y)$. *Solving* a BGP Q is the problem of enumerating the set $Q(G)$ of its solutions.

2.2. The AGM bound and worst-case-optimal (wco) algorithms

The *AGM bound* [2, 8] of a BGP Q, Q^*, is the maximum number of solutions Q may have on a graph of size N and with the same numbers N_l for its labels. An algorithm solving Q is *worst-case optimal (wco)* if it runs in time $\tilde{O}(Q^*)$, where the tilde hides factors that are polylogarithmic on N or independent of it (like functions of $|Q|$ or of the number d of variables in Q).

The AGM bound can be computed as $Q^* = \pi_{e \in Q} N_{\ell(e)}^{w_e}$, where the exponents w_e are the solution of the following linear program, where $\mathrm{var}(e)$ are the variables that appear in triple pattern e: $\min \sum_{e \in Q} w_e \log N_{\ell(e)}$ s.t. $w_e \geq 0 \; \forall e \in Q, \sum_{e \text{ s.t. } x \in \mathrm{var}(e)} w_e \geq 1 \; \forall x \in V(Q)$.

2.3. Yannakakis' algorithm

Yannakakis' algorithm [10] solves (certain) joins over general relational tables. Consider a graph where the nodes are the tables to join and there is an edge between each pair of tables sharing an attribute. When that graph is acyclic, the algorithm solves the query in instance-optimal time, proportional to input plus output, provided it solves semijoins $R \ltimes S$ in time $\tilde{O}(|R| + |S|)$. The algorithm arbitrarily fixes a root for the acyclic graph and performs the following phases:

1. (Bottom-up phase) From the leaves to the root, the table of the parent node is semijoined with that of each child node.
2. (Top-down phase) From the root to the leaves, the table of each child node is semijoined with that of the parent node.
3. (Final join) Now the tables have been filtered so that every entry occurs in the output, and the full join is computed.

2.4. Generalized Hypertree Decomposition (GHD)

A *generalized hypertree decomposition (GHD)* [16] of a BGP Q is a tuple $T = (N(T), E(T), \mathcal{B})$, where $(N(T), E(T))$ are the nodes and edges of a tree, and function $\mathcal{B} : N(T) \to 2^{V(Q)}$ assigns a set of variables (called a "bag") to each tree node. A GHD must satisfy[2]:

1. For every triple pattern $e \in Q$, there must be a node $v \in N(T)$ such that $\mathrm{var}(e) \subseteq \mathcal{B}(v)$, that is, the (one or two) variables of each triple pattern must occur in some bag.
2. For every variable $x \in V(Q)$, the set of nodes $v \in N(T)$ such that $x \in \mathcal{B}(v)$ must form a connected subtree of $E(T)$.

[1] In this paper we consider only triple patterns where the labels are constants. This captures most BGPs of interest in real cases. Our version of the AGM bound also considers this restriction.

[2] We ignore here a third condition that is useful only to compute the so-called generalized hypertree width.

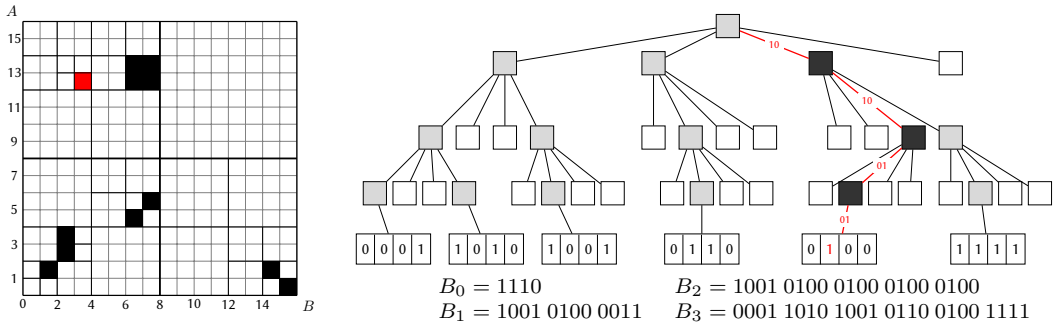

Figure 1: On the left, a grid representing a relation $R(A, B)$ (e.g., the red point is the tuple $(12, 3)$). On the right, the quadtree representing R. The grayed squares are internal nodes, each 1 at the last level indicates a tuple. The labels in the path from the root to a leaf form a string that encodes the coordinates that form the tuple. On the bottom right, the compact representation of the quadtree. Note how the last bitvector coincides with the last-level bits.

