UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE COMPUTACIÓN

ENHANCING QDAGS FOR HIGHER-DIMENSIONAL QUERIES

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MATILDE RIVAS LAGOS

PROFESOR GUÍA:
GONZALO NAVARRO
PROFESOR GUÍA 2:
DIEGO ARROYUELO

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS
AIDAN HOGAN
JUAN REUTTER

SANTIAGO DE CHILE
2026

Resumen

El *join* es una operación fundamental en las bases de datos relacionales y de grafos. Las soluciones tradicionales, con un enfoque de join de a pares, suelen generar resultados intermedios grandes que exceden el tamaño final del resultado. Una tarea común en las bases de datos de grafos es la búsqueda de subgrafos, lo que requiere llevar a cabo varios *joins*. Aunque los algoritmos de join *Worst-Case Optimal* (WCO) tienen una complejidad computacional acotada por la cota Atserias-Grohe-Marx (AGM), típicamente requieren gran indexamiento, que a su vez requiere una cantidad significativa de recursos de memoria, lo que los hace imprácticos para resolver consultas de subgrafos a gran escala.

Esta tesis extiende el Qdag, un framework que representa bases de datos como quadtrees comprimidos y provee algoritmos de join WCO en espacio cercano al óptimo. Sin embargo, el Qdag original está limitado a consultas de baja dimensionalidad (hasta 5 variables) debido a la dependencia exponencial del tiempo de ejecución con la dimensión del resultado.

Presentamos dos contribuciones principales para superar esta limitante. Primero, introducimos una versión modificada del Qdag, llamado *high arity Qdag*, con un nuevo algoritmo de join que resuelve consultas de aridad arbitraria a través del uso de un arreglo inicializable de contadores, eliminando el límite sobre el número de variables en la consulta. Segundo, desarrollamos un algoritmo de join que utiliza *generalized hypertree decompositions* (GHD) como planes de consulta, potencialmente logrando una mejor complejidad que la cota AGM. Este algoritmo basado en GHD no solo reduce la dimensionalidad efectiva de los joins a resolver, también aprovecha la naturaleza compacta de los Qdags para almacenar los resultados intermedios de manera eficiente.

Evaluamos nuestras soluciones con el dataset de Wikidata, mostrando que el algoritmo basado en GHD presenta mejoras de rendimiento significativas en relación al Qdag original. Aunque el *high arity Qdag* tiene un peor rendimiendo que el Qdag original en queries de aridad baja, permite procesar patrones complejos previamente fuera del rango de capacidades del Qdag, manteniendo la cota AGM.

Estos resultados muestran que nuestras mejoras éxitosamente extienden las capacidades del Qdag, manteniendo sus ventajas fundamentales. También establecen una base para seguir mejorando los algoritmos de join y la búsqueda en grafos.

Abstract

Join processing is a fundamental operation in relational and graph databases, with traditional pair-wise approaches often generating large intermediate results that exceed the final output size. A common task in graph databases is subgraph matching, which requires executing many joins. While Worst-Case Optimal (WCO) join algorithms achieve optimal time complexity bounded by the Atserias-Grohe-Marx (AGM) bound, they typically require heavy indexing, which needs significant memory resources that make them impractical for large-scale subgraph matching queries.

This thesis extends the Qdag, a framework that represents databases as compressed quadtrees and supports WCO join algorithms with near space-optimality. However, the original Qdag is limited to lower-dimensional queries (up to 5 variables) due to its exponential runtime dependence on output dimension.

We present two main contributions to overcome these limitations. First, we introduce a modified Qdag structure with a new multijoin algorithm that handles queries of arbitrary arity through an initializable array of counters, removing the limit on the number of query variables. Second, we develop a join algorithm that uses generalized hypertree decomposition (GHD) as a query plan, potentially achieving better complexity bounds than AGM. This GHD-based algorithm not only reduces the effective dimensionality for applying WCO algorithms but also leverages the compact nature of Qdags to efficiently store intermediate results.

We evaluated our solutions on the Wikidata dataset, showing that the GHD-based algorithm provides significant performance improvements over the original Qdag. While the high arity Qdag has a worse time performance than the original Qdag in lower-arity queries, it enables processing of complex graph patterns previously beyond the Qdag's capabilities, and it is still WCO.

These results show that our enhancements successfully extend Qdag capabilities while preserving their core advantages, and establish a foundation for further improving join algorithms.

Este trabajo se lo dedico a todas las personas que ven valor en hacer cosas, en crear, en pensar, en sentir, en frustrarse, y en no dejar que un conjunto de if-else y vectores de tokens vivan la vida por ellos.

Acknowledgments

He vivido un sinfín de vidas en el transcurso de esta tesis y agradezco a cada persona que tocó alguna de esas vidas por recordarme todas las posibilidades y maravillas que esconde este mundo. Si no me creen lo primero, les doy algunas cifras: tejí dos chalecos y empecé otros tres que probablemente nunca terminaré; viajé a 20 ciudades; fui a clases de salsa, bachata, ballet, yoga, twerk y pilates; también tomé talleres de canto y cerámica; participé en dos competencias de pitch (una en Francia!) con proyectos diferentes y mi equipo salió segundo en ambas; lei 160 libros y no terminé ninguna serie; participé de al menos 10 noches de trivia y sacamos podio en la mayoría; hice 12 disfraces; tuve 4 trabajos diferentes: el IMFD, el proyecto AVES, Fire2A en el ISCI y CeroAI (más la Ocean Hackathon que a veces se sentía como un trabajo real); participé en 2 papers que llevan meses en revisión; hice 2 cursos online sobre medio ambiente; viví la pandemia, me independicé y me comprometí; nacieron mis tres sobrinos; conversé con mucha gente interesante e hice y deshice amistades.

Concrétamente, quiero agradecer a mi familia por enseñarme a darlo todo por las cosas que me importan, a tomarme en serio lo que amerita ser serio, y con risas lo demás. Seremos controladores pero al menos somos chistosos. Muchas gracias a mi mamá y mi papá por todos los abrazos y consejos que me dieron a causa de la tesis.

Agradezco a todos mis amigos por mantenerme anclada durante este proceso. A mis amigas del colegio, que me hacen recordar que hay vida afuera de Beauchef. A los frens del DCC: ColoColo es Chile. Muchas gracias por ser una burbuja de compasión, chistes estúpidos y entendimiento sin tener que decir ninguna palabra.

Debo hacer mención especial de algunas personas cuya amistad me permitió seguir adelante en los momentos más difíciles. Ann, gracias por recibirme tantas veces en tu casa y por enseñarme a no estresarme por tonteras. La frase "qué haría la Ann en esta situación" me ha ahorrado harto cortisol, así que podrías decir que tu perspectiva me ha salvado al menos unos días de vida. Has sido un lugar seguro y lleno de risas para mí. Kari y Benja, qué manera de reirnos. Es realmente impresionante como dentro de la misma conversación podemos hacer un chiste estupidísimo que nos saca carcajadas, y luego alguno dice un comentario tan sabio que saca aplausos. Gracias por mantenerme aflote, espero con ansias el reencuentro.

Por sobretodo agradezco a Cristóbal por acompañarme en esta travesía que a ratos parecía infinita. Sin su paciencia para debuggear, este trabajo derechamente no hubiera sido posible. Así como sin su cariño, su confianza en mis capacidades o sus palabras de ánimo. Gracias por aguantar mi desorden y por lavar los platos cuando estuve abrumada por meterme en tantas cosas a la vez. Espero que sepas que nunca cambiaré.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the field of relational databases, the natural join operation is a fundamental primitive that combines data from multiple tables by combining tuples from participating relations that share the same values for common attributes. The results of a join query provide a comprehensive view of the data by aggregating information from various sources. Traditional query plans have often relied on pair-wise joins, transforming the query into a series of intermediate join operations between two tables at a time. However, this approach is known to be suboptimal in many cases, generating large intermediate results that may exceed the final output size [10].

To illustrate this limitation, consider finding triangles in a social network represented by a single relation $R(x, y)$ that denotes connections between users. A traditional pair-wise plan for the query $R(x, y) \bowtie R(y, z) \bowtie R(z, x)$ would first, for example, compute all $R(x, y) \bowtie R(y, z)$ tuples, potentially generating $O(N^2)$ intermediate results for a graph with N edges, even though the final output cannot exceed $O(N^{3/2})$ triangles [4].

The definition of the Atserias, Grohe and Marx (AGM) bound has been a significant breakthrough in this area, providing tight bounds on the output size of a join query in terms of the sizes of the participating relations [4]. Building upon this, the concept of Worst-Case Optimal (WCO) join algorithms has emerged, where the algorithms have a worst-case runtime that is essentially bounded by the AGM bound. In other words, if the worst-case output size is Q* results, a WCO join algorithm will run in time O(Q*), possibly multiplied by polylogs or data-independent factors.

The emergence of graph databases has introduced new challenges in join processing. Graph databases represent data as nodes and relationships, having applications in social networks, bioinformatics, and knowledge bases like Wikidata[1]. A common task done in these systems is to search for occurrences of a particular configuration of nodes and edges within the network. For example, we may want to obtain all the nodes and edges in a network that form a square, or a more complex shape. These configurations are called basic graph patterns, and in this work we simply refer to them as patterns. Pattern matching queries are based on the join

[1]More information about Wikidata can be found here `https://www.wikidata.org/wiki/Wikidata:Main_Page`

operation, making efficient join processing central to their performance.

While WCO join algorithms achieve optimal time complexity, they usually lack efficiency in their space requirements. Current implementations require significant memory resources to store intermediate results, index structures and auxiliary data structures [3]. This space requirement becomes prohibitive when processing large graphs, in which the number of potential matches can grow rapidly with the size of the input. For example, the open-source graph database MilleniumDB [13] uses 156 bytes per triple to store the Wikidata dataset [2].

Compact data structures offer a promising approach to reduce memory usage while maintaining efficient query processing capabilities, by representing data in compressed form while supporting fast operations. The first index proposed to address this is the Qdag, which represents graph databases as compressed quadtrees and offers a WCO join algorithm that is time-optimal and nearly space-optimal [9]. However, the Qdag is limited to lower-dimensional queries (in regards to the number of variables in the query), as its running time is exponential in the dimension. This means that it stops being competitive when the results yield relations with 5 or more attributes, or in other words, it cannot search for graph patterns composed of more than 5 nodes within a competitive time.

In this thesis, we explore how the integration of Qdags into a Generalized-Hypertree-based query decomposition strategy affects the trade-off between time complexity and space efficiency of subgraph matching queries in graph databases. Our initial hypothesis is that the proposed algorithm will reduce query times not only in complex queries but also in simpler ones. This is because Qdags excel in low-attribute queries (subgraph patterns composed of 3 nodes), so it is ideal for solving the subqueries yielded by the query decomposition. We also hypothesize that these improved times will come at a very small memory cost when compared to existing solutions, due to the fact that Qdags would use a significantly less amount of space to store intermediate query results.

The main objective of this thesis is to build upon the Qdag to extend its applicability to more complex graph pattern operations. The first specific objective is to introduce modifications to Qdags that allow them to be used to solve higher-arity join queries and semijoin queries. We also aim to present an algorithm to address the efficiency challenge in WCO join algorithms and go beyond the AGM bound, through using a generalized hypertree decomposition of the query as a query plan. Our final specific objective is to compare our algorithms to existing solutions in terms of time and space usage.

The complete source code for both solutions can be found in a public Github repository: https://github.com/matildeRivas/ccqQDAGS.

1.1 Thesis Outline

The thesis is organized as follows:

Chapter 2: Graph Databases presents graph databases and their challenges.

Chapter 3: The Qdag introduces the central data structure used in our algorithm, along

with key concepts of compact data structures.

Chapter 4: Higher-arity Qdags presents an alternative Qdag structure and multijoin algorithm for solving queries of arbitrary arity.

Chapter 5: GHD-based algorithm describes a new join algorithm for solving cyclic queries.

Chapter 6: Experimental Evaluation presents an empirical analysis of our algorithms, comparing them to the original Qdag.

Chapter 7: Conclusions summarizes our findings and contributions. We also discuss directions for future research.

Chapter 2

Graph Databases

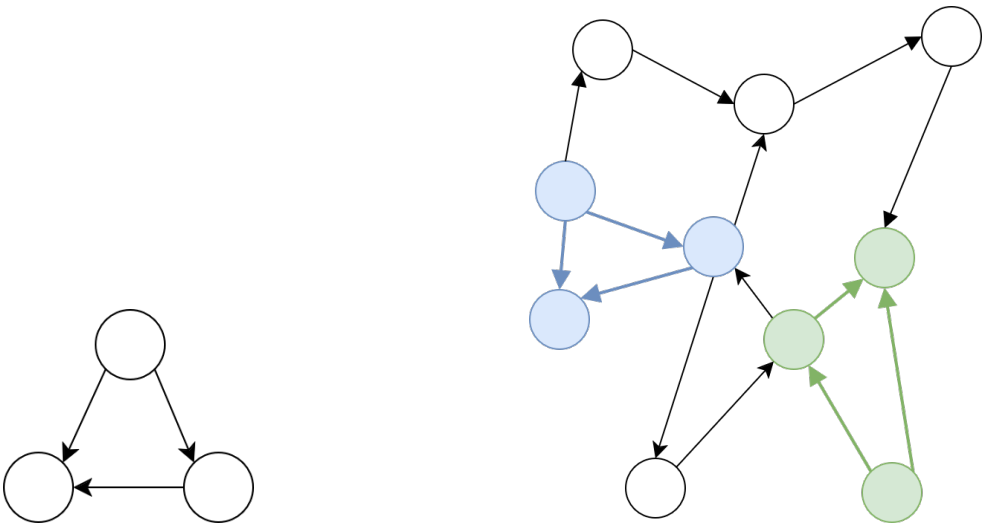
This chapter is dedicated to presenting graph databases, which are the basis of our research motivation. We start by explaining how they store data, the concept of graph pattern matching and how it relates to the join operation, and the advances that have been made in the field of join queries.

2.1 Graph Databases and Worst Case Optimal Joins

Graph databases represent information as a network, in which nodes are entities and edges indicate how they are connected. To understand how two entities are related, we look at the triples they form. A triple is composed of a subject, a predicate, and an object. The two nodes are the subject and the object, while the arc is the predicate. Graph databases are commonly used to integrate data coming from different sources, since it is not necessary to know all of the data types that will be stored beforehand. Let us say we have a database that stores all of an organization's information, and a node representing an employee in the organization. This employee could be connected to another employee through the predicates `referredBy` and `supervises`, and to a node representing a city through the predicate `livesIn`. To add more information that is only applicable to a particular employee, for example `leadsProject`, we do not have to update an "employee" table and modify all of its tuples as we would in a relational database. Instead, we just add the node representing the project and the edge linking the employee to the project.

Graph pattern matching is looking for instances of a template of nodes and edges in the larger graph. For example, say that we have a graph database representing a social network and we want to find all groups of three mutually connected friends. To do this, we would look for triangles in the graph, as in Figure 2.1.

Edges often connect different types of nodes through various relationships. Pattern matching is done by executing join queries over the edge relations of a graph. Consider finding researchers who collaborate with people from the same university. We can express this as a join query: $\text{Collaborates}(x, y) \bowtie \text{WorksAt}(x, z) \bowtie \text{WorksAt}(y, z)$. The query joins three



(a) (b)

Figure 2.1: An example of searching for a triangle pattern (a) in the network (b). Highlighted nodes and edges are found matches.

relations: a collaboration between researchers (x, y), and their institutional affiliations (x, z) and (y, z).

Traditional pairwise join query plans face performance challenges when matching complex graph patterns [10]. A traditional plan might first compute Collaborates $\bowtie$ WorksAt to find all collaborator-institution pairs, even though many of these intermediate results may not lead to a complete pattern match. An alternative is to conduct a multiway join, in which multiple relations are processed at once. Worst Case Optimal (WCO) join algorithms compute multiway joins in a single pass, eliminating many unnecessary intermediate results and thus reducing the use of resources.

The AGM bound proves that for the triangle query above, no instance of relations with N edges can output more than $O(N^{3/2})$ triangles. Traditional pair-wise join plans produce $\Omega(N^2)$ intermediate results and exceed this bound, making them sub-optimal in the worst case. WCO join algorithms, such as Leapfrog Triejoin [12], achieve the AGM bound by avoiding large intermediate results through different strategies, commonly based on either ordering the participating attributes in a way that is optimal to a specific query, or building additional data structures [11].

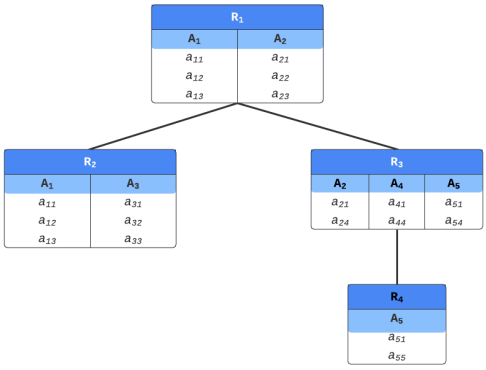
The theoretical optimality of WCO join algorithms extends beyond triangle queries to general join patterns. For a join query over relations of size N, if the AGM bound states that the maximum output size is $O(N^k)$, then WCO algorithms guarantee a runtime of $O(N^k)$ [4]. This matches the best possible runtime in the worst case, as any algorithm must at least read all results.

2.2 Yannakakis Algorithm

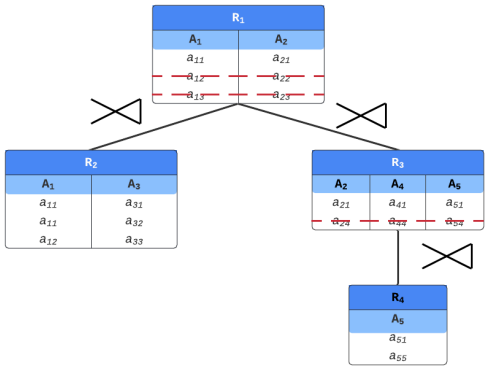
If a join query is acyclic, then it can be solved using the Yannakakis algorithm in time $O(|input| + |output|)$ [14]. For a query to be acyclic, it needs to have a *join tree*, in which each node is a participating relation and two nodes are connected by an edge if they share an attribute. 2.2a depicts a join tree for the query $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$.

The general idea of the Yannakakis algorithm is to do two sweeps of the join tree, first from the leaves to root, and then from root to the leaves. In each sweep a node will constrain the next node (either its parent or child depending on the sweep) by doing a semijoin between the two relations, eliminating the tuples that do not have a match. A final sweep is made by doing a join on the reduced tables. Figure 2.2 shows this process step by step using a basic example.

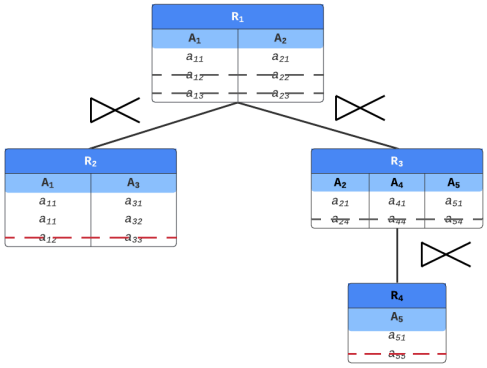
Since the first two sweeps are proportional to input size, and the final sweep is proportional to the output size, this algorithm's runtime is $O(|input| + |output|)$.



(a) Original query join tree



(b) Join tree after bottom-up traversal



(c) Join tree after top-down traversal

A1	A2	A3	A4	A5
a11	a21	a31	a41	a51
a11	a21	a32	a41	a51

(d) Output after final join

Figure 2.2: Example of Yannakakis algorithm for acyclic join queries.