For each $v \in N(T)$, let $Q_v = Q|_{\mathcal{B}(v)} = \{e \in Q,\ \text{var}(e) \subseteq \mathcal{B}(v)\}$ be the (sub-)BGP formed by the triple patterns that occur in the bag of v. Then we define [9] $fhw(Q, T) = \log \max_{v \in N(T)} Q_v^*$. We can solve Q in time $\tilde{O}(2^{fhw(Q,T)} + |Q(G)|) \subseteq \tilde{O}(Q^*)$ by running a wco algorithm on each Q_v and then using Yannakakis' algorithm on the acyclic join query where each node v of $N(T)$ is replaced by $Q_v(G)$, the table of all the solutions of Q_v, and the edges are $E(T)$.

The *Fractional Hypertree Width* [9] of Q, $fhw(Q)$, is the minimum of $fhw(Q, T)$ over all possible GHDs T of Q. From a GHD T^* reaching that minimum, we can solve Q in time $\tilde{O}(2^{fhw(Q)} + |Q(G)|)$, which is always wco and possibly better. Finding T^* is NP-hard (on $|Q|$) [14], but this is still within the $\tilde{O}(\cdot)$ term.

3. Qdags

We now describe *Qdags* [11], which represent the relations geometrically in compressed form and offer a wco algorithm, of time $O(Q^* \cdot m2^d \log N)$, to solve BGPs of $|Q| = m$ triples, with $|V(Q)| = d$ variables, on graphs of $|E| = N$ edges. We describe them in the relational setting.

3.1. Compressed quadtrees

The Qdags model represents two-attribute tables using two-dimensional grids. By mapping the universe to an integer domain, each row in the table is regarded as a point in the grid. A (region) quadtree [17, 18] is a hierarchical data structure that represents points in a two-dimensional grid of size $\ell \times \ell$, where for simplicity ℓ is assumed to be a power of 2. The structure is defined recursively as follows (see Figure 1): (i) For $\ell = 1$, the quadtree is just a single value, 1 if the cell contains a point, 0 if not. (ii) For $\ell > 1$, if the grid contains no points, the quadtree is a leaf node. (iii) For $\ell > 1$, if the grid contains points, the quadtree is an internal node with four children, each representing one of the $\ell/2 \times l/2$ quadrants of the grid.

To represent tables with $d > 2$ attributes, quadtrees are generalized to 2^d-trees. Given a d-dimensional grid G of size ℓ^d, each internal node splits into 2^d children. Each child represents

a subspace of size $(l/2)^d$. A leaf in the quadtree represents either an empty submatrix or a single point. As with the 2-dimensional representation, each point in the structure is uniquely identified by the sequence of $\log \ell$ d-bit labels in the path from the root to its leaf. When the dimension of a grid is unspecified, we will refer to its corresponding tree as a Qtree.

Quadtrees can be represented compactly [19] using d bits per internal node, as an array of $\log \ell$ bitmaps $B_0, B_1, \ldots$. Bitvector B_i represents the internal nodes at depth i as a concatenation of their *signatures*. The signature of an internal node are 2^d bits telling which children represent nonempty (1) or empty (0) submatrices of the node. Note that the last bitmap naturally concatenates the 0s and 1s indicating full or empty cells of the matrix.

Note that there is a node in B_{i+1} per 1-bit in B_i. This allows navigating the compact structure in constant time: the jth child of the kth node at level i is found at position $2^d \cdot \text{rank}(B_i, 2^d(k-1)+j)+1$ in B_{i+1}. Operation $\text{rank}(B_i, p)$ counts the number of 1s in $B_i[1, p]$, and can be computed in constant time by precomputing and storing $o(|B_i|)$ bits on top of B_i [20, 21].