2.3 Generalized Hypertree Decompositions

A query can be represented as a hypergraph[1], in which each attribute is expressed as a node, and relations are represented by hyperedges [5]. A generalized hypertree decompositions (GHD) of a hypergraph is a pair (T, λ), in which T is a tree and λ is a function mapping a set of hyperedges of the hypergraph to each node of T. A GHD node contains a subset of attributes such that for every relation in the query, its attributes are contained in one or more nodes. In the tree, nodes that share a common attribute must form a connected subtree. If each tree node contained the attributes of a single relation then it would be a join tree.

We define the width of a GHD node v as the AGM bound of the join query over $\lambda^{-1}(v)$, in other words, the maximum number of tuples returned by the join query of all the attributes contained in the node. The Fractional Hypertree Width (FHTW) bound of a query is the minimum width of all the possible GHDs of the query.

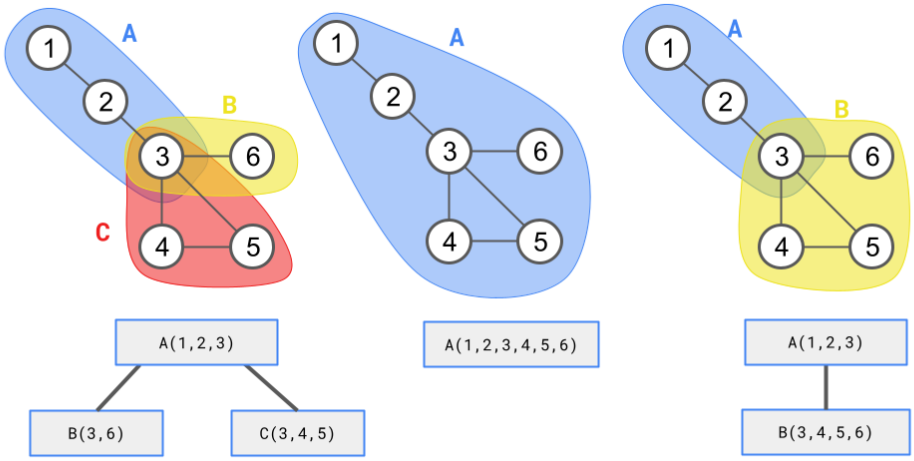


Figure 2.3: Different GHD options for a query pattern composed of six attributes.

2.4 EmptyHeaded

EmptyHeaded is a graph processing engine that proposed a novel join algorithm and query planner, based on GHD and tries [1]. Given a join query and its hypergraph representation, EmptyHeaded uses a heuristic to select a GHD for the query. This GHD is effectively a query plan, with each node in the GHD representing a multiway join that must be calculated. A worst-case optimal join algorithm is used on each node of the GHD, simplifying the query into a tree with the intermediate results as nodes. Yannakakis algorithm is then performed on the resulting acyclic query. This algorithm runs in time $O(N^w + |output|)$, where w is the width of the GHD used. This means the performance of EmptyHeaded is strongly tied to the

[1]The hypergraph does not fully capture the query, as it loses the order of the attributes in a relation.

selection of a query's GHD. Regardless of this, EmptyHeaded's algorithm complies with the AGM bound, and in some cases it can outperform this bound and instead be bounded by the FHTW bound [10]. In this work, we will refer to this algorithm as being "GHD-optimal".

EmptyHeaded stores triples as tries in main memory, and a recent set of experiments showed that it requires more space for indexing (in terms of bytes per triple) than almost all of the other alternatives it was compared to [3]. Also, even though EmptyHeaded offers fast runtimes, the experimental evidence suggests that its space requirements prevent it from performing well in real-world scenarios.

Chapter 3

The Qdag

In this chapter we introduce the fundamental concepts underlying this thesis and its related work. We begin defining the basic data structures used in succinct representations, then proceed to describe the data structure upon which our work is based: the Qdag.

3.1 Succinct Data Structures and Basic Operations

The field of succinct data structures aims to represent objects using space close to their information-theoretic lower bound while supporting operations efficiently. Central to many succinct representations is the bitmap or bitvector, defined as follows:

Definition 3.1.1 (Bitmap)**.** A bitmap $B[1, n]$ is a sequence of n bits, where $B[i] \in \{0, 1\}$ for $1 \leq i \leq n$.

Two essential operations on bitmaps are *rank* and *select*. For a bitmap $B[1, n]$ and $b \in \{0, 1\}$:

$rank_b(B, i)$ = the number of b's in $B[1, i]$

$select_b(B, i)$ = the position of the *i-th* b in B.

For example, take the bitmap $B = 00100110$. The operation $rank_1(B, 4) = 1$ because there is only one 1 in the first four bits, and $select_0(B, 3) = 4$ because the third 0 is in the fourth bit of the bitvector.

These operations can be implemented in constant time using additional data structures that require $o(n)$ bits of space, where n is the length of the bitmap [8]. Bitmaps are fundamental building blocks for many compact data structures because of their space efficiency and support for fast operations.

3.2 Region Quadtrees

A region quadtree (quadtree) is a hierarchical data structure designed to represent points in a two-dimensional grid of size $\ell \times \ell$, where ℓ is assumed to be a power of 2. The structure is defined recursively as follows:

Base cases:

- For $\ell = 1$: The quadtree is represented by a single bit, 1 if the cell contains a point, 0 if empty.
- For $\ell > 1$: If the grid contains no points, the quadtree is a leaf node.

Recursive case: When $\ell > 1$ and the grid contains points, the quadtree is an internal node with exactly four children, each representing one of the $\ell/2 \times l/2$ quadrants of the grid.

The position of a point (x, y) in a quadtree can be determined by interleaving the bits of x and y coordinates in their binary representation, creating a single value called the Morton code or Z-order value. The labels that trace the path from root to leaf form a unique sequence for each point in the grid.

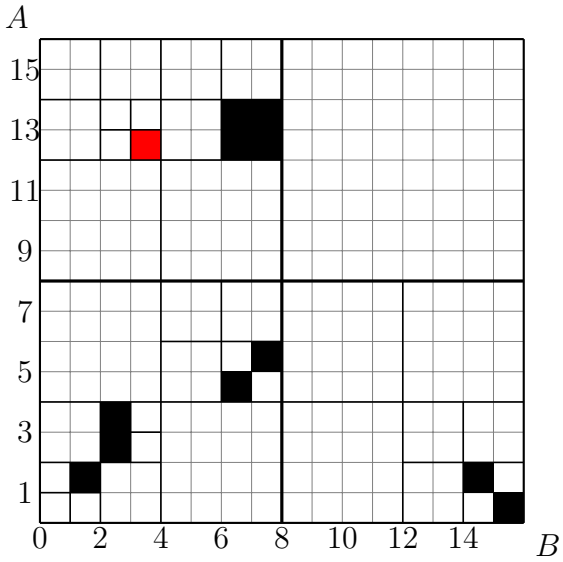
3.2.1 Generalization to Higher Dimensions

The definition of a quadtree can be generalized to represent points in higher dimensions. 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 $(l/2)^d$, and these subspaces are ordered following the Morton code. A leaf in the quadtree is either empty or represents one point.

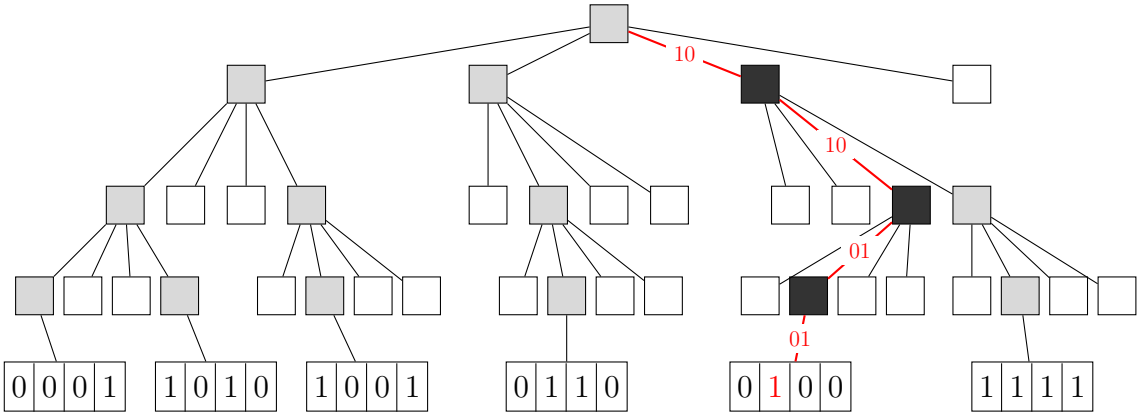
The Morton ordering is determined by interleaving the binary representations of the coordinates. For any point $(x_1, ..., x_d)$ in the d-dimensional space, we take the most significant bit of each coordinate and concatenate them to form a d-bit string. This string, interpreted as a binary number i where $0 \leq i < 2^d$, determines which child of the current node contains the point. This process repeats with subsequent bits until reaching a leaf node.

As with the 2-dimensional representation, each point in the structure is uniquely identified by the sequence of $\log(l)$ d-bit labels along the path from the root to its corresponding leaf.

A relation R with d attributes can be represented as a grid of points of dimension d. Each dimension corresponds to the domain of its corresponding attribute, and every point in the grid coincides with a tuple in the relation. By extension, a relation can also be represented as a quadtree. This is illustrated by Figure 3.1, which shows a grid and a quadtree representation of a relation.



(a) Grid representing the tuples in relation $R(A, B)$.



(b) Quadtree depicting the points of $R(A, B)$.

Figure 3.1: A quadtree representing the relation $R(A, B) = \{(0, 15), (1, 1), (1, 14), (2, 2), (3, 2), (4, 6), (5, 7), (12, 3), (12, 6), (12, 7), (13, 6), (13, 7)\}$. (a) Representation of R(A, B) in a $2^{2^d} \times 2^{2^d}$ grid. The tuples in R are represented by the black squares. (b) 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 using Morton code. For example, the highlighted path forms the string '**10100101**', the first dimension (A) is denoted by the bits in odd positions, while the second dimension (B) is at the even bits. Decoding the string results in the tuple being $(1100, \mathbf{0011}) = (12, 3)$, which correspond to the red square in (a).

3.2.2 Compact Representation

Quadtrees can be implemented as an array of bitvectors $B_0, B_1, ...B_n$ each representing a level of the tree. The number of levels in the tree is $log(\ell)$, where ℓ is the size of the relational domain. Each bitvector B_i stores the nodes at depth i as a concatenation of bits, with each node being 2^d bits long. The bits in a node represent its child nodes or, going back to the grid of points, a quadrant in its decomposition. If a bit is 1, then it means there are points in that child node, if it is a 0 then that subtree is empty. Levels only store non-empty nodes, so a level contains as many nodes as there were 1s in the previous level. This means that the j-th child of a node that starts at position n at level B_i is found at position $2^d \cdot rank_1(B_i, n+j)+1$ in B_{i+1}.

3.3 Qdag

A Qdag is a data structure that builds upon compact quadtrees to represent relations and, more importantly, evaluate multijoin queries in WCO time. This structure is composed of a quadtree representing a relation and a $O(2^d)$ mapping function that allows for the virtual traversal of the generalization of said quadtree to a higher dimension [9]. With this, we can extend a relation with new attributes and perform joins as intersections in a higher-dimensional space while only storing the original relations, and without calculating or storing intermediate join results.

To better understand this last point, take the relations $R(A, B) = \{(1, 2), (3, 5)\}$ and $S(B, C) = \{(2, 4), (1, 5)\}$, represented as Qdags. Using the Qdag multijoin algorithm, the join operation $R(A, B) \bowtie S(B, C)$ is actually calculated as $R'(A, B, C) \cap S'(A, B, C)$, with R' and S' being the relations R and S extended to the higher dimensional space $\{A, B, C\}$, so $R'(A, B, C) = \{(1, 2, 4), (1, 2, 5), (3, 5, 4), (3, 5, 5)\}$ and $S'(A, B, C) = \{(1, 2, 4), (3, 2, 4), (1, 1, 5), (3, 1, 5)\}$. The intersection of R' and S' is $\{(1, 2, 4)\}$, which is the result of $R(A, B) \bowtie S(B, C)$.

Recent experiments show that Qdags can even compress the database representation [9]. They also conclude that the Qdag index uses less space than other WCO algorithms. In particular, Qdags require around 250 times less space than EmptyHeaded. This gives Qdags the advantage of fitting higher up on the memory hierarchy, and thus being faster in some cases. Another benefit of the Qdag algorithm is that the output of a join query is obtained in its compressed quadtree representation, so further processing can be done to it, like using it for another query.

With regard to experimental runtime, Qdags are competitive in queries with low dimensionality. When producing relations of five or more attributes, they fare worse than other WCO algorithms.

3.3.1 Data Structure

Definition 3.3.1 (Qdag)**.** Let $Q_{d'}$ be a quadtree representing a relation with d' attributes. A Qdag Q_d for $d \geq d'$ is defined as a pair $(Q_{d'}, M)$, where $M : [0, 2^d - 1] \to [0, 2^{d'} - 1]$. The Qdag represents a quadtree Q (which is called the *completion* of $Q_{d'}$) according to the following rules:

1. If $Q_{d'}$ represents a single cell, then Q represents a single cell with identical content.
2. If $Q_{d'}$ represents a d'-dimensional grid empty of points, then Q represents a d-dimensional grid empty of points.
3. Otherwise, both $Q_{d'}$ and Q have internal root nodes. For each $0 \leq i < 2^d$, the i-th child of Q is the quadtree 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.

The Qdag represents the same relation R as its completion.

The quadtree component of the Qdag is implemented in a compact way, as explained in Subsection 3.2.2. The mapping function M of a Qdag is used to lift the quadtree's dimension without having to directly store all of its new nodes. Consider a point defined in d' dimensions as $(x_0, \ldots, x_{d'})$. When extending the dimension to d, we generate an augmented set of points by adding new coordinate values y_{d*} such that $y_{d*} \in \{0, \ldots, d\} \setminus \{0, \ldots, d'\}$, covering all possible combinations of these new dimensions. The mapping function indicates that the i-th child of a node in the Qdag Q_d points to the $M[i]$-th child of the corresponding node in the original quadtree $Q_{d'}$. To traverse the Qdag, we look at the node to which it originally corresponds with the mapping function. Note that attributes may be reordered or interleaved in the extension process. For example, a Qdag with attribute set $\mathcal{A}' = \{B, D\}$ may be extended to $\mathcal{A} = \{A, B, C, D\}$.

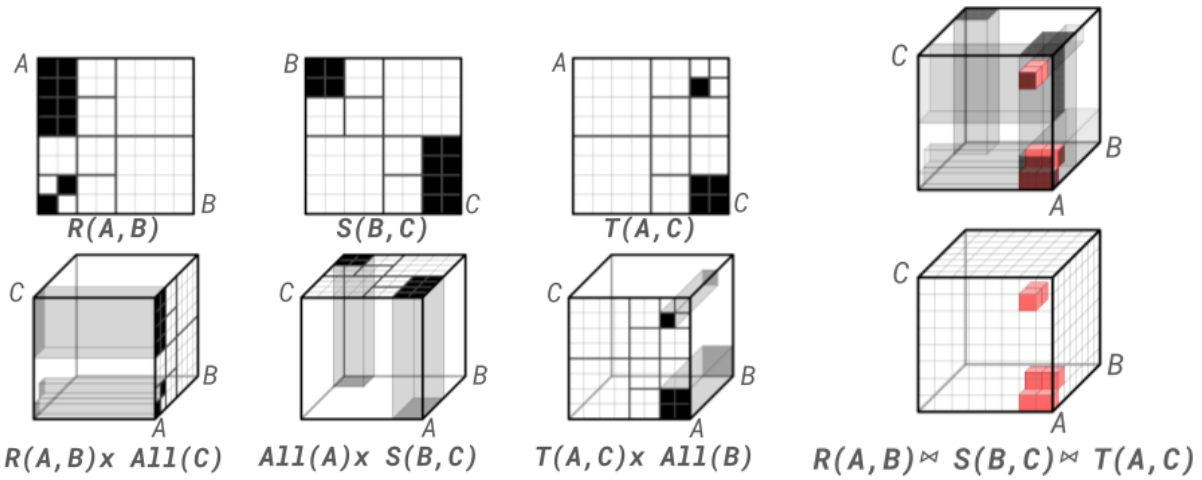
3.3.2 Original Join Algorithm

The Qdag proposes a WCO join algorithm we will refer to as multijoin. In general terms, the multijoin algorithm goes as follows. The Qdag representations of the participating relations are extended to all the attributes appearing in the query. The intersection of all the Qdags is computed, generating a quadtree that contains the values present in all the relations and thus the results of the join query. This is done by simulating a synchronized traversal along all the extended Qdags. A graphical representation of the multijoin process can be seen in Figure 3.2.

The key operations in the multijoin algorithm are `Extend` and `AND`, which will now be detailed.

Extending dimensions

The first step in the multijoin algorithm is to extend each relation to include all attributes in the query. When we extend a Qdag to include new attributes, we are moving from a



(a) (b)

Figure 3.2: An illustration of the multijoin algorithm for $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$. (a) shows how the relations are extended to another dimension and (b) depicts the intersection process.

space with a certain dimension d' to one of higher dimension d, but we need to maintain the relationships between nodes in both spaces. For instance, if we start with a relation containing attributes (A, B, D) and extend it to include C, creating (A, B, C, D), we are moving from a structure where each node has 8 children ($d' = 2^3$) to one where each node has 16 children ($d = 2^4$).

Consider a node in the extended space with Morton code '1010'. This code represents the tuples whose binary encoding will continue with a '1' for attributes A and C, and a '0' in B and D. If the corresponding node in the original space does not have children, then the extended node will not either. To check this we project the code by keeping only the bits corresponding to our original attributes (A, B, D). In this case, we get $M[1010] = 100$, because we keep the first, second, and fourth positions. Both Morton codes '1000' and '1010' in the extended space project to '100' in the original space, because they only differ in attribute C, which was not in our original relation.

The `Extend` operation, detailed in Algorithm 1, first computes the mapping function M and then creates a lookup table of size $O(2^{2^{d'}})$. This table contains one entry per possible $2^{d'}$ bit encoding, indicating the encoding of the corresponding set of children in dimension d. These two, M and the lookup table, allow jumping between dimensions d and d'. The first lowers a node's dimension and the latter lifts it by returning the node's encoding in the extended dimension; we refer to this operation as "materializing a node".

Algorithm 1 EXTEND $(Q, \mathcal{A})$

Require: A qdag $Q = (Q', M')$ representing a relation $R(\mathcal{A}')$, and a set $\mathcal{A}$ such that $\mathcal{A}' \subseteq \mathcal{A}$.
Ensure: A qdag (Q', M) whose completion represents the relation $R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$.

```
create array M[0, 2^d − 1]
d ← |A|, d' ← |A'|
for i ← 0, ..., 2^d' − 1 do
    m_d' ← the d'-bits binary representation of i
    m_d' ← the projection of m_d' to the positions in which the attributes of A' appear in A
    i' ← the value in [0, 2^d' − 1] corresponding to m_d'
    M[i] ← M'[i']
end for
return (Q', M)
```

Tree traversal and intersection

To navigate through a Qdag we use the operations VALUE and CHILD, shown in Algorithms 2 and 3. The first one indicates whether the subgrid represented by a Qdag is empty, a full single cell (tree leaf), or non-empty (internal tree node). CHILD allows us to descend through the i-th child of an internal node.

Algorithm 2 VALUE (Q)

Require: A qdag $Q = (Q', M)$ with grid side ℓ.
Ensure: The integer 1 if the grid is a single point, 0 if the grid is empty, and $\frac{1}{2}$ otherwise.

```
if ℓ = 1 then return the integer Q'
if Q' is a leaf then return 0
return 1/2
```

Algorithm 3 CHILD (Q, i)

Require: A qdag $Q = (Q', M)$ on a grid of dimension d and side ℓ, and a child number $0 \leq i < 2^d$. Assumes Q' is not a leaf or an integer.
Ensure: A qdag $Q_i = (Q'', M)$ corresponding to the i-th child of Q.

```
return (Q[M(i)], M)
```

After the Qdags are extended to the resulting dimension, the multijoin algorithm calls a recursive function called AND, both shown in Algorithms 5 and 4 respectively. This function is in charge of traversing the Qdags simultaneously, getting the intersection, and generating the output. Given the current tree level and the current node in each Qdag, the algorithm checks which children are present in every node and recurses only on them.

To traverse the Qdags, the algorithm keeps a list of the current nodes being visited on each Qdag's quadtree. The lookup table for each Qdag is used to materialize its rspective node, obtaining its encoding in the extended dimension. These encodings are intersected using the

bitwise-and operations (&), resulting in a bitvector of size 2^d whose active bits (set to '1') determine which children to recurse on.

The descent and pruning continue until the leaves of the tree 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 which should be '1' in the current level. The result Qdag is constructed from these lists.

Algorithm 4 AND $(Q_1, \ldots, Q_n)$

Require: n Qdags $Q_1, \ldots, Q_n$ representing relations $R_1(\mathcal{A}), \ldots, R_n(\mathcal{A})$.
Ensure: A quadtree representing the relation $\bigcap_{i=1}^{n} R_i(\mathcal{A})$.
1: $m \leftarrow \min\{\text{Value}(Q_1), \ldots, \text{Value}(Q_n)\}$
2: **if** $\ell = 1$ **then return** the integer m
3: **if** $m = 0$ **then return** a leaf
4: **for** $i \leftarrow 0, \ldots, 2^d - 1$ **do**
5: $\quad C_i \leftarrow \text{AND}(\text{Child}(Q_1, i), \ldots, \text{Child}(Q_n, i))$
6: **end for**
7: **if** $\max\{\text{Value}(C_0), \ldots, \text{Value}(C_{2^d-1})\} = 0$ **then return** a leaf
8: **return** a quadtree with children $C_0, \ldots, C_{2^d-1}$

Algorithm 5 MULTIJOIN $(R_1, \ldots, R_n)$

Require: Relations $R_1, \ldots, R_n$ stored as quadtrees $Q_1, \ldots, Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$.
Ensure: A quadtree representing the output of $J = R_1 \bowtie \cdots \bowtie R_n$.
1: **for** $i \leftarrow 1, \ldots, n$ **do**
2: $\quad$ Let Q_i be the qdag $(Q_i, \text{Id}(A_i))$
3: $\quad Q'_i \leftarrow \text{EXTEND}(Q_i, \mathcal{A})$
4: **end for**
5: **return** $\text{AND}(Q'_1, \ldots, Q'_n)$

3.3.3 Limitations

Although Qdags are competitive in queries with low dimensionality, when producing relations of five or more attributes they can be slower than other WCO algorithms [9]. Higher arity relations also significantly increase the amount of space needed to perform the join operation, due to the lookup tables used to materialize Qdag nodes. Since the table for a relation R is of size $O(2^{2^d})$, where d is the number of attributes in the original relation, if R has 5 attributes then the lookup table for just one participating relation would have 2^{32} entries.

Chapter 4

Higher arity Qdags

As stated in Chapter 3, a fundamental limitation in the original Qdag implementation was its constraint on the number of participating variables in join queries, which was restricted to a maximum of five. This significantly hampered the applicability of Qdags to complex real-world queries where joins may involve multiple variables across relations. For example, finding friend groups that are connected by a mutual friend in a social network.

Adding support for more complex joins required making changes to both the underlying data structures and the multijoin algorithm. In this chapter we present our first approach to enabling support for join queries of any number of variables in the Qdag structure; we call this Qdag variant the higher-arity qdag.

4.1 Multijoin algorithm

The high arity join algorithm works much in the same way as the original one: participating Qdags are extended to all attributes, they are virtually traversed simultaneously and we descend into the nodes' intersections. Here is where the main difference lies: the children through which we must recurse are no longer obtained by materializing the Qdag nodes and doing a bitwise `and` (&) operation between them. With higher arity join queries, extended Qdag nodes can get to be large and the previous strategy is no longer viable.

To obtain the children through which we must descend in a node, we now use an array of counters. Each counter represents a position in the resulting Qdag node, and keeps track of how many Qdags have a child in that position. At a given node, for each Qdag we obtain the positions of all its children in the node and increment the corresponding counters in the array. When a counter reaches the number of Qdags that are part of the join query, it means that all Qdags have a child in the bit corresponding to the counter, and thus should be part of the traversal.

4.2 Implementaion

4.2.1 Adding Bitmap Support for Large Nodes

In the original Qdag representation, each level of the quadtree is encoded as a bitmap. These bitmaps are internally implemented using an array of 64-bit words stored in an array. Each bit in the vector represents the presence or absence of a subtree at a particular position in the level.

Originally, the implementation imposed an upper limit on the size of individual nodes: a node could span at most 32 bits. This constraint ensured that all node representations fit entirely within a single 64-bit word of the bitmap, simplifying encoding and query logic.

In order to support higher-dimensional queries, where $k^d > 64$, it became necessary to lift this restriction. In the new approach, nodes are allowed to span multiple 64-bit chunks in the array. This required revising the encoding logic to correctly handle nodes that cross word boundaries to account for multi-word node representations. Figure 4.1 shows an example of the bitmap representing a level in a Qdag with 7 attributes, in which every node would be composed of $2^d = 2^7 = 128$ bits. In this case a node would be represented by two contiguous 64-bit words, and if we were to materialize it, we would have to retrieve both of these.

This extension enables the data structure to accommodate arbitrarily large values of 2^d, so it supports a wider range of queries than the original Qdag.

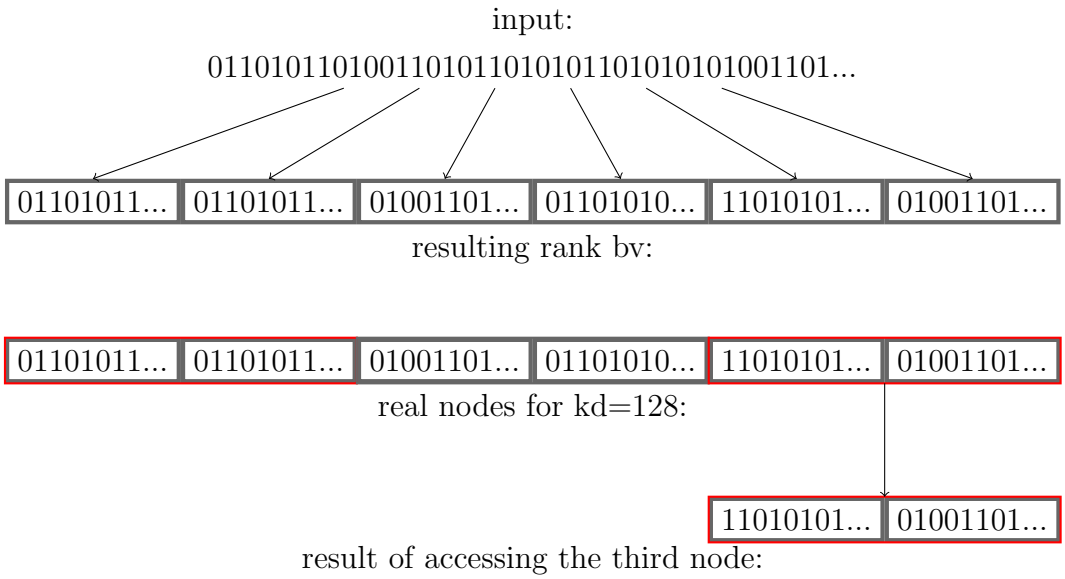


Figure 4.1: Example of the bitvector of a qdag with 7 attributes ($2^d = 128$). A qdag node spans two 64-bit words in the bitvector.

4.2.2 Extending dimensions

The original Qdag process of extending dimensions consisted of creating a lookup table and the mapping function M, which allow jumping between dimensions within a Qdag. Creating a lookup table is no longer feasible in higher-arity join queries, since the space required to store them grows exponentially with dimension, $O(2^{(2^d)})$. In high-arity Qdags, instead of materializing nodes and working with bitwise operations, we work with the indices of the children within a node, i.e. the position of the bits set to 1.

We replaced the lookup tables with a mapping table M', which maintains a one-to-many relationship between the children of the original Qdag and its extension. For each index i corresponding to a child in the original Qdag, $M'[i]$ returns a list of positions j such that $M[j] = i$. In other words, $M'[i]$ indicates which children of the extended node depend on child i of the original Qdag. In this way, we use M to answer the question: which original Qdag position (bit) resulted in a given position in an extended node? And we use M' to answer: which positions does this bit in the original node transform to in the higher dimension?

For example, take Qdag Q that represents the relation $R(A, C)$. If we were to extend Q to incorporate the attribute B to represent $R'(A, B, C)$, we would generate the tables M' and M as shown in Tables 4.1 and 4.2.

index in d'	**indices in d**
0	0, 2
1	1, 3
2	4, 6
3	5, 7

Table 4.1: M' table used to extend Q from $\{A, C\}$ to $\{A, B, C\}$. The bit at the index 1 in the original dimensional space $\{A, C\}$ maps to the positions 1 and 3 in the space $\{A, B, C\}$. The bitstring '00' turns into both '010' and '000' in the extended space.

index in d	**index in d'**
0	0
1	1
2	0
3	1
4	2
5	3
6	2
7	3

Table 4.2: M table used to go from the space $\{A, B, C\}$ to $\{A, C\}$. A bit at position 2 in the extended space maps to the 0 position in the original dimension, as does a bit at the 0 position.

This mapping enables efficient traversal of the extended dimensional space by providing direct

access to relevant positions without having an entry for each possible node in the original Qdag. Instead, we keep an entry for each dimension in the original Qdag. This means that if $d = 2$, instead of having three separate entries for "0011", "0001", and "0010", we keep entries for the indices "2" and "3". The space complexity of this auxiliary structure is proportional to 2^d rather than 2^{2^d}, making it feasible even for high-dimensional joins.

4.2.3 Traversal and Intersection

As in the original multijoin algorithm, the traversal and intersection of the trees are performed through the recursive function `And`. At each tree level, the algorithm looks at the current node in every Qdag to determine which child nodes are common to all of them and then proceeds to recursively process only those mutually existing children.

Intersecting nodes

As stated in Section 4.1, we utilize an array of counters to intersect the nodes and determine which children to recurse on. This array has at most as many counters as the arity of the join query (2^d). For each node of the participating Qdags we obtain the positions of their bits that are set to 1. Taking a bit with index i within a node in a Qdag, for each element j in $M'[i]$, we increment the counter located at $C[j]$ by one. If at any moment $C[j] == n$, with n being the number of Qdags participating in the join, then we must continue descending through the jth child in every Qdag, so it is added to the list of children to recurse. The intersection condition is checked each time a counter is incremented to avoid traversing the entirety of C, which is of size 2^d. In this way, the work done is proportional to the size of possible join results.

Using a traditional array to store the counters is not ideal since initializing each one in 0 would require a time proportional to 2^d. To reduce the cost of building C, we use what we refer to as an "initializable array". This data structure models an array of counters that only initializes a counter when it is accessed for the first time. This means that we only initialize counters that represent positions that could potentially yield a join result.

Figure 4.2 exemplifies the intersection process of three nodes using M' and C.

Obtaining child indices

A key part of the new intersection method is that instead of materializing a node, we obtain the indices of the bits that are set to 1. Given that nodes in higher-arity joins can have high dimensions, iterating through all $k^{d'}$ bits to find set bits would be inefficient. We added a function called `select_next` to the existing bitmap structure to directly identify the set bits. For a bitmap $B[1, n]$ and $b \in \{0, 1\}$:

$select\_next(B, i)$ = index of the first $j \geq i$ for which $B[j] = 1$

M'_R

0	0, 1
1	2, 3
2	4, 5
3	6, 7

M'_S

0	0, 4
1	1, 5
2	2, 6
3	3, 7

M'_T

0	0, 2
1	1, 3
2	4, 6
3	5, 7

Node in R: **0100**

$M'_R[R \to select\_next(0100)] = M'_R[1] = \{2, 3\}$

$C[2] = 0 + 1$

$C[3] = 0 + 1$

Node in S: **0101**

$M'_S[S \to select\_next(0101)] = M'_S[1] = 1, 5$

$C[1] = 0 + 1$

$C[3] = 1 + 1$

$M'_S[S \to select\_next(0001)] = M'_S[3] = \{3, 7\}$

$C[5] = 0 + 1$

$C[7] = 0 + 1$

Node in T: **0110**

$M'_T[T \to select\_next(0110)] = M'_T[1] = 1, 3$

$C[1] = 1 + 1$

$C[3] = 2 + 1$

$M'_T[T \to select\_next(0010)] = M'_T[2] = \{4, 6\}$

$C[4] = 0 + 1$

$C[6] = 0 + 1$

$C = [-, 2, -, \mathbf{3}, 1, 1, 1, 1]$

Figure 4.2: Example of intersecting nodes from relations R, S, and T. In C, dashes symbolize that a counter was not initialized. After processing the third Qdag, $C[3] = 3$, which is the number of relations, so the descent would continue down the node at position 3 in the extended dimension. The mapping function M of each Qdag is then used to determine the equivalent node in every relation, for example in Qdag T we would descend on $M_T[3] = 1$.

A node can be made up of multiple 64-bit "words". To obtain the next set bit, we use a bit mask in the current word (the one i is a part of) to isolate bits from index i onward, and return the index of the least significant bit within the word. If the masked word results in 0, then the next set bit is not in this word and we must scan subsequent words until we find one that is non-zero. We then retrieve the position of the least significant set bit in that word.

Chapter 5

GHD-Based Algorithm

Experimental results show that there are some graph patterns in which the Qdag does not perform as well as other systems. As we discussed in the previous chapters, Qdag's WCO join algorithm is the most efficient when queries have up to three attributes, but it is not always competitive in queries of 4 or 5 attributes. Also, the WCO algorithms tested by Arroyuelo et al. were generally slower than traditional ones when searching for "star patterns", which are comprised of a central node that is related to three or more other nodes [9].

Nevertheless, Qdags obtain competitive time performance while requiring much less space than the alternatives. EmptyHeaded beats the Qdag in execution time when searching for certain patterns, but at a very high cost in terms of memory usage. This is due to the production of large intermediate results during the execution of the algorithm.

In this chapter we explore the possibility of improving Qdags' performance in more complex queries, particularly in query patterns that contain more than 3 attribute nodes. We propose a new join strategy inspired by EmptyHeaded's approach, that leverages Qdags' success with lower-dimensional queries. We provide implementations using both the original dimensionally-restricted Qdags and the Qdag variant introduced in Chapter 4. First we describe the algorithm in general, then we share its implementation details, including the necessary modifications made to Qdags. Finally, we describe the algorithm execution process.

5.1 Proposed Algorithm

The general idea of our proposed algorithm is to, as in EmptyHeaded, use a GHD of the initial query as a query plan. That is, transform the (possibly cyclic) query graph into a tree in which the nodes represent sub-queries, including cyclic joins. The sub-queries are then solved using Qdag's multijoin algorithm as explained in Section 3.3.2. This yields a join tree of Qdags, on which Yannakakis algorithm for acyclic join queries is used to obtain the final results. Figure 5.1 shows a diagram of our proposed strategy.

Qdags' high performance in low-dimension join queries and the fact that they are composi-

tional (the output of the join query is also a Qdag) make them an ideal candidate for solving the GHD nodes. This would address EmptyHeaded's high memory usage while potentially improving query times.

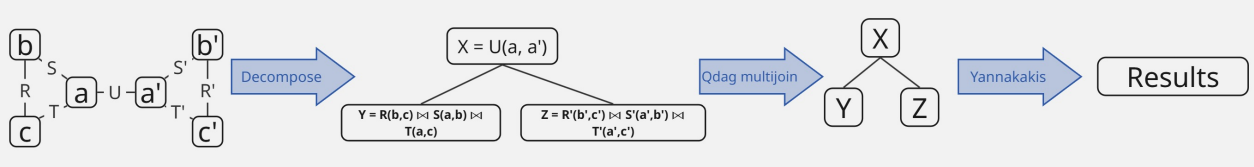


Figure 5.1: Diagram depicting our proposed strategy. In this example the query is a join between 6 relations, R has the attributes b and c, S the attributes a and b, and so on. The cyclic query is decomposed into a GHD with three nodes, each representing a join between relations. These joins are solved using Qdag's algorithm, obtaining a join tree with three nodes. Finally, Yannakakis algorithm is used to get the query results.

5.2 Semijoin support in Qdags

In this section we detail the addition of the semijoin operation to the Qdag, needed to perform Yannakakis' join algorithm. The essence of the algorithm is the same for both of our implementations, but there are differences that will be detailed in the end of this section.

The semijoin operation is a key component of Yannakakis' algorithm for solving acyclic join queries. A semijoin between relations R and S, denoted as $R \ltimes S$, produces the subset of tuples from R that participate in the join with S, preserving only the attributes of R. In other words, it reduces relations by filtering out tuples that would not contribute to a join result while preserving the original schema structure of the filtered relation. It can be expressed as $R \ltimes S = \pi_R(R \bowtie S)$, where $\bowtie$ represents the natural join operation and π_R denotes projection onto the attributes of R. The semijoin operation can be done between multiple relations; in this thesis we implement a semijoin operator that operates across multiple relations but only constrains the leftmost relation. This enables us to filter a target relation against multiple constraint relations simultaneously, and only tuples from the leftmost relation that have matching values in all subsequent relations are preserved in the result set. For example, consider the relations $R(a,b) = \{(1,2),(3,2),(4,5)\}$, $S(b,c) = \{(2,2),(2,6),(8,3)\}$, $T(a,d)\{(1,7),(4,3),(1,5)\}$. The operation $R \ltimes (S,T)$ would yield the result $\{(1,2)\}$.

As explained in Section 3.3.2, extending the dimension of a Qdag creates many "virtual tuples". Say we have the relations $R(a,b)$ and $S(b,c)$. When doing a join, we would want to retrieve all matches resulting from $R = \{(1,2),(3,2),(4,5)\}$ and $S = \{(2,7),(2,8),(4,9)\}$, which would produce the join result $R \bowtie S = \{(1,2,7),(1,2,8),(3,2,7),(3,2,8)\}$, but in a semijoin, we do not need to materialize these results. If a tuple from R is part of a match, we just want to retrieve or preserve that original tuple. In our example, computing $R \ltimes S$ would yield $\{(1,2),(3,2)\}$ because those tuples in R have matching values in S. Doing a semijoin instead of a standard join also avoids creating an additional output Qdag structure, since we can modify the constrained Qdag to reflect the results.