3.2. The multijoin algorithm

Consider the triangle query $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$. The key idea to compute such a multijoin builds on its equivalent with the following expression:

$$(R(A,B) \times All(C))) \cap (S(B,C) \times All(A)) \cap (T(A,C) \times All(B)),$$

where $All(X)$ is the set of *all* possible values for attribute X. We call $R(A,B) \times All(C)$ the *extension* of $R(A,B)$ to attribute C: for every $(a,b) \in R$, the extension contains the triples (a,b,c) for all values c. Tables S and T are extended analogously. Qdags implement a wco multijoin algorithm (which includes BGPs) following this idea, in two stages (see Figure 2):

1. The quadtrees representing the joined relations are *extended* to all the d attributes appearing in the query. The extended 2^d-trees are not materialized, but simulated with a new structure called a Qdag, with minimum time and space overhead (see next).
2. The Qdags are intersected, by traversing them all in synchronization and stopping when any of the subgrids is empty. The result of the intersection is a (materialized) 2^d-tree.

Extending dimensions. The values of the extended grid are all copies of those of the original one. In general, a *Qdag* Q extends a $2^{d'}$-tree $Q_{d'}$ representing a d'-dimensional grid, to $d > d'$ dimensions, and is represented as a pair $(Q_{d'}, M)$, where $M : [0, 2^d - 1] \to [0, 2^{d'} - 1]$ is the coordinate mapping function, according to the following rules:

1. If $Q_{d'}$ is a single cell, then Q represents a single cell with identical content.
2. If $Q_{d'}$ is a d'-dimensional empty grid, then Q represents a d-dimensional empty grid.
3. Otherwise, both $Q_{d'}$ and Q have internal root nodes. For each $0 \le i < 2^d$, the i-th child of Q is the 2^d-tree represented by the Qdag $(C(Q_{d'}, M[i]), M)$, where $C(Q_{d'}, j)$ denotes the j-th child of $Q_{d'}$'s root node.

In our example, if the octants of $R(A,B) \times All(C)$ are read in the order $A_1B_1C_1, A_1B_1C_2, A_1B_2C_1, A_1B_2C_2, A_2B_1C_1, A_2B_1C_2, A_2B_2C_1, A_2B_2C_2$, then its M function is $M[0,7] = \langle 0,0,1,1,2,2,3,3 \rangle$, where numbers 0 to 3 indicate the quadrants A_1B_1, A_1B_2, A_2B_1, A_2B_2.

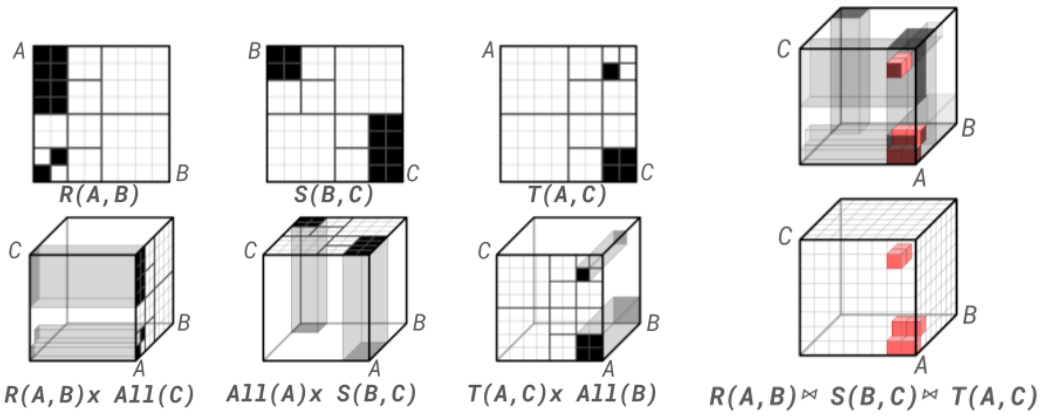

Figure 2: The multijoin algorithm for $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$. The left part shows how the relations are extended to another dimension and the right part depicts the intersection process.

Tree traversal and intersection. We start at the root of all the m Qdags $Q^1, \ldots, Q^m$ (which are all on the same attributes) and recursively traverse their 2^d children. If any child is empty on some Qdag, it is also empty in the output 2^d-tree. Otherwise we enter into that child in all the Qdags and continue recursively.

The descent may continue until the leaves of the trees are reached. All the leaves reached are intersection results and are written to the output. The output is written from bottom to top when returning from the recursive call. If intersection results have been found in the subtree of a child node, then their positions are stored in a list containing all the bits that should be '1' in the current level. The final bitmaps B_i are constructed from these lists.