5.2.1 Active bitmap

In our semijoin algorithm, when a match is found during the traversal of the participating Qdags, instead of creating a new Qdag with the resulting tuples, we mark the corresponding node in the original Qdag that is being constrained. To implement this, we added an additional bitmap array to the quadtree structure, called `active`. Like with the original node bitmaps, we maintain one `active` bitmap per level of the quadtree. The `active` bitmap indicates which bits of the original quadtree (each representing a tuple) are currently 'active' in the relation—in other words, which tuples have not been filtered out through a semijoin operation. If a bit is set in the original level bitmap but it is not set in `active`, then it is as if it were not set in either. Only bits that are set to 1 in both bitmaps are considered to have a subtree when traversing a Qdag.

Initially, when creating the Qdag, the active bitmap contains only 1s because all tuples in the relation exist and must be considered. Later, when performing a semijoin, some tuples are filtered out and their corresponding bits are unset. It is important to note that the active bitmap maintains the same dimensions as the node bitmap for each level, with a one-to-one bitwise correspondence. This ensures that each bit in the active bitmap corresponds directly to the same position in the node bitmap.

The original bitmap cannot be modified because it is static and is used for tree navigation, maintaining node order, and rank operations. The quadtree and node configuration were generated based on specific tuples. Removing tuples could alter the tree structure and would require recalculating all values, which would be costly considering the amount of semijoins that are done in our proposed algorithm. Therefore, the active bitmap approach allows us to logically filter tuples without physically modifying the underlying data structure.

It is important to note that the addition of this bitmap affects the multijoin algorithm and general materialization of Qdags as well. Whenever a node is materialized we must check its `active` bitmap. If a bit is not marked in the `active` bitmap, the corresponding tuple will be excluded from query processing without physically removing it from the data structure.

When traversing Qdags during a multijoin, we only descend through bits that are set in `active`. In the case of high arity Qdags, this means that the `select_next` method we described in Section 4.2.3 will operate on `active`, obtaining the set bits that are still active in the Qdag.

In the original Qdag variant, which works with materialized nodes, the series of 2^d bits that form a node are now computed as a bitwise *and* operation (&) between the original node bitmap and its corresponding `active` bitmap of the quadtree, in the initial dimension d'. For example, if the original node has bits 1101 and the `active` bitmap has 1100, the resulting node would be the materialization of 1100 using the lookup table, effectively filtering out the third tuple.

5.2.2 Tree traversal

The semijoin algorithm works similarly as the original multijoin: all the participating Qdags are extended to the query dimension and then a recursive function is used to virtually traverse and intersect the trees. The extension method depends on the Qdag variant used. When the semijoin is performed on the original Qdag variant, the mapping function and auxiliary lookup tables are created. In the case of high arity Qdags, the two mappings are calculated.

The intersection and traversal of the Qdags is performed by a recursive function called `semiAnd`. This function descends through the participating Qdags following the same traversal strategy employed in the `Multijoin` algorithm, applying the appropriate method for each variant. When the algorithm identifies a join result during the traversal, it updates the active bitmap of the leftmost Qdag by marking the bits that correspond to the matching tuple at each level of the tree. The specific bit positions to mark are determined using the mapping function M, which translates the indices of the extended dimensional space (d) back to the original dimension of the leftmost relation (d').

5.2.3 Pruning

As stated previously, multiple join results can originate from the same tuple in the constrained relation. Since we are doing a semijoin, how many matches a tuple is a part of, or what they are, is not relevant; we just want to know which tuples in the constrained relation are part of any matches. Generating all possible matches would be unnecessary work, as it would require traversing the same subtree to mark the same tuple (bit) multiple times without providing additional information. An ideal optimization would be to stop checking a node once we determine that all its bits and its descending subtree will survive the semijoin. This pruning prevents redundant computation by avoiding the re-exploration of subtrees that have already been processed and determined as satisfying the join conditions, translating into significant time savings, particularly when processing dense datasets.

To implement this optimization, we maintain a temporary bitmap throughout the semijoin operation that both tracks the results at the leaf level and indicates internal nodes with completed subtrees. When traversing a set of relations, if we encounter a node from the constrained relation (the leftmost Qdag in the semijoin) that is already marked in this temporary bitmap, we treat it as a leaf node and avoid descending further into its subtree. We mark internal nodes during the same recursion that traverses the relations during the intersection. If, upon returning from the recursive call, all 2^d children of the current node, v, are marked in the temporary bitmap, we also mark the current node. Then, when returning via another branch and reaching another node that corresponds to the same node v in the constrained relation, it will already be marked.

Once the intersection traversal is complete, we recursively traverse the temporary bitmap to propagate the 1s from the leaves bottom-up to all their ancestors. In this way, we rebuild the path from root to leaf of all the results.

After this propagation, we perform a bitwise AND (&) between each level of `active` and the

temporary bitmap, updating the former. The final result is a quadtree with an accompanying bitmap that efficiently represents which tuples in the original relation participate in joins with other relations, without the need to materialize all possible join combinations.

Skipping marked bits

As stated previously, when descending through the Qdags we skip the subtrees that are already marked in the temporary bitmap. The implementation of this optimization occurs during the child selection phase of the recursive tree traversal. When determining which children nodes to recurse into, the algorithm consults the temporary bitmap to identify and skip bits that are already marked as processed. This selective traversal can significantly reduce the computational overhead in subsequent join operations within the same multijoin or semijoin execution.

Original Qdag Variant

For the original Qdag variant, this is implemented through bitwise operations. When selecting children to recurse in a given node, the algorithm performs a bitwise AND operation between the left node and the negation of the corresponding node in the temporary bitmap, in the original dimension. This bitwise operation produces a result that indicates which bits remain to be explored in the left Qdag subtree, filtering out the already-processed regions. This operation can be expressed as:

$$remaining\_bits = left\_Qdag\_node \,\&\, \sim temporary\_bitmap\_node$$

Once the filtered result is obtained, the algorithm materializes these remaining bits and performs an intersection with the materialized nodes from the other Qdags participating in the semijoin operation. This intersection ensures that only the unprocessed bits from the leftmost Qdag that are present across all Qdags are considered for further traversal. It is important to note that the filtering of the processed bits is done in the original dimension of the leftmost Qdag, not the extended dimension.

High Arity Variant

In the case of high-arity Qdags, the implementation differs significantly as nodes are not materialized during traversal. While the algorithm still uses the temporary bitmap to mark results and completed subtrees, it uses the array of counters C, introduced in Section 4.2.3, to avoid traversing though completed subtrees.

To mark a subtree as fully processed, the corresponding value in the counter array C is set to $n + 1$, where n represents the number of relations participating in the semijoin operation, as shown in Algorithm 6. This marking strategy is effective because the traversal algorithm only descends through nodes whose counter value is exactly n. By setting the counter to $n + 1$, the algorithm ensures that marked subtrees are never revisited during subsequent traversal phases.

Algorithm 6 Filter Children for traversal during High-Arity Semijoin

Require: $level$: current tree level, $node$: current node identifier
Require: C: counter array, n: number of relations in join
Require: M': mapping from children to element sets
Require: Q: Qdag structure, $result$: bitmap for filtering

```
1: procedure FilterChildren(level, node, C, n, M', Q, result)
2:     children_array ← empty array of size k_d
3:     n_children ← 0
4:     Q.get_children_result(level, node, children_array, n_children, result)
5:     for i = 0 to n_children − 1 do
6:         cur_child ← children_array[i]
7:         size ← |M'[cur_child]|                ▷ Size of element set for current child
8:         for j = 0 to size − 1 do
9:             element ← M'[cur_child][j]        ▷ Get j-th element from child's set
10:            C[element] ← n + 1                ▷ Mark subtree as complete
11:        end for
12:    end for
13: end procedure
```

5.3 Algorithm Implementation

5.3.1 Handling Generalized Hypertree Decompositions

To support generalized hypertree decompositions (GHDs), we implemented a tree structure. Each GHD node contains a list of Qdags corresponding to the relations assigned to that node in the decomposition, along with pointers to its child nodes. For the purposes of this work, we assume that the decomposition process has been performed manually, as automatic query decomposition falls outside the scope of this thesis. Beyond standard accessor and setter methods for managing children and relations, the GHD class provides methods for executing join operations across the decomposition structure.

Core Operations

The GHD implementation supports the three primary operations necessary for Yannakakis' algorithm on acyclic queries:

- `exec_multijoin`: This method executes the multijoin operation between all Qdags contained within the node.
- `constrained_by_children`: This recursive function does a bottom-up traversal of the tree, performing a semijoin that constrains the Qdag in the current node based on the Qdags of its child nodes.

- `constrain_children`: This operation does a top down traversal of the tree, in which at every node it iterates through each child node and performs a semijoin between the child's Qdag and the current node's Qdag to constrain the child relation.

5.3.2 Algorithm Execution

In this section we go through each phase of the algorithm's execution, from the initial construction of data structures through the final result computation. This is also shown in Algorithm 7.

Initialization Phase The algorithm begins by reading the input data and constructing the necessary data structures. Qdag representations are created for all input relations and then a GHD is constructed manually. Once the initialization is complete, the algorithm is invoked on the root node of the GHD.

Node Reduction Each GHD node is reduced by calling the method `exec_multijoin`, replacing its list of relations for the resulting Qdag. This function supports both recursive and parallel execution. In the first, each node sequentially passes the `exec_multijoin` operation to its children. In the parallel version, multiple nodes can be processed concurrently by different threads. After this phase, a GHD node only contains one Qdag (relation) and a list of child pointers, resulting in a simplified query tree in which each node contains the intermediate results of its corresponding subquery.

Constraint Propagation After the simplified join tree is obtained, the Yannakakis algorithm begins. This phase consists of two operations. First, the `constrained_by_children` method performs a bottom-up traversal of the tree, executing one semijoin per node to propagate constraints from leaves toward the root. During this traversal, the Qdag of each node is constrained by the Qdag of its children, eliminating its tuples that cannot contribute to the final result. Following this, `constrain_children` is invoked and does a top-down traversal of the tree. In this step, each already-reduced Qdag node filters the tuples of its children's Qdags by performing semijoins.

Final Join The algorithm performs a final join operation using the Qdag multijoin algorithm on all nodes in the simplified tree structure. Since the constraint propagation phases have eliminated non-contributing tuples, this final join operates on significantly reduced intermediate results while still producing the complete query answer represented as a Qdag.

Algorithm 7 GHD-Optimal Join Algorithm - serial version

Require: root: root node of the GHD tree
Ensure: qResult: Qdag containing the join result

```
1: function YANNAKAKIS(root)                    ▷ Execute multijoin on all tree levels
2:     root.DEEPEXECMULTIJOIN                   ▷ Bottom-up constraint propagation
3:     root.CONSTRAINEDBYCHILDREN               ▷ Top-down constraint propagation
4:     root.CONSTRAINCHILDREN                   ▷ Final multijoin to obtain query result
5:     crossProduct ← All Qdags in tree
6:     qResult ← MULTIJOIN(crossProduct)
7:     return qResult
8: end function
```

Chapter 6

Experimental Evaluation

In this chapter we share the experimental evaluation of our proposed algorithms, comparing them to the original Qdags both in terms of memory usage and query time. First, we describe the general methodology used for assessing algorithmic performance. Then, we share how our proposed GHD-optimal algorithm compares to the original Qdag multijoin in both high-arity and regular Qdags.

All of our experiments ran on an AMD EPYC 7343 CPU at 1.5 GHz, with 32 cores and 64 threads, 32 MB of cache, and 1 TB of RAM. Our source code was compiled using g++ with flags –std=c++17, -O3, and -fopenmp in the case of the GHD-optimal algorithm.

6.1 Methodology

To evaluate the work proposed in this thesis, we adopted the methodology used to assess the Qdag upon its publication [9] with some minor modifications. We measured the performance of our algorithms when querying for patterns on a subgraph of the Wikidata database provided by the Wikidata Graph Pattern Benchmark [7].

The graph contains 81,426,573 RDF triples (subject, predicate, object), which use 2,101 predicates. We store the information as a relational database in which a relation corresponds to a predicate, so we have 2,101 "tables" in total. The predicate tables store the pairs of subjects and objects that are connected through that predicate and form a triple on the graph, as depicted in Figure 6.1. More formally, the relation p_i contains every pair (s, o) such that the triple (s, p_i, o) exists in the graph. This approach limits the kinds of queries that can be answered. For example, if we wanted to know all the predicates that link a given subject and object, we would have to query every predicate table. In a single-table approach that stores all the triples as (subject, predicate, object), this would be a straightforward query.

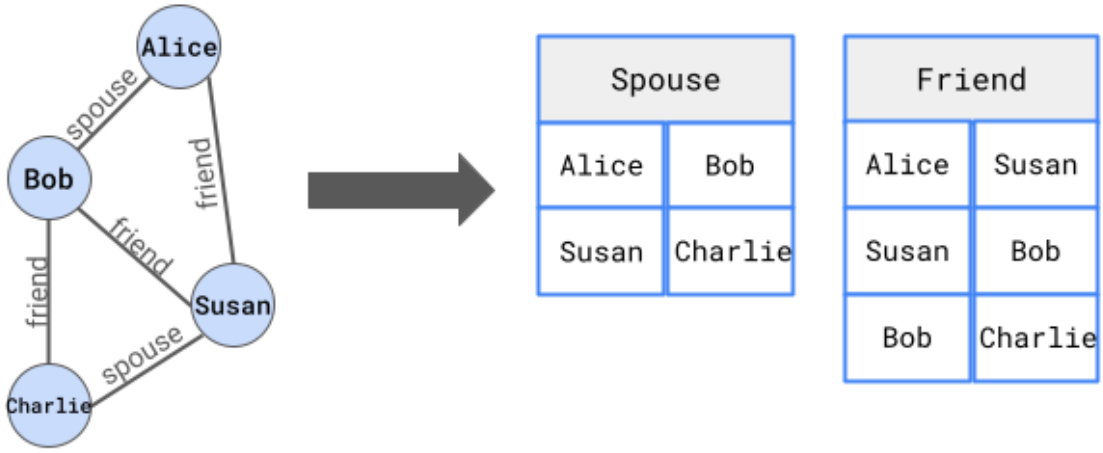


Figure 6.1: A graph showing social relationships between people transformed into the tables "Spouse" and "Friend".

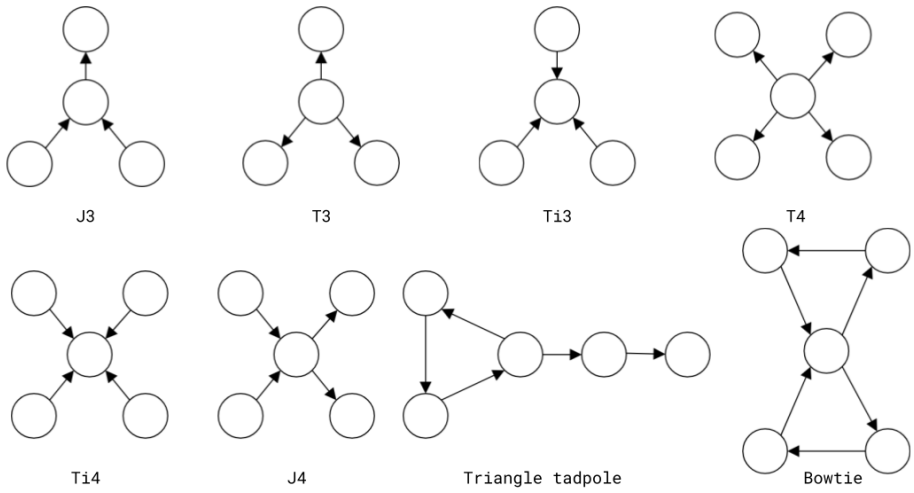
6.1.1 Experimental setup

We evaluated the different Qdag variants using 12 distinct query patterns, as illustrated in Figure 6.2. For each pattern, we executed 50 queries using the Wikidata predicates described in the previous section. Six of these query patterns were selected from the Wikidata Graph Pattern Benchmark [7], specifically choosing the patterns that previously demonstrated the poorest performance for Qdags [9]. The remaining six patterns consist of cyclic queries that had not been previously tested on Qdags, allowing us to assess performance on more complex queries. Some query patterns exceed the original Qdag limitation of 5 attributes, so they are only used to assess higher-arity Qdags.

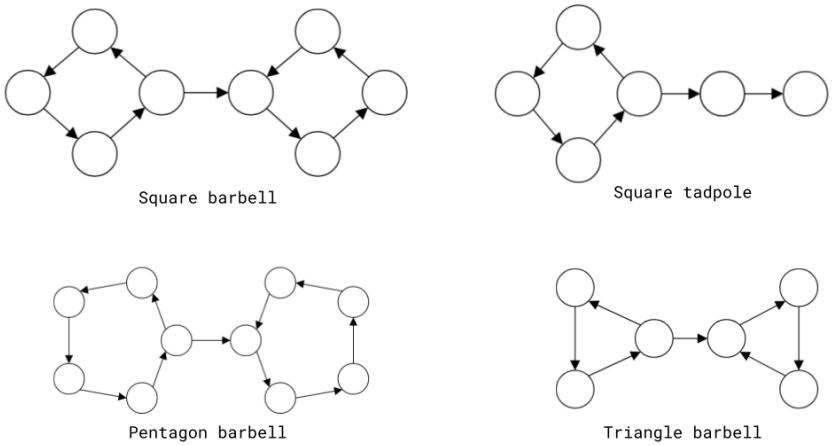
The experimental data was generated following the methodology established by the Wikidata Graph Pattern Benchmark. First, we deployed the Wikidata subgraph on an Apache Jena server, then systematically queried the database to identify 50 distinct predicate combinations for each pattern that would produce non-empty results [1]. Since cycles represent the most computationally expensive components of query processing, we implemented additional controls to ensure experimental diversity: we limited the frequency with which any given set of predicates could form the same cycle structure across different queries. This approach prevented any single predicate combination from dominating the results, which is important because some cycles process much faster or slower than others, and repeated inclusion of the same high-performing or low-performing cycle would skew the overall performance measurements.

Patterns can be decomposed in different ways, including grouping all relations in a single multijoin. To test how the GHD of a query affects its execution time, we compared the performance of the join algorithm using different GHDs on queries J3, T3, Ti3, J4, T4, and Ti4. Figure 6.3 shows how the relations in these queries were grouped in each decomposition. Although some decompositions appear to be the same, in practice they are different due to the data of the relations. For example configurations 1 and 3 in 6.3a are both schemes in

[1]The code is publicly available at https://github.com/matildeRivas/query-pattern-finder



(a) Patterns queried with both Qdag variants.



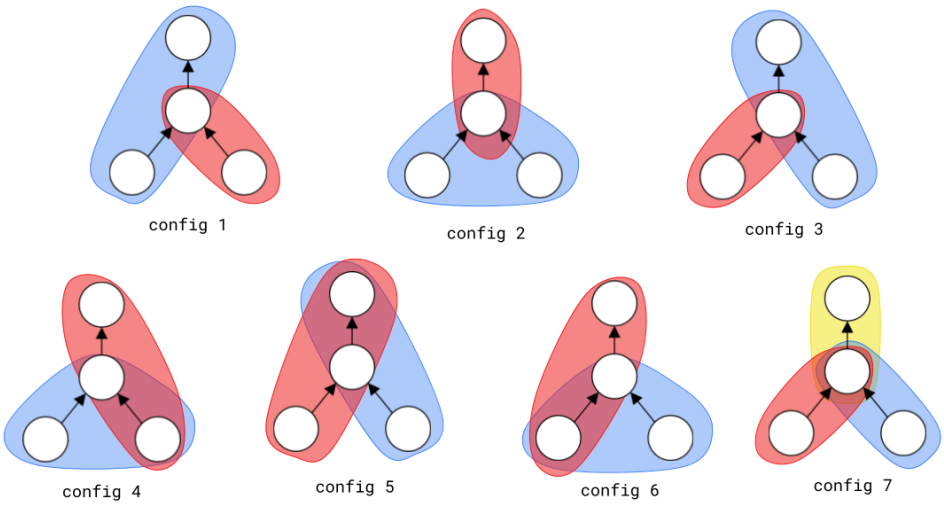
(b) Patterns queried with high arity Qdags.

Figure 6.2: The query patterns used in the experiments.

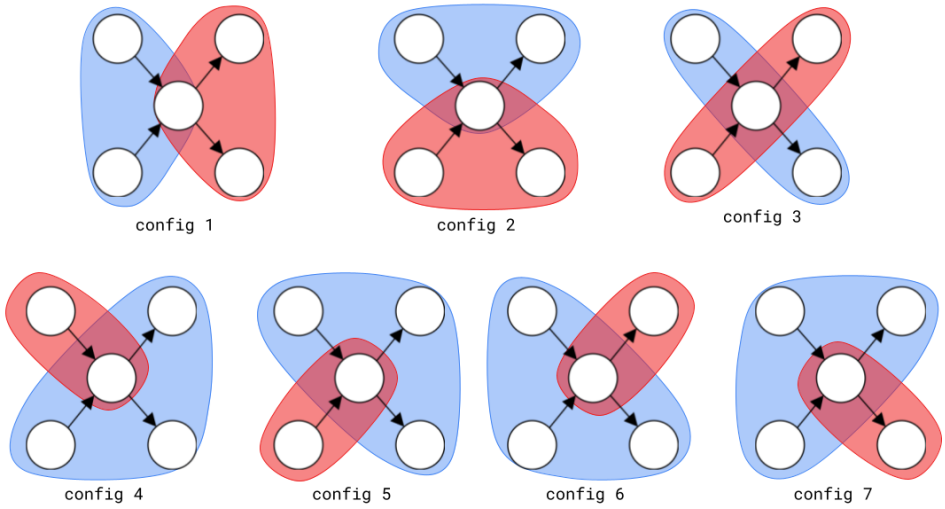
Pattern	J3	J4	T3	Ti3	T4	Ti4
Average	1131.18	292.78	29.6	203545.22	1304.88	2248722.76
Median	5.0	4.0	2.0	2.5	2.0	3.0
Pattern	triangle tp	square tp	bowtie	triangle bb	square bb	pentagon bb
Average	5.7	4.42	6.38	2.92	8.54	20.88
Median	1.0	2.0	1.0	1.0	2.0	2.0

Table 6.1: Average and median number of results per query pattern used in the experiments.

which a path of two relations is in one GHD node, and the remaining relation is in another one. In the abstract these are both the same decomposition, but for a given set of three relations, the two GHDs may yield very different runtimes.



(a)



(b)

Figure 6.3: Query decompositions tested in patterns made up of (a) three attributes (J3, T3, Ti3) and (b) four attributes (J4, T4, Ti4).

6.2 Results and Discussion

6.2.1 Space usage

Table 6.2 shows the space used to store the input relations in bytes per tuple. The Qdag variant modified for the GHD-based algorithm uses almost 50% more space than the original Qdag (published by Arroyuelo et al. [9]). This is because our new Qdags have an additional bitmap `active`, which is dimensionally the same as the bitmap used to store the quadtree. Even with this increase in space usage, Qdags still drastically outperform other systems in this regard. In comparison, for the Wikidata benchmark Emptyheaded uses 1292.28 bytes per tuple and Jena 48.42 bytes per tuple [9].

Qdag variant	Average Space
original	6.76
new	9.97

Table 6.2: Index space in bytes per tuple.

The GHD-optimal algorithm produces intermediate results which the original multijoin does not. The reduction step of the algorithm, in which the multijoin is applied to every node in the GHD, generates a new Qdag representing the join result of each node. Appendix A contains the size in bytes of these intermediate results for each query, computed using the high arity Qdag. Results that we specifically refer to in this discussion are highlighted in the Appendix.

The size of these intermediate results varies on the query decomposition. While in the great majority of cases the first multijoin pass reduces the size of the working data in orders of magnitude, in some decompositions the multijoin produces intermediate results that are larger than the original input data. For example, the participating relations in one specific Ti4 query were initially stored in $3,051,200$ bytes (for 624,395 total tuples). The three decompositions we tried yielded the following intermediate result sizes, respectively: 40,548, 16,305,972 and 390,516; the size of the final join results was 16,746 bytes. In this particular case, the second decomposition greatly increased the size of the working data, producing intermediate results that are 100 times larger than the final join results.

Tables 6.3 and 6.4 show the ratio between input size and result size, and intermediate result size and result size. These values were calculated using the decomposition that performed the best in terms of query execution time.

Table 6.3: Space usage factors for input relations and intermediate results relative to final result size when using the original Qdag across different query patterns.

Pattern	Avg input	Median input	Avg intermediate	Median intermediate
J3	567.49	280.63	81.23	21.73
T3	726.91	453.81	63.86	33.21
Ti3	241.82	95.26	59.12	16.89
J4	830.73	468.62	39.13	4.35
T4	1247.34	1073.23	63.7	7.81
Ti4	503.59	216.17	243.84	8.0
triangle tadpole	1419.76	1037.65	244.87	52.74
bowtie	2096.97	1954.46	52.19	1.84

In the next section we will further analyze how these intermediate results relate to the total query execution time.

6.2.2 Query decompositions

As can be seen in Figures 6.4 and 6.5, the chosen decomposition of a query directly affects its performance. While the average query times for a given pattern appear similar across

Table 6.4: Space usage factors for input relations and intermediate results relative to final result size when using high-arity Qdags across different query patterns.

Pattern	**Avg input**	**Median input**	**Avg intermediate**	**Median intermediate**
J3	611.32	312.02	97.49	28.16
T3	783.74	551.59	85.59	38.55
Ti3	263.05	114.2	72.82	19.03
J4	877.18	533.5	41.97	5.25
T4	1355.29	1295.96	76.14	8.63
Ti4	572.66	253.34	56.24	8.58
triangle tadpole	1549.43	1082.39	402.73	61.55
square tadpole	2017.64	1786.57	377.76	51.08
bowtie	2193.56	2114.0	56.63	2.02
triangle barbell	2216.49	1952.94	609.47	591.07
square barbell	1330.69	1228.72	378.57	231.35
pentagon barbell	445.47	404.46	54.99	21.51

different decompositions, when examined at a deeper level, individual query performance shows there is a significant difference between the best and worst performing decompositions for each individual query. It is also apparent that the decompositions behave in the same way in both Qdag variants.

In queries composed of three relations (J3, T3, and Ti3) Configuration 7 was equivalent to doing a multijoin with all three, which is why this decomposition is consistently slower than the others. The first three decompositions (Configurations 1 to 3) are akin to a pairwise join and in average they are a bit slower than Configurations 4 to 6, in which two GHD nodes contain two relations. Nevertheless, as stated before, this decomposition scheme is not always the one that yields results the fastest.

In patterns like T3, the first three tested decompositions are identical in terms of how the attributes relate to each other, and only differ in the actual tuple values. Each decomposition yielded the fastest results of the three in a third of the queries, and the best-performing decomposition for a query outperformed the worst-performing one by an average factor of 12. This shows that the "pattern" of decomposition is not the only relevant factor, and the data values themselves hold relevant information.

From these results, we can conclude that there is no decomposition that is always optimal for a given pattern, and which decomposition is best suited for a query depends on the participating relations. This means that the specific properties of the participating relations and their data must be taken into account to select an appropriate decomposition strategy.

To investigate whether there exists a predictable relationship between decomposition efficiency and intermediate result size, we measured the percentage of queries in which the decomposition that yielded the least number of tuples after the reduction step was also the fastest to execute. The results, presented in Table 6.5, reveal mixed patterns across different query patterns. In T4 and Ti4, there is a strong correlation, but in all others the correlation is low. These lower correlation strengths indicate that intermediate result size is not a significant factor in determining execution time. This makes sense because a small set of tuples can yield large results, or require a lot of processing due to how they are distributed in the

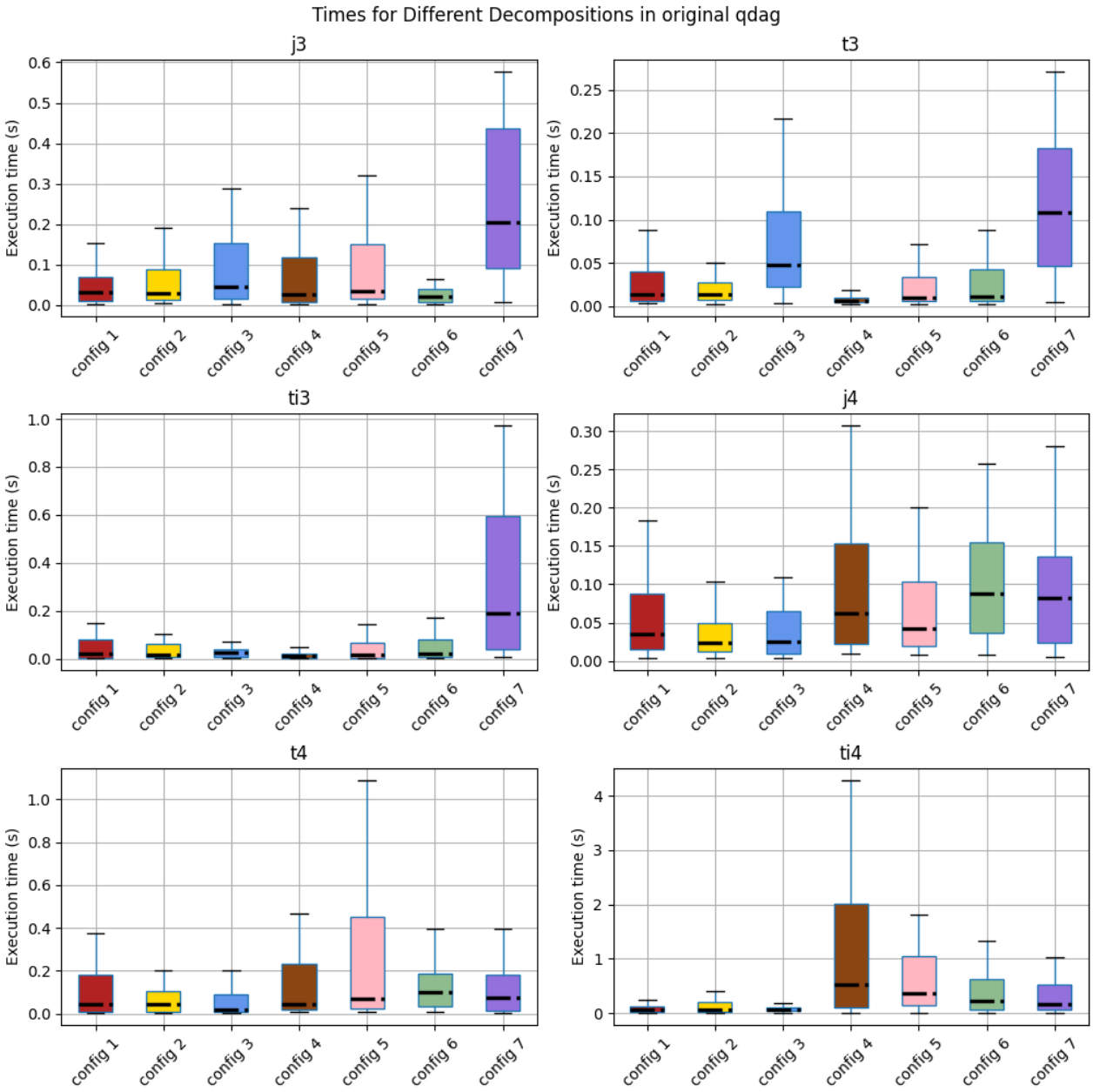


Figure 6.4: Query times (in seconds) of different generalized hypertree decompositions of query patterns, using the original Qdag variant on the Wikidata benchmark.

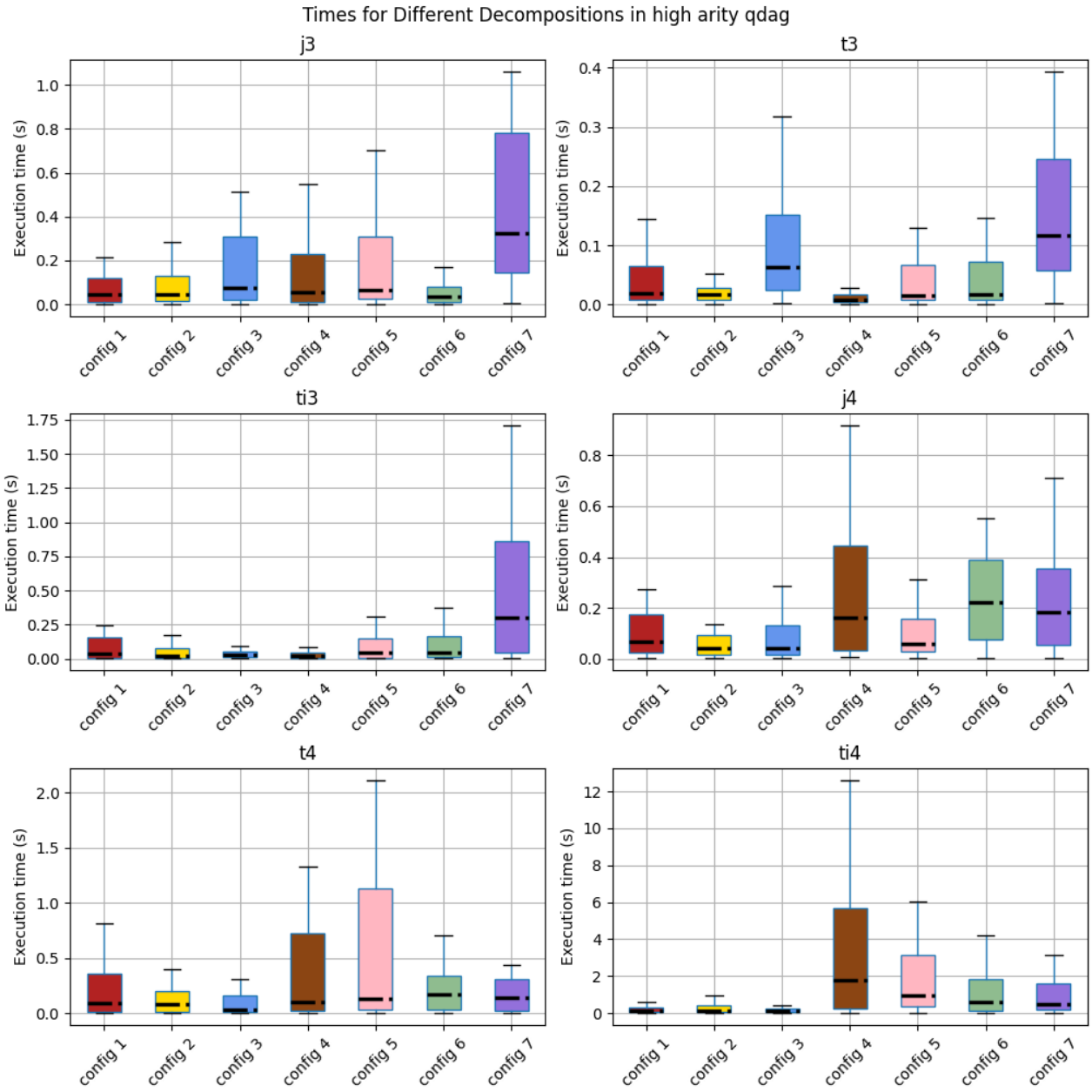


Figure 6.5: Query times (in seconds) of different generalized hypertree decompositions of query patterns, using the high arity Qdag variant on the Wikidata benchmark.

quadtree.

Pattern	Coincidence
J3	60%
J4	46%
T3	52%
Ti3	36%
T4	70%
Ti4	76%

Table 6.5: Percentage of queries in which the fastest decomposition was the one that yielded the least amount of intermediate results after the multijoin reduction.

We tested a simple heuristic for choosing the decomposition of pattern queries composed of three relations to determine whether relation size could serve as a reliable predictor of the overall execution time. The heuristic was based on relation size, where we systematically placed either the smallest, medium-sized, or largest relation (in terms of number of tuples) in a single node of the decomposition tree, with the remaining two relations grouped in another node. There is a noticeable difference in average execution time when the largest relation is isolated in a single node of the decomposition, as can be seen in Figure 6.6. This configuration tends to produce higher average query times compared to decompositions that isolate smaller relations. However, the high degree of variability in all configurations means that this difference, while observable in aggregate statistics, does not translate to reliable performance predictions for individual queries, in particular in the Ti3 pattern.

Although the heuristic based on the original relation sizes seems like a good naive approach, it is not consistent. Simple heuristics based solely on relation size are inconsistent because smaller relations may generate large intermediate results when they have high connectivity or overlapping value ranges with other relations, while larger relations might be more selective. Heuristics that take into account all the characteristics of the underlying data should be explored. Some data properties worth considering are the number of tuples in each relation, how the data is distributed across attributes, and how value ranges overlap.

6.2.3 General execution time

When comparing the execution time of our algorithms, we chose the GHD that performed the best for each of the 50 queries. For plotting and statistical purposes, we recorded timeouts as 1800 seconds. Query times for each algorithm in the original Qdag version are compared in Figure 6.7, while results for the high-arity variant are shown in Figures 6.9 and 6.10, which is a close up of only the GHD-based algorithm results.

Only the multijoin algorithm in its high-arity variant timed out, Table 6.6 shows the number of queries the algorithm timed out in for each pattern and the average execution time the GHD-based algorithm took in those queries. In patterns T4 and Ti4, the multijoin algorithm timed out twice and the GHD optimal algorithm took 1.05s and 0.52s in average to retrieve results for the each query respectively. In the case of square barbell queries, the multijoin timed out in 12% of the queries, for which our proposed algorithm retrieved results in 0.75s

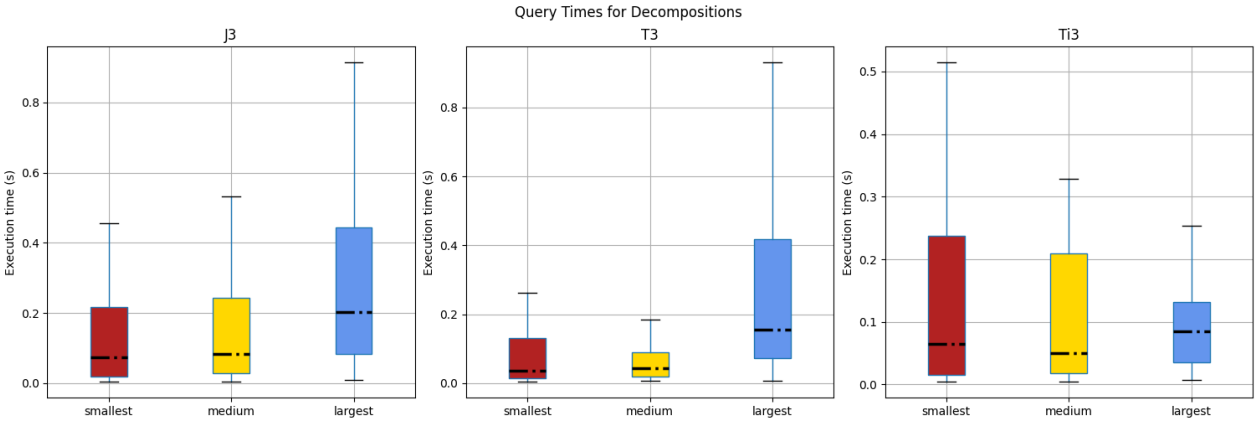


Figure 6.6: Query times (in seconds) of different generalized hypertree decompositions of queries with 3 relations. In each decomposition, the smallest, medium or largest relation was in a single node, and the remaining two in another. The original Qdag variant was used to query the Wikidata benchmark. These results were obtained without using the -O3 optimization flag.

in average. The multijoin algorithm was unable to compute most pentagon barbell queries (70%) before timing out, while the GHD-based algorithm was able to retrieve results at an average of 0.57 seconds.