4. Semijoins over Qdags

The semijoin operation is a key component of Yannakakis' algorithm for solving acyclic join queries. A semijoin between relations R and S, denoted $R \ltimes S$, produces the subset of tuples from R that participate in the join with S, preserving only the attributes of R; formally, $R \ltimes S = \pi_R(R \bowtie S)$. For the bottom-up phase of Yannakakis' algorithm, we implement a multi-semijoin over various relations, $R \ltimes S_1 \ltimes \cdots \ltimes S_m$, instead of solving them sequentially.

4.1. Active bitmaps

We solve semijoins $R \ltimes S_1 \ltimes \cdots \ltimes S_m$ with the same steps of extending and traversing described in Section 3.2 for the multijoin $R \bowtie S_1 \bowtie \cdots \bowtie S_m$. The difference is that, when an output cell is found during the traversal, instead of emitting the position of its 1 to later build the join output, we *mark* the corresponding cell in the Qtree of R. To implement this marking, we add an additional "active" bitmap A_i to each level of the Qtree structure. Bitmap A_i indicates whether the corresponding quadtree node is active, that is, has not been filtered out by a semijoin. If a bit is not set in B_i or is not set in A_i, we act as if its subgrid were empty when traversing a

Qdag that extends the Qtree. Actually, as we will see, bits set in A_i must be set in B_i, so we simply replace B_i by A_i when performing the intersection traversal.

We cannot simply unset the bits in bitmaps B_i that are not set in A_i, because bitmaps B_i are static and used for tree navigation, maintaining node order, and computing rank operations. Modifying those bitvectors requires linear-time recomputation of the Qtree structure. The active bitmap approach allows us to logically filter tuples without physically modifying the underlying data structure.

4.2. Tree traversal

To prepare for the semijoin, we create bitmaps A_i for R as copies of B_i. Later, when performing a semijoin, some tuples are filtered out and their corresponding bits in bitmaps A_i are unset.

As explained, the semijoin algorithm works similarly as the multijoin: all the participant Qtrees are extended to Qdags on the query dimension and then a recursive traversal intersects them. Instead of generating the output, however, we mark the bits reached in the Qtree of R.

We use another set of temporary bitmaps T_i that is aligned with the bitmaps A_i, and initialized at zero. Whenever we reach a tuple that belongs to the semijoin, we set its corresponding bit in (the last-level) bitmap T_i, using function M to map the index of the bit within the Qdag leaf to its index within the Qtree leaf. After the traversal finishes, we traverse the quadtree in DFS order and update the bitmaps T_i as follows, upon returning from the recursion: If $T_i[p, p + 2^{d'} - 1] \neq 0$, set the corresponding bit of its parent's signature in T_{i-1} to 1. Finally, the T_i bitmaps become the new A_i bitmaps after the semijoin.

It is tempting to avoid the use of T_i and act directly on the bitmaps A_i, but relation R may have been semijoined already and have its active bitmaps A_i in use, so we need to retain them for proper navigation over its surviving cells while we further eliminate some using T_i. This is because Yannakakis' algorithm has a bottom-up and then a top-down pass over the same Qtrees. Note also that its final join is carried out over the Qtrees restricted by semijoins, so it must use the bitmaps A_i for the intersections.

4.3. Pruning

The semijoin process traverses the Qdag that extends R, but it marks the corresponding leaves in the Qtree of R, and thus the same Qtree nodes can be marked multiple times. In a join, each marking corresponds to new results, but in a semijoin they are redundant. To reduce this redundancy, we will make use of the bitmaps T_i of the internal nodes, which are unused so far.

A bit set at a bitmap T_i will indicate that *all* the descendant leaves of the corresponding node are already marked, so there is no need to revisit that subtree. During the traversal, if we arrive at a Qdag node whose corresponding Qtree node is marked in T_i, we will prune the backtrack as if that node of R were a leaf. Correspondingly, when we return from the backtrack recursion, if the signature $A_i[p, p + 2^d - 1]$ of a node is equal to $T_i[p, p + 2^d - 1]$, then all the children of the node are marked and we set to 1 the corresponding bit in its parent, in T_{i-1}.