Pattern	Number of timeouts (after 1800s)	Average query time for GHD [s]
T4	2	1.05
Ti4	1	0.52
square barbell	6	0.75
penta barbell	35	0.57

Table 6.6: Number of queries per pattern that resulted in timeouts (execution was longer than 1800 seconds) when using the multijoin and high arity Qdags, alongside the average execution time for those queries when using the GHD-based join algorithm and high arity Qdags.

As can be seen in Figure 6.7 and Table 6.7, in the original Qdag our proposed algorithm outperforms the multijoin in all patterns save for triangle tadpole, in which the averages differ by ∼0.05 seconds, while their medians are just 0.01 seconds apart. In the T3 query pattern the GHD-based algorithm has a better average query time (0.01 seconds) than the multijoin (0.04 seconds), but the latter has a lower median. The multijoin algorithm also exhibits more variability in query times than the GHD-based one, which is consistent in most patterns except for Ti3 and Ti4.

In the high arity Qdag version, the GHD-based algorithm performs significantly better than the multijoin across all queried patterns, often by orders of magnitude. For simple patterns such as J3 and Ti3, the GHD algorithm achieves average query times of 0.05 and 0.16 seconds respectively, compared to multijoin's 0.92 and 3.56 seconds. The performance gap becomes more pronounced for complex patterns, where the multijoin becomes prohibitively expensive in terms of query time. The GHD-based algorithm completes T4 queries in 0.15 seconds on average, while multijoin requires 100.17 seconds. The most significant difference occurs

in the square and pentagon barbell patterns, in which the GHD-based algorithm completed the queries at an average of 0.45 and 0.43 seconds respectively, while the multijoin took 305.99 seconds on average for the square barbell and 1514.88 seconds in the pentagon barbell queries, with multiple queries timing out after 1800 seconds.

Comparing the performance of both Qdag variants reveals that the original Qdag yields faster results than the high arity version in both the multijoin and the proposed GHD-based algorithm, across all comparable patterns. Table 6.9 shows the average factor by which queries worsen for each pattern. We can see that the multijoin algorithm worsens by a greater factor than the GHD-based algorithm. This can be explained by the fact that while the high-arity intersection process, namely the `select_next` operation, is slower and more complex, the GHD-based algorithm lowers the dimensionality of the queries that need to be computed. This results in less data being processed at a time and faster results.

The original Qdag's inability to handle larger queries limits its applicability, making the high arity variant necessary despite the performance trade-offs.

Pattern	Mean MJ	Median MJ	Std. dev. MJ	Mean GHD	Median GHD	Std. dev. GHD
J3	0.29	0.04	0.64	0.03	0.01	0.04
T3	0.04	0.0	0.16	0.01	0.01	0.02
Ti3	1.13	0.06	4.08	0.09	0.01	0.42
J4	1.46	0.15	4.59	0.03	0.02	0.04
T4	58.03	0.04	278.46	0.08	0.01	0.17
Ti4	39.72	6.97	98.48	0.09	0.04	0.22
triangle tadpole	0.05	0.01	0.13	0.11	0.02	0.18
bowtie	0.13	0.02	0.25	0.02	0.01	0.03

Table 6.7: Statistics for query times per pattern using the original Qdag.

Pattern	Mean MJ	Median MJ	Std. dev. MJ	Mean GHD	Median GHD	Std. dev. GHD
J3	0.92	0.12	2.06	0.05	0.02	0.08
J4	7.38	0.64	24.05	0.05	0.03	0.07
T3	0.13	0.01	0.49	0.02	0.01	0.04
Ti3	3.56	0.18	13.16	0.16	0.01	0.81
T4	100.17*	0.15	369.92	0.15	0.02	0.34
Ti4	162.41*	29.87	381.55	0.38	0.08	1.82
triangle tadpole	0.21	0.02	0.6	0.19	0.04	0.31
square tadpole	3.56	0.54	10.36	0.25	0.07	0.5
bowtie	0.67	0.08	1.3	0.04	0.01	0.06
triangle barbell	2.82	0.55	5.05	0.11	0.1	0.1
square barbell	305.99*	78.77	571.92	0.45	0.3	0.37
penta barbell	1514.88*	1800.0	521.58	0.43	0.29	0.58

Table 6.8: Statistics for query times per pattern using the high arity Qdag, asterisks indicate that some queries timed out after 1800 seconds.

6.2.4 Pruning during semijoin

We measured the effect that the semijoin pruning step, described in Section 5.2.3, had on the overall execution time by calculating the percentage change in the query time when the

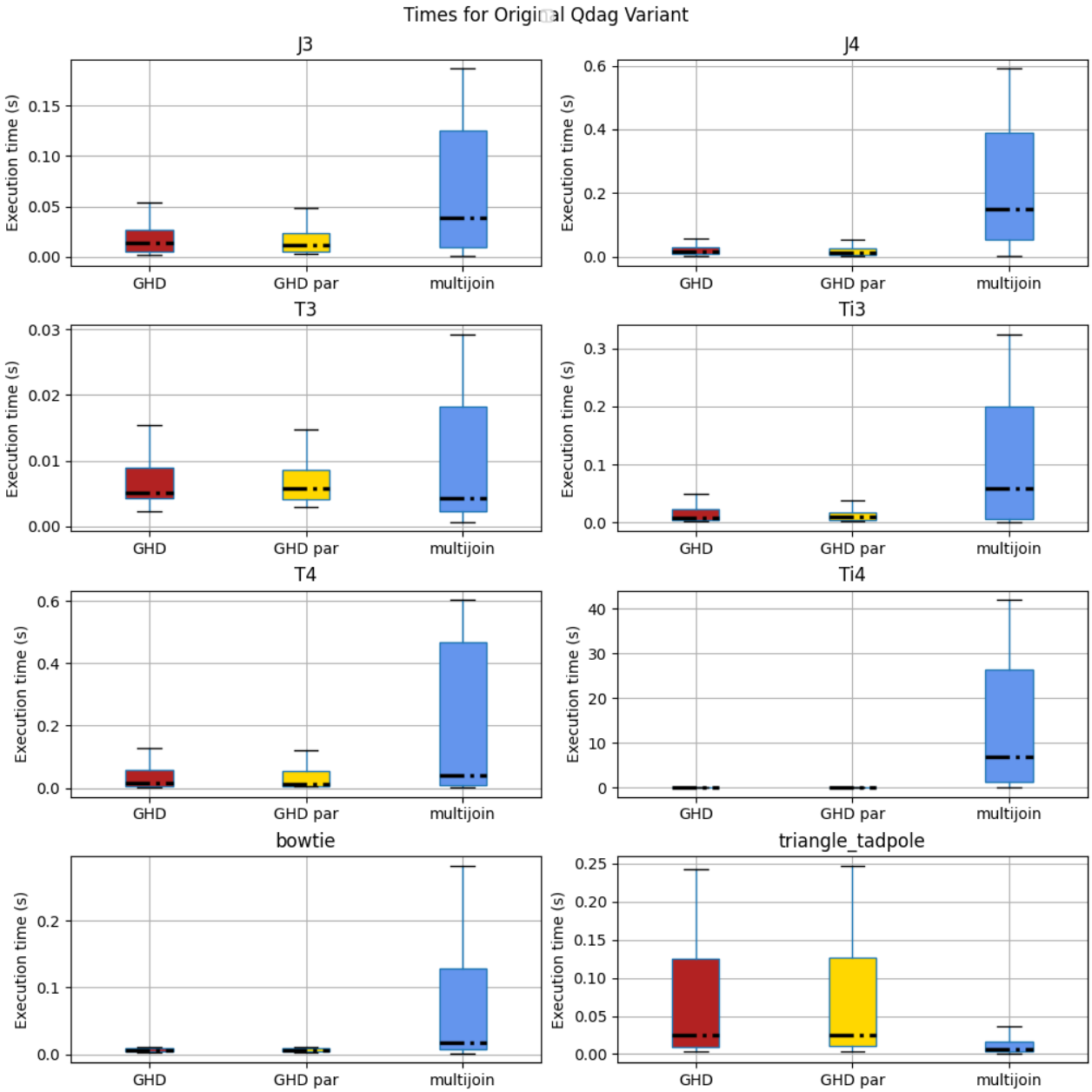


Figure 6.7: Query times (in seconds) for query patterns, using the original qdag variant on the Wikidata benchmark.

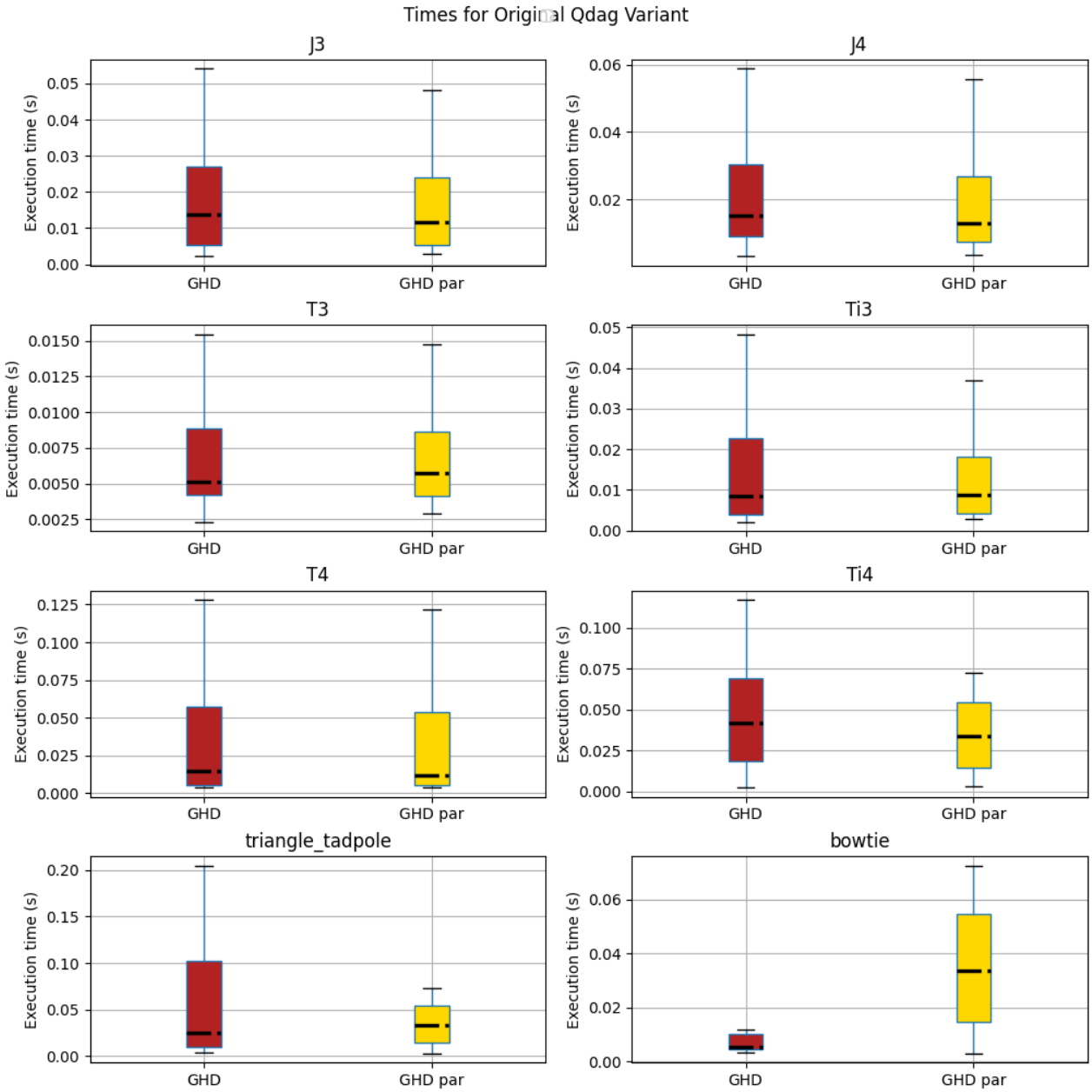


Figure 6.8: Query times (in seconds) for query patterns, using the GHD-based algorithm and the original Qdag variant on the Wikidata benchmark.

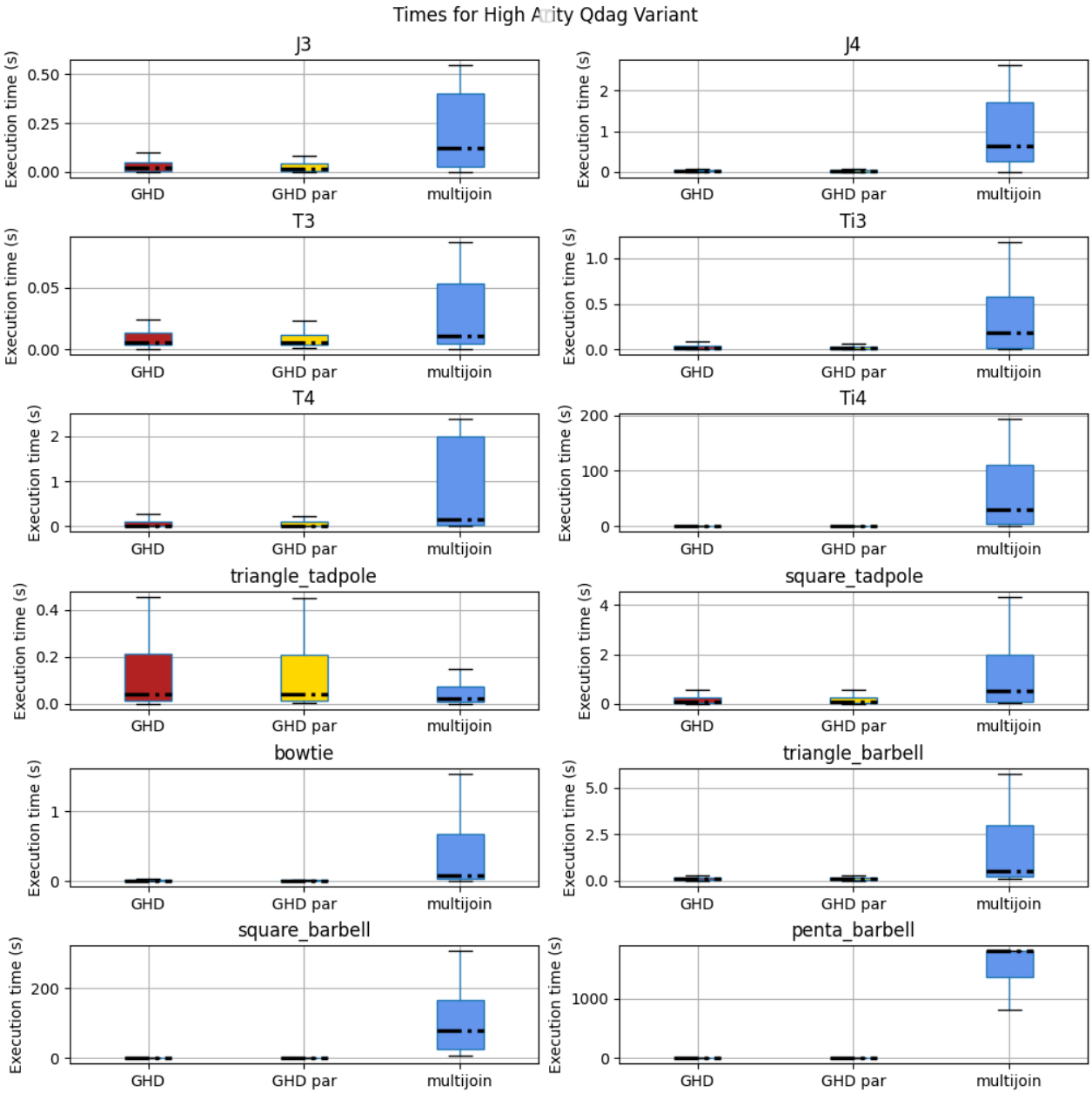


Figure 6.9: Query times (in seconds) for query patterns, using the high arity Qdag variant on the Wikidata benchmark.

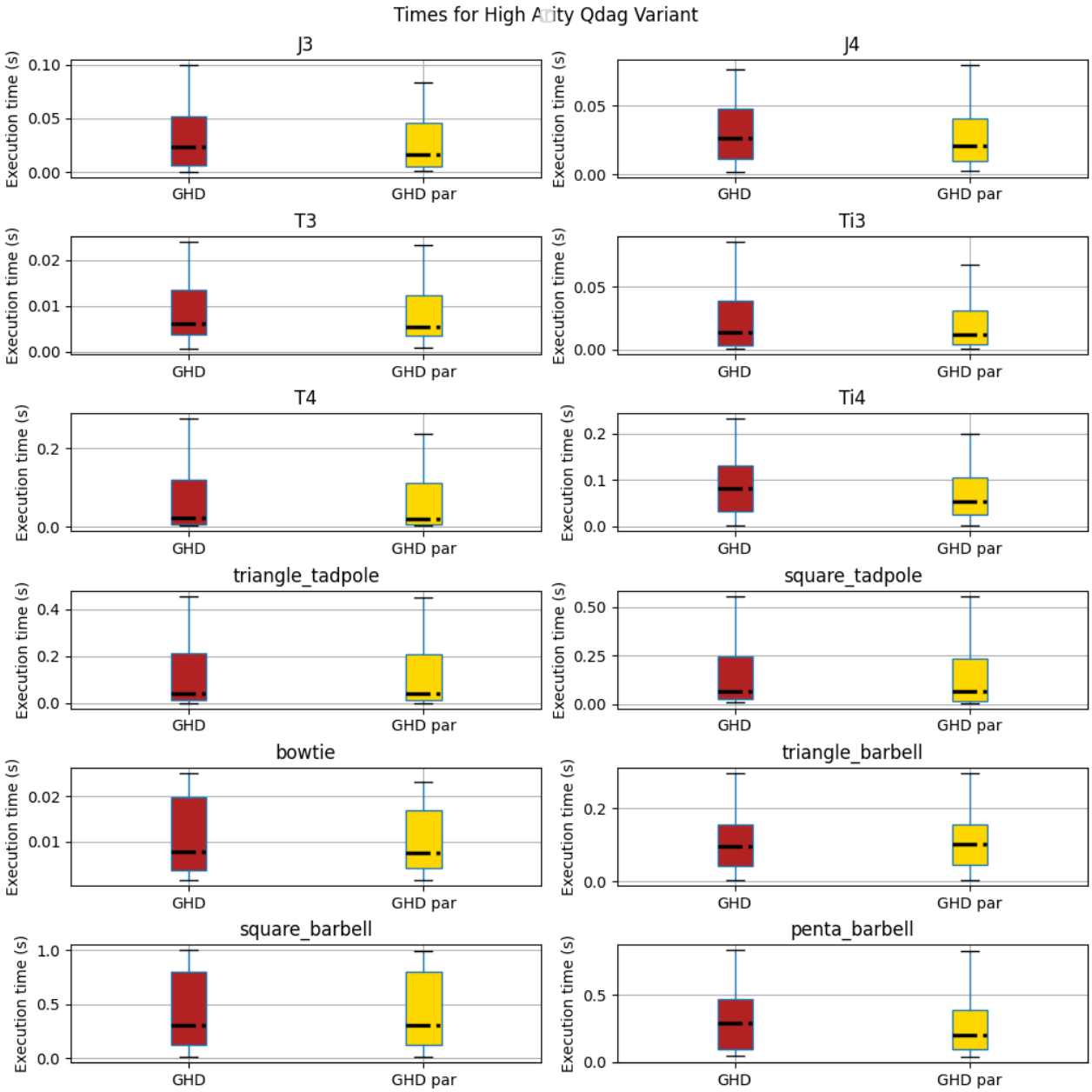


Figure 6.10: Query times (in seconds) for query patterns, using the GHD-based algorithm and the high arity Qdag variant on the Wikidata benchmark.

Pattern	Mean factor mj	Mean factor ghd
J3	2.9	1.48
T3	2.33	1.17
Ti3	2.8	1.25
J4	4.36	1.56
T4	3.77	1.51
Ti4	4.27	1.95
triangle_tadpole	3.34	1.54
bowtie	4.43	1.35

Table 6.9: Multiplying factor

GHD-based algorithm was executed with and without the pruning step. Figure 6.11 shows the distribution of the percentage change in all queries performed with the original Qdag variant. A positive percentage change means that the pruning step reduced query time, while a negative value indicates query time was actually increased when pruning was done. Around half of the queries performed better without pruning, which means that the amount of work saved by stopping the descent of an already-visited subtree does not compensate for the overhead produced by checking the pruning conditions and materializing the temporary bitmap.

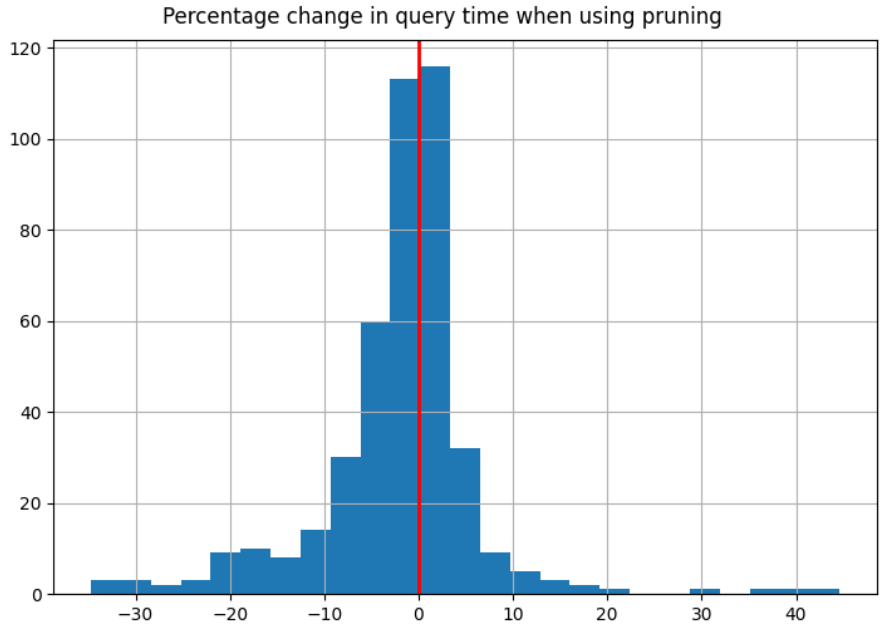


Figure 6.11: Histogram showing the percentage change in total query time when using the pruning optimization during semijoins, over all queries done. A positive percentage change indicates that pruning decreases execution time, and a negative number means query time increased when pruning was done.

6.2.5 Reduction Parallelization

We measured the effect of parallelizing the node reduction step in the GHD-based algorithm by comparing execution times between the parallel and sequential versions across different pattern types. The results show mixed outcomes for parallelization, with performance improvements heavily dependent on the specific pattern being processed. Patterns T3, Ti3, Triangle Tadpole, and Triangle Barbell show negative percentage differences, indicating that the parallel version actually increases execution time by 4-11% on average, due to parallelization overhead outweighing the benefits for these less computationally intensive queries. In contrast, more complex patterns such as Ti4, J4, and the pentagon barbell demonstrate improvements with the parallel approach, showing 11-18% reductions in execution time on average and maximum improvements reaching up to 46% for Ti4. The mixed results suggest that parallelization of the node reduction step provides benefits for computationally demanding patterns where the parallel processing gains exceed the thread-coordination overhead, while simpler patterns may be better served by the sequential approach. This opens the possibility of defining a heuristic for determining which node-reduction approach to use based on the complexity of the query.

Pattern	Mean difference	Median difference	Most hindrance	Most improvement
J3	1.92	6.29	-136.56	37.1
J4	11.09	8.83	-67.08	45.03
T3	-7.41	4.05	-91.47	42.71
Ti3	-11.88	-1.85	-121.36	41.02
T4	10.57	9.24	-34.31	43.53
Ti4	17.5	23.02	-34.98	46.77
triangle tadpole	-7.75	-1.06	-88.15	9.41
square tadpole	12.57	9.58	-7.34	42.91
bowtie	0.33	4.28	-60.55	39.56
triangle barbell	-4.58	-4.08	-23.09	15.3
square barbell	2.3	0.88	-14.29	35.79
penta barbell	18.51	18.95	-4.22	45.86

Table 6.10: Percentage change between using the parallel GHD-based algorithm execution and the sequential one. A positive value means the parallel version decreased execution time and a negative value indicates execution time was increased by the parallel version.

Chapter 7

Conclusion

In this thesis, we have implemented an alternative Qdag variant that is not limited to queries of at most 5 attributes, extending its capabilities and applicability for more complex queries. This high arity Qdag variant addresses a fundamental constraint of the original implementation, permitting queries of graph patterns that were previously unsupported. While our results show that this variant has a lower performance than the original Qdag for join queries with fewer than 5 attributes, it is a promising proof of concept that can be improved. We give some directions of future work on this in Section 7.1.

We also added support for semijoin operations to both Qdag variants, which opens the possibility of implementing other algorithms that benefit from semijoins in the future.

Another significant contribution of this thesis is the implementation of a new algorithm for conducting join queries in graph databases that uses generalized hypertree decompositions as query plans, and is capable of surpassing the AGM bound. This algorithm was implemented in both the original Qdag and the high arity version. Our experimental evaluation showed that the new algorithm generally delivers faster results compared to the existing multijoin approach, with limited space overhead. The algorithm also exhibits less performance variability than the multijoin, and in the high arity version completed queries that had resulted in timeouts when using multijoin algorithms.

From the experimental analysis, we can conclude that the chosen decomposition strategy significantly influences the execution time. While most tested decompositions yielded faster results than the multijoin, a "bad" decomposition can create large intermediate results that slow down the query or create what is essentially a pairwise-join query plan.

Our investigation of decomposition selection heuristics showed that simple approaches based solely on relation size are insufficient to predict optimal performance, as the computational cost depends on how multiple data characteristics combine during query execution.

Based on our performance evaluation across both Qdag variants, queries involving fewer than 5 attributes are best served by the GHD-based algorithm implemented on the original Qdag variant, as this provides the best performance for small queries. For larger queries that exceed the 5-attribute limit, the GHD-based algorithm implementation in the high arity

version should be used. In this work we were not able to detect a pattern for when it is advisable to use the parallel version of the GHD-based algorithm, or the pruning step of the semijoin algorithm since they both gave varied results.

This thesis successfully achieved its main objective of extending the capabilities of the Qdag data structure introduced by Arroyuelo et al. [9], allowing it to handle more complex graph pattern operations. The first specific objective was fully met through the introduction of modifications that enable Qdags to solve higher-arity join queries while guaranteeing Worst Case Optimality. Similarly, the second objective was achieved by introducing a more efficient approach to joins based on generalized hypertree decompositions, making the index more apt for real-world data and queries. However, the final objective regarding the comparison of our algorithms to existing solutions was only partially met. We were only able to compare our approaches to the original Qdag, leaving out solutions such as Emptyheaded. Despite this limitation, the experimental results obtained provide valuable insights into the performance characteristics of our proposed methods in terms of both time and space usage. Our contributions provide solutions for complex pattern matching that was previously unsupported or inefficient, and establish a foundation for future research in query optimization and decomposition selection strategies.

7.1 Future Work

High arity Qdags are necessary for solving larger queries, but the join operation's running time is higher than in the original Qdag variant in both of our proposed algorithms. This increase in execution time is due to the Qdag intersection process being more laborious, it requires obtaining the index of every set bit in a node instead of doing a bitwise And between materialized nodes. To reduce the high arity Qdag's execution time we propose storing nodes in a level as sparse arrays instead of bitmaps. In this way obtaining the 1s in a node should be much faster, since it eliminates the need to sequentially select all the set bits.

Our GHD-based join algorithm currently requires the user to provide a GHD for the join query. If this GHD is suboptimal, the algorithm may cease to be WCO. The next step in this work is to determine a heuristic for selecting a query's GHD that can guarantee worst-case optimality or even FHTW optimality, add automatic query decomposition to our algorithm, and compare the resulting system to EmptyHeaded. To create this heuristic, we need to study how skew, value overlap, data size, attribute distribution and other data properties affect query time.

Another runtime enhancement that can be done in the GHD-based algorithm is to find a heuristic for determining when it would be convenient to use semijoin pruning or the parallel GHD node reduction method. Since both of these had mixed results, sometimes speeding up query times but at other times slowing down, it would be ideal to see if this can be determined beforehand with some accuracy.

Finally, another future point of study is using Qdags to represent and operate on relations of more than two attributes. Currently, it is technically possible, but it has not been formalized or explored in terms of applications.

Bibliography

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded. *ACM Transactions on Database Systems*, 42(4):1–44, 2017.

[2] D. Arroyuelo, D. Campos, A. Gómez-Brandón, G. Navarro, C. Rojas, and D. Vrgoc. Space & time efficient leapfrog triejoin. In *Proc. 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, page article 2, 2024.

[3] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD-/PODS '21, page 102–114, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[5] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *IN PROC. OF WG'05*, 2005.

[6] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. Wikidata graph pattern benchmark (wgpb) for rdf/sparql, October 2019.

[7] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. *Lecture Notes in Computer Science The Semantic Web – ISWC 2019*, page 258–275, 2019.

[8] J. Ian Munro. Tables. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

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

[10] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, February 2014.

[11] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*, GRADES'15, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[13] D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil-Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. MillenniumDB: An open-source graph database system. *Data Intelligence*, 5(3):560–610, 2023.

[14] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

Appendix A

Space Usage of GHD-optimal algorithm

These results were obtained using the high arity Qdag variant. The intermediate result size is the space used by the Qdags resulting from the multijoin reduction step of the GHD-optimal algorithm.

Table A.1: Space used to store the relations at each step of the algorithm of a J3 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
1,205,382	232,724	510,524	482,876	2,938
2,504,814	3,966,860	422,108	2,023,748	5,986
196,950	87,452	35,252	112,268	2,938
4,991,334	1,996,556	684,524	4,473,620	556,498
4,507,782	160,076	181,916	4,284,236	2,938
2,133,390	1,768,892	75,908	332,228	2,938
1,050,678	59,324	839,780	193,364	3,298
462,102	293,132	136,412	377,612	105,130
1,912,086	1,215,044	639,956	83,492	3,370
2,278,686	592,244	121,028	2,025,044	,2938
109,710	18,692	16,652	85,340	2,938
11,628,534	154,556	332,108	11,389,868	54,538
1,357,614	3,267,308	342,236	1,283,108	7,618
402,054	179,084	376,796	242,660	3,346
2,453,814	77,252	79,604	2,332,796	3,010
5,539,302	5,593,412	67,508	48,620	2,938
1,575,582	374,108	1,127,900	101,180	3,274
2,566,782	806,108	2,256,164	485,828	2,938
6,538,782	3,483,356	143,732	2,994,548	5,962
198,510	118,556	69,596	23,420	2,938
177,270	25,820	21,548	144,284	2,938
1,329,030	129,212	166,028	1,172,420	3,442
3,700,806	103,028	458,036	3,207,044	2,938
1,412,430	170,852	1,167,740	111,524	2,938
278,406	33,044	136,292	118,412	2,938
7,015,254	4,013,348	579020	3067724	82,354
1,475,022	489,188	667,172	378,260	2,938
175,566	61,748	40,940	113,276	4,426
1,568,382	379,292	644,588	1,167,068	6,922
4,600,998	4,088,948	259,700	613,820	6,466
108,558	21,332	42,836	60,836	2,938
2,466,870	3,928,916	547,052	2,025,596	241,906
4,634,406	646,172	1,140,644	3,049,436	2,938
3,031,926	54,908	26,780	2,974,076	4,378
3,002,118	154,076	1,933,412	2,837,276	11,818
5,537,382	5,431,436	84,140	60,836	3,154
421,806	187,052	545,180	221,684	2,938
897,462	116,348	334,508	464,564	2,938
2,979,462	103,124	2,618,060	307,124	2,938
3,802,326	96,836	3,695,540	26,708	3,058
3,073,062	87,788	585,764	2,912,660	2,938
97,110	58,484	58,124	28,436	2,938
2,258,094	1,488,668	194,588	611,012	2,938
1,422,126	1,317,092	111,908	159,452	28,762
9,061,686	8,247,716	5,847,80	843,092	825,778
18,450,102	14,052,428	1,485,380	4,284,236	2,938
772,662	2,060,516	50,948	667,748	2,938
1743294	953,420	129,380	778,580	9,370
1,611,846	155,372	48,212	1,421,516	7,786
734,694	5,765,564	18,236	91,436	3,034

Table A.2: Space used to store the relations at each step of the algorithm of a T3 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
4,106,262	3,475,004	460,292	465,188	3,466
1,254,126	673,292	583,676	334,436	2,938
4,250,622	1,675,076	296,684	2,577,884	3,826
476,094	29,540	158,900	300,260	2,938
4,614,318	1,173,404	2,237,948	3,039,260	93,466
4,722,510	3,782,012	541,292	1,333,244	6,754
413,238	238,196	41,468	199,580	2,938
5,537,526	3,973,892	446,036	1,516,364	2,938
1,280,718	27,116	137,732	1,128,812	2,938
881,334	138,596	722,852	255,356	15,154
1,442,382	1,226,804	175,460	85,748	2,938
298,734	122,204	120,860	70,052	2,938
1,647,678	83,348	25,364	1,570,700	3,442
4,282,302	3,735,980	356,996	892,244	2,938
6,370,854	121,028	63,332	6,433,196	2,938
1,730,646	1,332,956	904,316	375,428	2,938
1,603,374	1,228,220	63,380	355,532	2,938
2,061,198	61,580	1,322,780	699,212	2,938
444,438	216,284	37,412	201,236	2,938
1,535,142	195,212	1,256,036	627,284	5,986
2,617,878	192,476	1,725,812	731,444	2,938
2,625,342	1,147,748	697,340	1,387,364	2,938
1,679,118	63,124	49,180	1,665,388	2,834
1,788,126	1,228,220	391,604	179,852	2,938
2,984,550	623,108	66,836	2,339,036	2,938
575,046	31,004	567,644	43,700	5,890
82,734	25,220	38,540	32,780	2,938
1,082,190	142,340	515,444	436,148	2,938
595,806	435,668	122,852	53,132	2,938
8,113,062	6,493,724	528,212	1,632,932	14,554
6,534,990	6,486,188	36,860	50,180	2,938
850,686	78,884	763,772	32,060	2,938
1,815,246	127,604	23,828	1,679,468	2,938
197,574	57,308	94,268	130,412	25,186
107,022	47,588	33,116	41,348	2,938
1,984,950	90,188	786,332	1,133,564	2,938
3,764,406	3,630,188	139,076	89,684	2,938
931,734	782,324	105,476	64,196	2,938
871,038	746,180	85,172	54,452	2,938
11,497,230	762,836	10,583,156	1,439,756	4,282
829,230	437,540	375,020	102,956	2,938
11,473,134	784,124	10,594,964	173,228	2,938
5,006,862	4,962,716	37,028	20,420	2,938
701,670	250,244	134,876	408,692	2,938
1,274,382	56,132	410,444	838,220	3,466
5,584,782	5,164,076	215,228	372,404	5,338
1,993,206	850,628	108,596	1,122,308	2,938
541,470	73,844	333,284	179,300	2,938
5,084,190	4,962,716	100,100	30,548	2,938
6,663,390	6,486,188	190,844	50,444	2,938

Table A.3: Space used to store the relations at each step of the algorithm in a Ti3 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
953,550	53,756	499,364	408,692	2,938
44,646	16,676	23,708	120,500	16,762
735,558	181,268	627,572	604,820	74,338
3,236,742	2,083,532	840,140	1,172,276	67,234
180,534	104,372	44,972	57,932	2,938
115,974	45,068	77,852	25,484	2,938
235,398	150,404	49,124	125,156	2,938
235,518	145,892	33,668	74,828	2,938
449,958	40,340	38,828	379,748	2,938
134,310	45,332	84,788	32,060	2,938
5,080,734	2,023,484	30,012,812	3,049,172	2,938
2,400,390	724,196	54,572	1,771,436	5,002
395,382	44,900	275,228	94,916	2,938
242,670	271,196	62,084	28,940	4,834
341,310	238,724	62,300	126,164	2,938
2,924,766	1,714,556	1,283,204	409,484	2,938
270,126	117,836	50,732	139,724	2,962
863,478	777,116	78,260	43,412	2,938
131,166	18,956	23,012	98,588	2,938
450,318	361,436	97,748	180,836	3,466
813,654	667,748	34,196	288,308	2,938
1,229,934	838,748	515,396	73,748	3,490
1,640,718	1,555,700	151,820	123,644	56,506
625,254	201,140	283,100	314,948	11,530
4,955,910	473,708	4,125,044	919,940	62,674
130,182	54,284	37,364	101,516	2,938
8,583,054	5,460,428	2,978,300	190,916	2,938
540,942	94,340	25,916	435,668	2,938
214,494	102,116	54,980	95,540	2,938
919,782	574,148	347,756	57,692	5,098
1,706,094	34,436	22,436	1,659,284	2,938
507,486	399,836	82,268	87,524	4,522
629,766	528,164	92,276	59,396	2,938
445,326	129,164	278,612	166,364	20,338
288,678	225,932	141,092	38,300	3,730
1,245,102	698,228	325,748	305,252	2,938
1,800,486	220,148	565,580	1,058,468	2,938
715,110	657,668	32,852	45,356	2,938
991,182	698,228	71,828	242,972	2,938
4,242,822	4,120,076	24,757,436	4,921,964	144,151,642
236,790	115,316	82,748	129,740	18,634
1,085,670	838,748	243,380	207,020	2,938
225,678	45,908	2,186,468	170,108	3,058
781,014	33,428	741,812	53,180	2,938
677,574	73,124	608,228	36,764	2,938
227,838	165,452	124,820	112,076	33,730
210,342	47,492	159,836	40,604	3,394
666,462	495,332	123,572	67,484	2,938
206,598	304,532	260,708	448,172	1,421,290
789,630	204,140	367,508	411,284	4,522