Pruning saves considerable time, especially on dense relations, which are the most costly to process. At the end of the intersection traversal, we reset the bitmaps T_i of the internal nodes to zero, to carry out the update of A_i described previously.

5. GHDs on Qdags

Assume a GHD has been defined for Q, which includes choosing a root node. Each bag is a union of variables of triple patterns, that is, $\mathcal{B}(v)$ equals the set of variables in Q_v for all nodes v (recall Section 2.4). Our algorithm solves the multijoin as follows.

1. It creates the tree corresponding to the chosen GHD, where each node v records the set Q_v of triple patterns assigned to it (a triple pattern may occur in several bags).
2. It solves, using the Qdag multijoin algorithm, the subqueries Q_v. At the end, each node has a Qtree t_v representing $Q_v(G)$.
3. It creates the bitmaps A_i, as copies of B_i, and the zeroed bitmaps T_i, for each t_v.
4. It runs the bottom-up pass of Yannakakis' algorithm (recall Section 2.3), by means of multi-semijoins $t_v \ltimes t_{v_1} \ltimes \cdots \ltimes t_{v_h}$ where v is the parent node of $v_1, \ldots, v_h$ in the GHD tree (we identify relations with their Qtrees here).
5. It runs the top-down pass of Yannakakis' algorithm, by means of simple semijoins $t_{v_i} \ltimes t_v$ for each child v_i of v in the tree decomposition.
6. It performs the final join between the trees t_v of all the nodes, restricted with their bitmaps A_i, using once again the standard Qdag multijoin algorithm.

5.1. Finding Optimal GHDs

We perform a branch-and-bound backtracking over all possible sets of bags, which will become the sets $\mathcal{B}(v)$ of the GHDs T. Only bags that are unions of triple patterns $e \in Q$ are considered. Bags that are subsets of others are removed. For each candidate set of bags, we check that it admits a tree $(N(T), E(T))$ satsfying the GHD conditions (Section 2.4).

Each qualifying set of bags $(S_i)_i$ is evaluated according to the AGM bound of the corresponding queries $Q_i = Q|_{S_i}$. Because Qdags include a prominent component 2^d in the cost of a d-variable query, we use $\mathit{fhw}(Q, (S_i)_i) = \log \sum_i 2^{|S_i|} \cdot Q_i^*$, so that 2^{fhw} better approximates the cost of solving all the sub-BGPs using Qdags. The values Q_i^* are found by solving the linear program of Section 2.2. Interestingly, we had better results using the simplification $\mathit{fhw}(Q, (S_i)_i) = \log \sum_i 2^{|S_i|}$, which for lack of space is the one we show in the experiments.

We first run a branch-and-bound search over a reduced space, to obtain a good initial solution. In the reduced search space, we do not explore the possibility of adding new bags to a set of bags that has no possible tree $(N(T), E(T))$. This reduced search turns out to find, relatively quickly, a good candidate. This increases the effectiveness of a second branch-and-bound search, now over the full set of valid GHDs. In both searches, we evaluate the bags from bigger to smaller, to detect as early as possible the the candidates that exceed the best fhw found so far.

6. Experimental Results

We implemented our prototype in C++11 and compiled with GNU g++ with -O3 flag. We measured the performance on a server featuring two Intel Xeon Silver 4110 processors at 2.10 GHz, with a 735 GB RAM. We measure user times and set a timeout of 600 seconds. We index the Wikidata Graph Pattern Benchmark [8], a Wikidata subgraph with $N = 81,426,573$ edges,

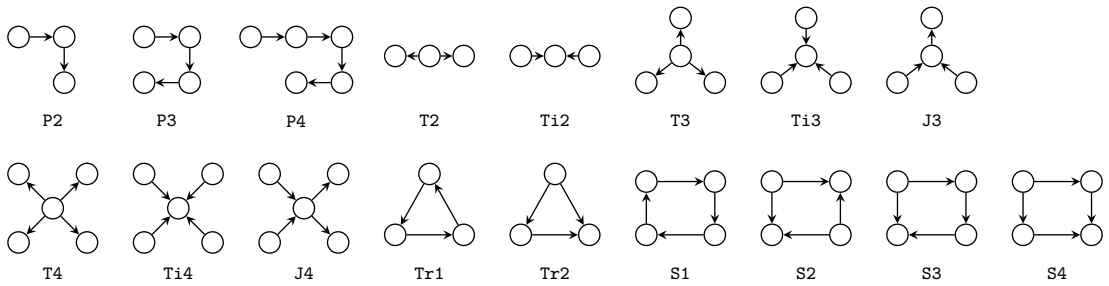

Figure 3: The 17 query shapes tested on the Wikidata graph.