Table A.4: Space used to store the relations at each step of the algorithm of a J4 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
367,496	7,332	20,868	5,916	3,138
3,890,768	362,964	30,516	57,132	3,138
3,653,984	25,788	7,884	23,076	3,138
1,794,176	23,556	8,196	16,452	3,210
1,077,632	62,892	144,180	66,204	14,346
4,226,120	73,716	69,252	109,716	3,402
1,155,080	193,908	563,988	694,380	3,138
7,288,088	208,332	181,884	32,484	6,042
5,545,904	6,367,524	1,025,484	2,349,636	3,138
8,915,816	2,191,332	428,724	79,188	98,994
1,913,072	78,996	50,964	198,276	12,594
935,720	38,748	8,124	6,084	3,138
1,483,832	10,644	59,700	12,252	3,138
5,261,600	2,721,444	77,820	45,612	29,010
6,634,592	77,436	60,636	676,428	3,138
6,228,704	295,836	292,092	5,867,532	31,554
3,555,848	1,446,492	4,721,772	676116	5,874
14,261,432	8,031,372	14,499,396	2,767,908	6,954
7,605,128	4,907,100	223,476	21,636	3,642
4,918,928	92,820	12,012	9,300	3,498
4,687,232	773,052	12,228	1,231,500	3,138
3,826,352	67,356	6,756	15,036	3,138
931,328	21,948	7,980	18,996	5,178
5,970,968	278,508	327,804	108,252	3,162
5,169,704	2,239,668	1,607,772	1,191,636	12,066
3,493,784	186,732	242,700	24,396	3,138
8,034,848	80,988	119,316	95,988	3,138
1,211,096	115,500	42,588	127,308	3,138
1,210,184	8,292	7,116	72,204	3,186
939,536	43,020	113,196	34,908	8,394
5,614,088	16,860	7,612,644	10,650,492	2,3634
5,205,752	101,940	177,660	303,156	3,138
2,038,760	119,100	85,500	57,300	3,138
2,281,928	258,084	34,548	67,596	45,042
458,936	26,748	20,604	28,284	3,138
6,140,096	98,460	5,901,276	106,740	253,650
6,324,608	39,396	54,564	10,620	3,138
3,711,488	14,364	21,660	31,212	4,122
12,279,584	2,341,332	1,658,700	1,234,308	3,690
1,594,304	17,148	230,340	8,052	3,138
3,014,576	69,828	91,596	348,612	3,522
9,442,208	28,236	1,910,604	169,332	3,618
1,287,512	359,076	219,972	117,660	9,906
641,000	11,820	6,612	21,828	3,186
3,483,296	162,132	169,476	47,220	7,218
1,448,216	66,804	922,860	6,828	3,138
2,340,456	5,300	21,380	11,804	3,034
590,912	9,924	14,676	279,468	7,410
4,560,344	7,620,156	79,572	73,692	103,770
6,783,080	115,188	116,532	5,921,148	4,386

Table A.5: Space used to store the relations at each step of the algorithm of a T4 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
2,805,200	39,468	124,860	24,492	3,138
6,205,856	28,884	228,228	208,164	3,138
9,391,136	158,244	6,676,164	141,372	3,642
1,377,968	49,380	15,156	218,964	3,138
6,676,088	33,012	43,044	139,644	55,170
6,632,264	66,252	28,740	14,220	3,138
1,67,672	44,868	133,788	50,364	3,138
4,091,384	125,436	18,924	7,044	3,138
4,462,328	246,972	121,140	80,004	3,138
1,966,832	343,740	262,764	549,492	63,618
9,316,616	28,068	276,780	371,628	3,138
2,678,360	2,6556	85,380	95,772	3,138
1,390,256	75,708	107,580	86,436	10,026
1,038,464	18,972	595,212	39,972	3,642
5,273,720	399,732	805,476	361,524	3,138
13,884,248	12,597,108	57,108	43,140	3,138
6,071,312	2,780,868	80,964	54,564	3,834
13,926,368	1,998,492	4,853,652	1,666,068	3,666
9,344,168	573,468	302,868	363,516	5,106
7,128,632	57,108	5,508	5,724	3,138
6,781,112	753,564	190,668	289,212	3,138
2,921,240	768,708	89,004	324,588	3,138
4,356,680	12,780	10,692	81,012	3,138
2,196,752	1,174,020	495,084	437,964	319,218
2,522,072	23,004	689,292	23,052	3,138
4,680,512	7,212	7,284	402,156	3,138
2,229,752	76,980	204,012	60,492	3,138
770,360	7,932	9,084	6,900	3,138
2,533,568	1,037,292	1,181,172	1,326,492	3,138
3,981,704	252,540	19,956	19,404	3,138
5,612,432	282,036	728,844	169,068	3,138
10,983,272	4,261,908	4,572,516	5,198,004	3,416,586
4,236,344	6,588	30,036	134,916	3,138
7,839,632	277,644	363,108	3,180,300	3,642
1,111,280	33,156	7,308	5,604	3,138
3,578,936	186,564	1,572,156	186,012	11,754
11,701,400	1,302,948	1,303,092	11,033,508	3,666
550,856	19,044	20,436	23,964	4,650
13,868,048	2,690,772	3,351,228	6,677,940	3,138
4,042,040	74,340	258,060	76,716	3,138
732,464	10,092	31,356	33,948	3,138
5,210,816	13,332	415,308	36,132	3,138
5,709,296	153,828	18,132	11,388	3,,138
6,535,376	3,090,180	911,268	461,532	3,138
4,118,888	247,212	427,092	53,796	3,138
9,830,288	146,172	1,846,836	555,252	3,138
888,896	45,108	137,148	45,084	7,482
1,670,888	10,884	17,436	19,836	3,138
3,162,752	227,580	35,556	32,316	3,138
1,103,240	144,564	37,044	25,308	3,570

Table A.6: Space used to store the relations at each step of the algorithm of a Ti4 query, in bytes. Intermediate results depend on the chosen GHD.

input	intermediate 1	intermediate 2	intermediate 3	results
772,088	1,234,332	234,204	43,812	16,651,290
465,920	5,964	6,516	14,820	3,138
430,208	44,460	43,332	261,228	5,802
1,973,912	12,852	22,620	360,396	3,138
806,552	69,948	41,124	121,620	3,738
749,312	76,092	347,772	143,196	4,794
12,733,328	4,637,676	203,436	296,316	3,138
852,896	50,268	92,316	187,092	3,738
3,382,880	49,548	33,780	520,020	3,138
2,169,968	31,476	157,044	17,604	3,138
1,007,144	447,204	69,660	79,956	4,890
1,552,664	28,620	328,740	68,628	3,138
1,077,968	8,508	155,052	5,796,516	3,714
2,137,880	102,468	1,150,332	127,308	3,666
460,280	14,388	22,836	24,036	3,738
1,509,272	37,620	437,292	49,932	3,186
2,363,888	72,036	54,588	54,684	4,794
1,248,080	332,916	91,164	94,308	4,866
12,310,496	42,180	3,186,060	102,804	3,138
897,872	358,716	160,908	453,636	89,994
1,871,408	2,134,092	1,380,036	4,274,436	3,663,357,114
203,120	5,964	21,732	5,508	3,138
3,592,184	1,435,644	156,444	222,660	3,642
393,896	13,836	33,468	8,724	3,138
2,479,592	14,844	122,748	67,884	62,202
14,306,192	1,623,852	6,546,276	1,934,340	3,138
2,550,536	619,044	29,292	26,916	3,138
494,456	12,540	17,868	47,628	3,138
2,597,720	462,852	730,668	854,604	4,074
759,176	27,972	142,644	69,564	3,138
3,660,704	48,348	527,100	56,340	3,138
1,434,464	10,884	7,932	99,228	3,138
1,437,440	54,492	120,156	141,156	4,266
4,157,888	207,324	660,684	1,214,964	4,218
794,984	98,892	74,748	72,348	3,138
776,552	72,444	73,836	90,468	9,114
4,593,536	16,860	16,548	163,764	14,562
2,698,256	203,316	1,239,180	76,668	43,554
1,020,488	259,068	101,244	20,100	3,138
290,600	61,956	110,700	35,796	5,442
622,280	51,036	13,956	5,508	3,138
1,014,848	18,372	41,172	89,364	3,138
2,351,528	99,924	55,740	285,156	3,138
3,051,200	40,548	16,305,972	390,516	16,746
1,616,912	93,156	183,708	85,308	10,218
4,432,328	250,452	86,604	245,028	9,138
1,641,752	55,236	3,363,420	70,116	185,586
513,848	40,116	10,836	8,292	15,954
458,696	59,580	62,988	106,764	3,714

Table A.7: Space used to store the relations at each step of the algorithm in a Bowtie query, in bytes.

input	intermediate results	results
3,428,292	7,884	4,842
6,450,876	5,508	3,138
4,998,708	5,844	4,722
6,277,188	5,508	3,138
2,835,180	5,508	3,258
4,383,084	6,276	3,138
4,110,564	8,652	3,138
7,509,300	8,028	3,138
4,347,060	6,756	3,138
7,529,100	6,372	3,138
7,325,652	15,684	4,026
10,724,388	9,972	3,402
10,453,908	6,276	3,138
1,585,596	5,508	3,138
1,273,068	13,164	3,570
11,843,916	13,116	3,570
7,041,636	7,620	3,138
5,595,660	5,580	3,138
2,690,436	5,940	3,138
1,919,316	8,316	3,138
6,975,036	6,108	3,138
5,337,852	6,036	3,138
5,134,404	15,348	3,834
7,474,980	1,720,000	3,402
6,685,068	868,956	3,594
6,685,068	868,956	3,138
6,534,084	803,940	3,690
8,301,420	5,940	3,138
7,140,756	5,964	3,138
7,978,260	6,636	3,234
6,581,676	1,720,000	3,690
7,785,228	804,756	3,354
6,699,756	6,396	3,186
7,832,820	1,730,000	3,762
6,747,348	7,236	3,234
7,042,908	869,772	3,186
14,083,476	7,860	3,138
5,936,580	7,500	3,162
6,668,628	6,324	3,138
9,391,212	6,324	3,138
7,312,068	6,324	3,138
12,107,172	6,324	3,138
7,842,492	6,324	3,138
4,264,620	5,868	3,138
9,141,636	5,508	3,138
4,404,348	6,324	3,138
15,143,364	5,940	3,138
12,722,700	5,508	3,138
12,619,356	188,652	3,138
13,107,228	5,508	3,138

Table A.8: Space used to store the relations at each step of the algorithm in a Triangle Tadpole query, in bytes.

input	intermediate results	results
400,354	6,492	3,162
363,538	5,916	3,138
449,530	8,124	3,138
10,879,330	1,284,420	3,138
2,303,746	29,772	3,138
3,437,074	555,372	3,138
5,889,394	296,628	5,298
6,191,842	128,676	3,546
3,117,994	219,660	3,138
9,274,498	7,959,590	3,138
5,425,258	771,228	3,138
1,488,874	231,732	3,138
12,763,234	83,532	3,138
9,374,602	8,556	3,138
1,915,474	204,948	3,138
457,642	242,796	3,354
9,074,578	611,652	3,138
1,442,674	163,788	3,138
11,451,898	67,884	3,138
13,905,850	10,969,600	3,138
13,900,474	8,867,920	3,138
2,368,282	204,948	3,138
833,914	95,964	3,138
7,630,282	1,620,440	3,138
2,907,874	1,462,260	3,138
8,756,770	7,959,590	3,138
6,022,378	139,476	3,138
876,442	13,668	3,138
4,873,498	144,276	3,138
927,562	137,388	3,138
904,834	89,508	3,138
898,474	79,548	3,138
975,442	95,964	3,138
7,500,346	1,866,320	4,146
10,516,498	2,292,250	3,138
7,385,770	842,628	10,290
4,558,594	1,870,400	10,290
1,262,290	19,020	3,138
10,299,226	7,959,590	3,138
3,356,002	59,436	3,138
8,211,226	489,180	3,138
7,513,330	15,828	3,138
1,140,370	11,700	3,162
1,705,546	184,116	3,234
7,543,618	189,636	3,282
5,455,186	65,100	3,138
4,825,258	1,024,480	4,962
4,982,746	1,064,000	5,370
6,513,562	949,644	3,138
6,358,138	2,847,880	3,138

Table A.9: Space used to store the relations at each step of the algorithm in a Square Tadpole query, in bytes.

input	intermediate results	results
3,716,796	59,316	3,674
5,607,564	9,924	3,530
11,795,028	1,170,000	3,530
7,938,564	2,780,000	3,530
6,482,148	1,190,000	3,842
10,153,716	5,532	3,530
10,085,652	5,532	3,530
11,558,700	5,980,000	3,842
16,696,620	4,720,000	3,842
11,662,908	3,490,000	3,842
1,178,364	54,948	3,530
7,710,276	1,050,000	3,962
6,384,060	298,284	3,530
1,768,308	341,796	3,530
11,679,540	245,916	4,082
10,780,908	71,196	4,634
11,796,396	1,350,000	4,082
12,441,900	39,756	3,530
7,557,012	85,764	3,530
7,408,956	18,804	3,530
12,532,164	744,684	3,674
10,999,140	537,420	3,674
10,995,444	327,684	3,674
6,244,572	673,404	3,530
7,253,556	4,720,000	4,106
6,306,588	180,324	3,530
16,661,172	16,100,000	3,530
6,244,236	43,188	7,058
7,057,908	12,156	4,034
6,321,804	16,620	5,210
10,077,420	526,332	3,530
5,166,204	2,780,000	3,530
4,519,308	280,644	3,530
4,182,780	583,332	3,530
5,193,084	52,332	4,034
4,690,548	11,100	4,034
5,044,308	150,780	4,034
5,146,644	200,436	4,034
5,042,748	56,772	4,322
5,042,748	56,772	4,322
11,428,908	2,770,000	3,530
5,195,316	3,450,000	3,554
8,780,820	1,870,000	3,530
2,841,396	30,468	3,530
2,557,788	91,332	3,530
3,570,924	171,012	3,530
4,297,068	68,844	4,058
4,476,948	14,796	4,106
8,501,700	11,000,000	3,554
8,491,428	20,748	3,554
8,628,348	33,900	3,554

Table A.10: Space used to store the relations at each step of the algorithm in a Triangle Barbell query, in bytes.

input	intermediate results	results
3,871,310	1,190,000	3,554
2,961,566	1,190,000	3,626
5,628,422	4,290,000	3,530
4,475,414	755,774	3,554
10,617,734	355,502	3,530
10,118,246	350,270	3,530
2,029,766	350,270	3,530
1,677,278	75,542	3,986
6,486,254	3,340,000	4,010
7,454,582	1,130,000	4,010
4,272,110	1,130,000	4,010
5,161,910	53,390	3,530
6,486,254	53,390	3,530
6,486,254	53,390	3,530
6,486,254	53,390	3,530
6,830,414	500,054	3,530
7,466,150	1,140,000	3,530
7,466,150	1,140,000	3,530
9,928,910	3,750,000	4,538
14,558,150	3,810,000	6,554
13,916,582	3,750,000	4,538
8,943,998	3,760,000	4,538
14,707,358	10,600,000	3,530
13,954,766	10,600,000	3,530
7,000,454	2,650,000	3,722
9,200,414	2,640,000	4,394
6,013,910	2,640,000	3,530
9,349,742	2,700,000	3,530
8,836,718	2,640,000	3,530
9,421,142	2,640,000	3,530
6,834,374	3,750,000	4,394
5,996,870	2,820,000	3,794
6,834,374	3,750,000	3,650
5,996,870	2,820,000	3,530
7,423,622	175,526	4,658
22,641,062	808,862	3,530
11,504,270	3,420,000	3,530
11,267,774	3,420,000	4,346
12,244,406	3,430,000	3,530
11,501,174	3,420,000	3,530
11,511,062	3,420,000	3,530
12,055,646	3,420,000	3,530
11,305,382	1,190,000	4,058
11,289,902	1,190,000	4,010
11,290,214	1,570,000	3,626
2,648,150	1,580,000	3,794
3,269,054	1,560,000	5,114
9,856,214	131,342	3,626
6,306,278	2,340,000	3,530
6,780,494	2,340,000	3,530

Table A.11: Space used to store the relations at each step of the algorithm of a Square Barbell query, in bytes.

input	intermediate results	results
8,126,226	584,798	18,474
4,753,842	502,526	9,450
8,126,226	609,158	17,610
6,086,130	499,502	8,970
5,062,098	502,022	9,450
5,982,954	503,174	9,450
5,175,138	502,838	14,634
8,329,674	498,782	7,722
4,930,938	498,134	7,722
9,010,938	498,134	7,722
12,949,146	514,766	8,970
5,836,050	498,398	7,722
8,287,530	2,628,850	25,770
14,254,410	757,046	8,298
8,287,530	2,628,850	8,970
8,861,082	2,086,770	7,914
7,897,530	2,026,550	8,682
10,653,906	2,060,820	7,914
10,653,906	2,061,830	7,914
16,620,786	2,315,800	7,914
10,653,906	2,060,820	7,914
10,653,906	2,053,890	10,794
9,863,994	2,048,700	8,970
10,653,906	2,082,090	7,914
10,653,906	2,060,820	7,914
10,653,906	2,061,830	7,914
10,653,906	2,061,490	10,794
9,584,226	2,027,460	7,914
9,863,994	2,056,670	8,970
10,653,906	2,053,890	10,794
9,863,994	2,066,730	7,914
15,857,730	6,221,290	7,722
12,263,562	6,229,650	7,722
8,008,338	6,212,100	7,722
16,615,818	7,115,920	7,722
19,222,914	6,402,490	7,722
16,463,010	10,600,700	9,930
17,383,866	10,638,500	9,930
24,350,058	10,670,400	9,930
18,621,426	10,580,200	9,930
6,592,818	26,870	8,202
11,321,010	2,797,210	8,682
20,554,602	1,124,630	7,914
25,699,482	13,479,400	9,546
25,699,482	13,676,800	9,546
25,699,482	13,479,400	9,546
13,040,802	1,506,090	7,722
5,019,642	1,425,420	7,722
13,214,946	1,756,550	7,722
13,077,618	1,468,240	11,178

Table A.12: Space used to store the relations at each step of the algorithm in a Pentagon Barbell query, in bytes.

input	intermediate results	results
10,569,982	139,966	29,050
10,569,982	139,966	29,050
9,125,686	139,558	24,442
14,887,198	174,646	30,970
9,946,750	177,286	30,970
3,852,382	498,646	24,442
8,169,430	498,646	24,442
20,681,806	504,838	24,442
8,868,382	525,862	24,442
11,427,598	498,646	24,442
17,119,822	1,025,710	28,282
19,726,078	1,049,970	24,826
21,991,246	1,247,690	24,826
19,172,614	1,043,110	24,442
17,526,574	664,318	55,162
17,529,286	664,318	55,162
20,759,206	664,342	81,274
6,087,406	541,198	28,666
9,282,694	765,550	35,194
12,891,934	471,238	48,250
9,072,742	471,790	52,090
8,948,182	403,726	33,274
9,072,742	423,406	33,274
12,884,686	755,110	24,826
12,834,526	755,110	24,826
12,844,270	755,110	24,826
13,325,974	755,110	24,826
13,489,318	5,113,270	30,202
14,247,406	5,113,270	30,202
13,018,294	5,113,270	30,202
21,213,598	5,113,270	29,050
34,823,590	504,454	24,442
8,085,430	591,070	34,426
12,834,142	571,078	24,442
20,721,190	727,366	24,442
16,117,750	5,113,340	45,562
14,888,638	5,113,340	45,562
15,045,238	5,113,370	46,330
14,888,638	5,113,340	43,258
23,083,942	5,113,340	43,258
9,911,374	401,398	24,442
9,911,374	408,670	156,922
14,005,414	471,790	156,922
9,885,814	420,550	24,442
17,824,606	458,614	157,306
20,613,790	5,115,620	33,274
12,975,190	669,670	30,202
18,296,806	10,580,300	42,490
21,551,278	10,580,300	30,970