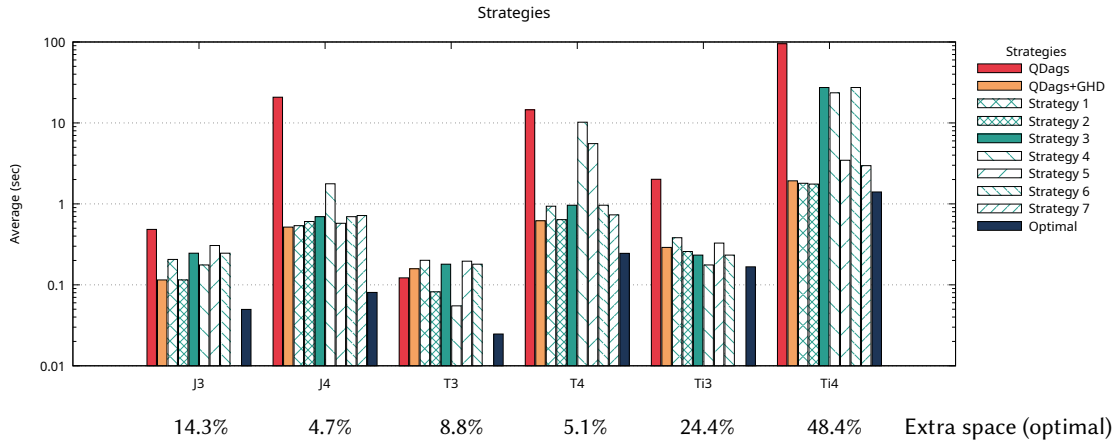

Figure 4: Average times, in seconds, of original versus our improved Qdags. Logscale.

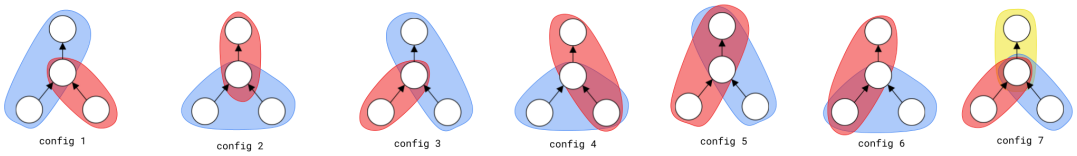

Figure 5: The 7 decompositions generated by hand for query shape J3.

$|U| = 51,999,296$ nodes and $|L| = 2,101$ predicates (labels). The benchmark defines 17 query patterns with path, star, and cycle patterns (see Figure 3). Each pattern is instantiated to 50 queries by choosing real Wikidata predicates at random, ensuring that the query has outputs.

Strategies. Figure 4 compares, on some interesting shapes, the times of QDags and our new technique (QDags+GHD). We also show 6–7 GHDs chosen by hand for each query shape (Figure 5 shows the 7 GHDs for J3). The last bar, "Optimal", chooses the best time among those 6–7 GHDs, for each of the 50 queries, and averages those point-wise minima, simulating an oracle that gives the best GHD for each individual query (not just for each query shape).

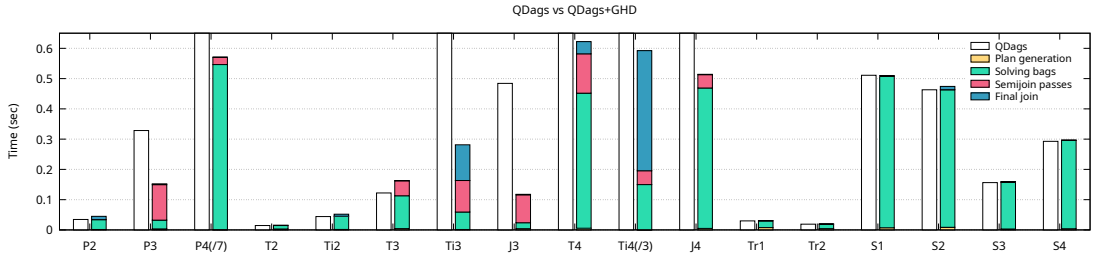

Figure 6: Average times breakdown, in seconds, of original (white) versus our improved Qdags. For legibility, P4 times are divided by 7 (QDags time is 10.6) and Ti4 times are divided by 3.

A first conclusion is that QDags-GHD improves the original QDags by up to 2 orders of magnitude, making them much more competitive. Only in T3 the new strategy is (slightly) worse than the original QDags. A second conclusion is that our automatic query plan generator is competitive with all those generated by hand, in most cases matching the best ones and clearly outperforming the others; the exceptions are T3 and Ti3. This shows that our prediction technique works quite well compared to hand-generated strategies. A third conclusion is that the oracle (the point-wise minimum) is far superior to our automatic plan generator, and to all the hand-designed strategies, always outperforming QDags by 1–2 orders of magnitude. This shows that (i) choosing the best plan in all cases is challenging, and (ii) our technique still has the potential of offering much bigger savings if we manage to generate better plans.

Additional space. The bottom of Figure 4 shows the extra space, on top of the indexes, used by the intermediate QTrees of GHD (with the optimal strategy). It is typically rather low, but it can grow up to 50% on top of the data, especially when the output itself is large. Still, note that the QTrees use little space (around 4.75 bytes per edge), so the extra working space is also low.

Time breakdown. Figure 6 compares original with our improved QDags for all the shapes and also shows a breakdown of the times of the different stages of our algorithm. First, note that the plan generation imposes a minimal overhead, despite taking nonpolynomial time on $|Q|$. Second, QDags are already fast on the smallest and cyclic queries, and our strategy does not improve them, choosing to solve them as a single bag. On the others, where our strategy makes a significant difference, the times are divided differently depending on the shape: most of the time is spent on solving the bags on P4, T4, and J4, whereas the final join takes a significant time on Ti3 and Ti4. The semijoin passes are generally noticeable, and even dominate on P3.

7. Conclusions and Future Work

We have show that combining QDags with Generalized Hypertree Decompositions yields orders-of-magnitude improvements on many relevant queries, making the result an attractive low-space alternative to solve larger BGPs. Future work involves exploring stronger query plan generation strategies, extension to BGPs with more variables, and multithreaded execution.

Acknowledgments

Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile, and by Fondecyt Grant 1260080, ANID, Chile.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

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

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

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

[4] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, 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), 2015, pp. 2:1–2:8.

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

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

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

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

[9] M. Grohe, D. Marx, Constraint solving via fractional edge covers, ACM Transactions on Algorithms 11 (2014) 4:1—-4:20.

[10] M. Yannakakis, Algorithms for acyclic database schemes, in: Proc. 7th International Conference on Very Large Databases (VLDB), 1981, pp. 82–94.

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

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

[13] D. Arroyuelo, D. Campos, A. Gómez-Brandón, Y. Linker, G. Navarro, C. Rojas, D. Vrgoc, CompactLTJ: Space & time efficient Leapfrog Triejoin on graph databases, The Very Large Databases Journal 34 (2025) article 67.

[14] W. Fischl, G. Gottlob, R. Pichler, General and fractional hypertree decompositions: Hard and easy cases, in: J. V. den Bussche, M. Arenas (Eds.), Proc. 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS), 2018, pp. 17–32.

[15] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, C. Ré, Emptyheaded: A relational engine for graph processing, ACM Transactions on Database Systems 42 (2017).
[16] G. Gottlob, N. Leone, F. Scarcello, Hypertree decompositions and tractable queries, Journal of Computer and System Sciences 64 (2002) 579–627.
[17] D. S. Wise, J. Franco, Costs of quadtree representation of nondense matrices, Journal of Parallel and Distributed Computing 9 (1990) 282–296.
[18] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2006.
[19] N. R. Brisaboa, S. Ladra, G. Navarro, Compact representation of Web graphs with extended functionality, Information Systems 39 (2014) 152–174.
[20] D. R. Clark, Compact PAT Trees, Ph.D. thesis, University of Waterloo, Canada, 1996.
[21] J. I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.